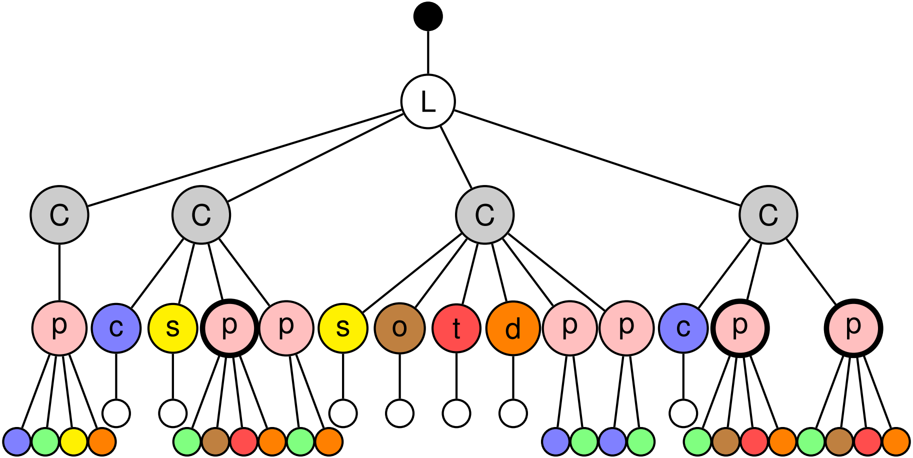
Example using multiple predicates

```
//performance[conductor][date]
```



Further examples with predicates

- `//performance[composer='Frederic Chopin']/composition` returns
  1. `<composition>Waltzes</composition>`
  2. `<composition>Piano Concerto No. 1</composition>`

Further examples with predicates

- `//performance[composer='Frederic Chopin']/composition` returns
  1. `<composition>Waltzes</composition>`
  2. `<composition>Piano Concerto No. 1</composition>`
- `//CD[@number="449719-2"]//composition` returns
  1. `<composition>Piano Concerto No. 1</composition>`
  2. `<composition>Piano Concerto No. 1</composition>`

  The two composition nodes have the same value, but they are different nodes

Functions

- XPath provides many functions that may be useful in predicates
- Each XPath function takes as input or returns one of these four types:
  - node set
  - string
  - Boolean
  - number

More about Context

- Each location step and predicate is evaluated with respect to a given *context*
- A specific context is defined as $(\langle N_1, N_2, \ldots N_m \rangle, N_c)$ with
  - a *context list* $\langle N_1, N_2, \ldots N_m \rangle$ of nodes in the tree
  - a *context node* N_c belonging to the list
- The *context length* m is the size of the context list
- The *context node position* $c \in [1, m]$ gives the position of the context node in the list

More about XPath Evaluation

- Each step s_i is interpreted with respect to a context; its result is a node list
- A step s_i is evaluated with respect to the context of step s_{i-1}
- More precisely:
  - for $i = 1$ (first step)
    if the path is absolute, the context is the root of the XML tree;
    else (relative paths) the context is defined by the environment;
  - For $i > 1$
    if $\mathcal{N} = \langle N_1, N_2, \ldots N_m \rangle$ is the result of step s_{i-1},
    step s_i is successively evaluated with respect to the context $(\mathcal{N}, N_j)$, for each $j \in [1, m]$
- The result of the path expression is the node list obtained after evaluating the last step

Node-set Functions

- *Node-set functions* operate on or return information about node sets
- Examples:
  - `position()`: returns a number equal to the position of the current node in the context list
    - `[position()=i]` can be abbreviated as `[i]`
  - `last()`: returns the size (i.e. the number of nodes in) the context list
  - `count(`*set*`)`: returns the size of the argument node *set*
  - `id()`: returns a node set containing all elements in the document with any of the specified IDs

Example about context

- The expression `//CD/performance[2]` returns the second performance *of each* CD, not the second of all performances
- The result of the step `CD` is the list of the 4 CD nodes
- The step `performance[2]` is evaluated once for each of 4 CD nodes in the context

Example about context (2)

- The result is the list comprising the second performance element child of each CD:

1.
```
<performance>
  <composition>Fantasias Op. 116</composition>
  <date>1976</date>
</performance>
```
2.
```
<performance>
  <composer>Franz Liszt</composer>
  <composition>Piano Concerto No. 1</composition>
</performance>
```
3.
```
<performance>
  <composition>American Suite</composition>
  <orchestra>Royal Philharmonic</orchestra>
  <conductor>Antal Dorati</conductor>
  <date>1984</date>
</performance>
```

Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The the following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!

Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The the following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!
- Saxon and Safari, e.g., correctly give the answer as (1) and (3) from the previous slide for all 4 expressions

Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The the following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!
- Saxon and Safari, e.g., correctly give the answer as (1) and (3) from the previous slide for all 4 expressions
- But, for `//CD/performance[date][2]`, eXist seems to return the second of all `performance` elements with a `date`

Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The the following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!
- Saxon and Safari, e.g., correctly give the answer as (1) and (3) from the previous slide for all 4 expressions
- But, for `//CD/performance[date][2]`, eXist seems to return the second of all `performance` elements with a `date`
- An earlier tool returned, for each `CD`, the second of its `performance` elements that had a `date` (if any)

More about the position() function

- `position()` is a function that returns the position of the current node in the context node set
- For most axes it counts forward from the context node
- For the "backward" axes it counts backwards from the context node
- The "backward" axes are: ancestor, ancestor-or-self, preceding, and preceding-sibling

Examples using position()

- So, to get the CD immediately before the one that was composed by Dvorak:
  `//CD[composer='Antonin Dvorak']/preceding::CD[1]`
- This selects the third CD
- To get the last CD (without having to know how many there are), use `//CD[position()=last()]`

Example using a different axis

- `//date/following-sibling::*` returns the following:
  1. ```
     <performance>
       <composer>Frederic Chopin</composer>
       <composition>Piano Concerto No. 1</composition>
     </performance>
     ```
  2. ```
     <performance>
       <composer>Franz Liszt</composer>
       <composition>Piano Concerto No. 1</composition>
     </performance>
     ```
- only one date element in the document has any following siblings

Examples using count

- `//CD[count(performance)=2]` returns CD elements with exactly two `performance` elements as children: the last 3 CDs

Examples using count

- `//CD[count(performance)=2]` returns CD elements with exactly two `performance` elements as children: the last 3 CDs
- `//CD[performance][performance]` of course does *not* do this:
  - it is equivalent to `//CD[performance]`
  - which returns CD elements with at least one `performance` child

More examples using count

- Assume we want the CDs containing only one `orchestra` element
- `//CD[count(orchestra)=1]` returns only one CD, where the orchestra is “London Symphony Orchestra”
- This is because we are counting the orchestra *children* of `CD` elements
- But orchestras are also represented below `performance` elements

More examples using count

- Assume we want the CDs containing only one `orchestra` element
- `//CD[count(orchestra)=1]` returns only one CD, where the orchestra is “London Symphony Orchestra”
- This is because we are counting the orchestra *children* of `CD` elements
- But orchestras are also represented below `performance` elements
- What about `//CD[count(//orchestra)=1]`?
  - But `//orchestra` is an absolute expression evaluated at the root
  - So the answer to `count(//orchestra)`is 4, not 1

More examples using count

- Assume we want the CDs containing only one `orchestra` element
- `//CD[count(orchestra)=1]` returns only one CD, where the orchestra is “London Symphony Orchestra”
- This is because we are counting the orchestra *children* of `CD` elements
- But orchestras are also represented below `performance` elements
- What about `//CD[count(//orchestra)=1]`?
  - But `//orchestra` is an absolute expression evaluated at the root
  - So the answer to `count(//orchestra)`is 4, not 1
- What we need is `/CD[count(.//orchestra)=1]`, where “.” represents the current context node
  - This gives us the CDs with the “Berlin Philharmonic” and “London Symphony Orchestra”

String Functions

- *String functions* include basic string operations
- Examples:
  - `string-length()`: returns the length of a string
  - `concat()`: concatenates its arguments in order from left to right and returns the combined string
  - `contains(`*s1*, *s2*`)`: returns true if *s2* is a substring of *s1*
  - `normalize-space()`: strips all leading and trailing whitespace from its argument

Boolean Functions

- *Boolean functions* always return a Boolean with the value true or false:
  - `true()`: simply returns true (makes up for the lack of Boolean literals in XPath)
  - `false()`: returns false
  - `not()`: inverts its argument (i.e., true becomes false and vice versa)

Boolean Functions

- *Boolean functions* always return a Boolean with the value true or false:
  - `true()`: simply returns true (makes up for the lack of Boolean literals in XPath)
  - `false()`: returns false
  - `not()`: inverts its argument (i.e., true becomes false and vice versa)
- Examples:
  - `//performance[orchestra][not(conductor)]` returns `performance` elements which have an `orchestra` child but no `conductor` child
  - `//CD[not(.//soloist)]` returns CDs containing no soloists

Boolean Functions (2)

- `boolean()`: converts its argument to a Boolean and returns the result
  - Numbers are false if they are zero or NaN (not a number)
  - Node sets are false if they are empty
  - Strings are false if they have zero length

Number Functions

- *Number functions* include a few simple numeric functions
- Examples:
  - `sum(set)`: converts each node in a node set to a number and returns the sum of these numbers
  - `round()`, `floor()`, `ceiling()`: round numbers to integer values

Summary

- XPath is used to navigate through elements and attributes in an XML document
- XPath is a major element in many W3C standards: XQuery, XSLT, XLink, XPointer
- It is also used to navigate XML trees represented in Java or JavaScript, e.g.
- So an understanding of XPath is fundamental to much advanced XML usage

Chapter 7

Optimising XPath Queries

Types of Optimisation

- In general, there are two types of query optimisation:
  - *logical* optimisation
  - *physical* optimisation
- Logical optimisation is concerned with, e.g., rewriting a given query to be *minimal* in size (i.e., to remove redundant parts)
- Physical optimisation refers to strategies to make query evaluation as efficient as possible
- In this chapter, we will study some aspects of logical optimisation for XPath
- Later chapters will discuss physical optimisation

XPath Fragment

- We will consider only a fragment of XPath
- Each location step is just
  - the name of an element, or
  - $*$, or
  - empty (giving rise to //)

  optionally followed by predicates

```xml
<bookstore>
 <book>
  <author><last-name>Abiteboul</last-name></author>
  <author><last-name>Hull</last-name></author>
  <author><last-name>Vianu</last-name></author>
  <title>Foundations of Databases</title>
  <isbn>0-201-53771-0</isbn>
  <price>26.95</price>
 </book>
 <magazine>
  <title>The Economist</title>
  <date><day>26</day><month>June</month><year>1999</year></date>
  <price>2.50</price>
 </magazine>
 <book>
  <isbn>0-934613-40-0</isbn>
  <price>34.95</price>
 </book>
</bookstore>
```

Some Queries on `bookstore`

On this specific document

- `/bookstore/book/isbn` gives the same result as `//isbn`
  - because every `isbn` is a child of `book` and every `book` is a child of `bookstore`
- `/bookstore/*/title` gives the same result as `/bookstore/(book|magazine)/title` and `//title`
  - because the only elements that can be children of `bookstore` and parents of `title` are either `book` or `magazine`
- `//magazine[date[day][month]]/title` gives the same result as `//magazine[date/day][date/month]/title`
  - because each `magazine` has only a single `date`

Some Queries on `bookstore`

On this specific document

- `/bookstore/book/isbn` gives the same result as `//isbn`
  - because every `isbn` is a child of `book` and every `book` is a child of `bookstore`
- `/bookstore/*/title` gives the same result as `/bookstore/(book|magazine)/title` and `//title`
  - because the only elements that can be children of `bookstore` and parents of `title` are either `book` or `magazine`
- `//magazine[date[day][month]]/title` gives the same result as `//magazine[date/day][date/month]/title`
  - because each `magazine` has only a single `date`

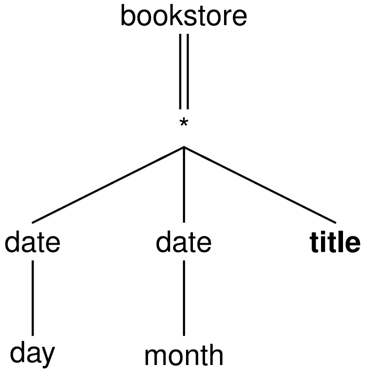
But these queries are *not* equivalent in general

XPath Queries as Tree Patterns

- We can view an XPath query *Q* in our fragment as a *tree pattern P*
- Each node test (element name or $*$) in *Q* becomes a node in *P*
- If *Q* has subexpression `A/B`, then nodes `A` and `B` in *P* are connected by a *single* edge
- If *Q* has subexpression `A//B`, then nodes `A` and `B` in *P* are connected by a *double* edge
- The node in *P* corresponding to the element name forming the output of *Q* is shown in boldface

Tree Pattern Example

`/bookstore//*[date/day][date/month]/title`



Containment and Equivalence of XPath Queries

- Given an XPath query Q and an XML tree t, the *answer* of evaluating Q on t is denoted by $Q(t)$
- For XPath queries P and Q, we say
  - P *contains* Q, written $P \supseteq Q$, if for all trees t, $P(t) \supseteq Q(t)$
  - P is *equivalent* to Q, written $P \equiv Q$, if $P \supseteq Q$ and $Q \supseteq P$
- Containment of XPath queries is useful
  - to show equivalence of queries for optimization
  - to determine if views can be used in query processing
  - to reuse cached query results

Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books

Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books

Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books
- `book` $\supseteq$ `book[price]`
  - There are no fewer books than books with prices

Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books
- `book` $\supseteq$ `book[price]`
  - There are no fewer books than books with prices
- `date[year]` $\supseteq$ `date[month][year]`
  - There are no fewer dates with years than dates with years and months

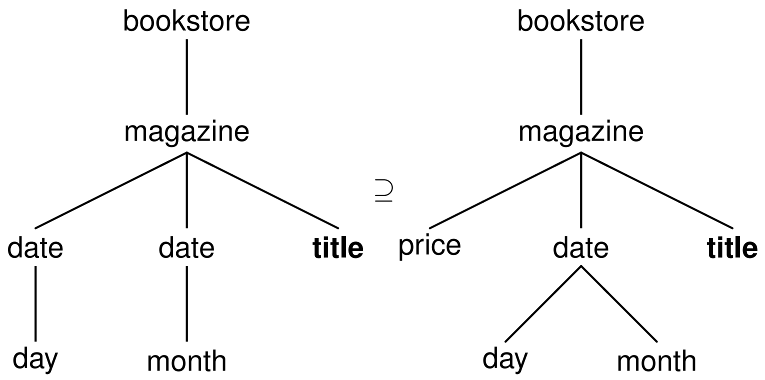
Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books
- `book` $\supseteq$ `book[price]`
  - There are no fewer books than books with prices
- `date[year]` $\supseteq$ `date[month][year]`
  - There are no fewer dates with years than dates with years and months
- `bookstore//title` $\supseteq$ `bookstore//book//title`
  - There are no fewer bookstores containing titles than bookstores containing books containing titles

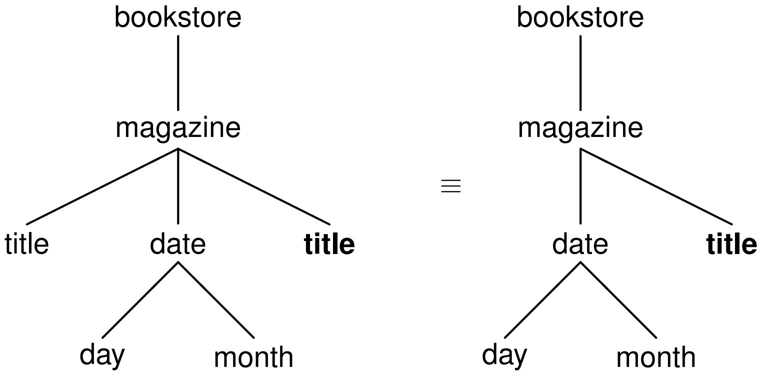
Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books
- `book` $\supseteq$ `book[price]`
  - There are no fewer books than books with prices
- `date[year]` $\supseteq$ `date[month][year]`
  - There are no fewer dates with years than dates with years and months
- `bookstore//title` $\supseteq$ `bookstore//book//title`
  - There are no fewer bookstores containing titles than bookstores containing books containing titles
- `magazine[date/year]` $\equiv$ `magazine[date/year][date]` so `[date]` is redundant

Example of Containment (tree patterns)



Example of Equivalence (tree patterns)



Using DTDs

- We can use DTDs to simplify expressions further
- Assume we know the document we want to query is valid with respect to a DTD *D*
- The DTD *D* specifies certain constraints
- e.g., every `book` element must have an `isbn` element as a child
- We already know that `/bookstore/book` $\supseteq$ `/bookstore/book[isbn]`
- Using the DTD *D*, we now know that `/bookstore/book` is *equivalent* to `/bookstore/book[isbn]`, but *only* when querying documents valid with respect to *D*

Constraints implied by a DTD

- Assume we are given the following DTD D (syntax simplified):

```
bookstore ((book|magazine)+)
book      (author*, title?, isbn, price)
author    (first-name?, last-name)
magazine  (title, volume?, issue?, date, price)
date      ((day?, month)?, year)
```

Constraints implied by a DTD

- Assume we are given the following DTD D (syntax simplified):

```
bookstore  ((book|magazine)+)
book       (author*, title?, isbn, price)
author     (first-name?, last-name)
magazine   (title, volume?, issue?, date, price)
date       ((day?, month)?, year)
```

- Some constraints implied by the DTD D:
  - every `author` element must have a `last-name` child (*child constraint*)
  - every `date` element with a `day` child must have a `month` child (*sibling constraint*)
  - every `book` element has at most one `title` child (*functional constraint*)

Examples

- `/bookstore/book[price]/author` is equivalent to `/bookstore/*/author` since
  - every book must have a price
  - book must be the parent of author

Examples

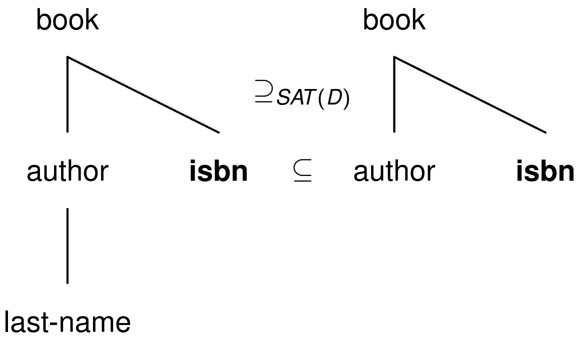
- `/bookstore/book[price]/author` is equivalent to `/bookstore/*/author` since
  - every book must have a price
  - book must be the parent of author
- `bookstore/book[author/first-name][author/last-name]` can first be rewritten as `bookstore/book[author/first-name][author]` and then as `book[author/first-name]`

Containment and Equivalence under DTDs

- We can use DTD constraints to find more equivalences
- When given a DTD D and a tree t known to satisfy D
- Let $SAT(D)$ denote the set of trees satisfying DTD D
- For XPath queries P and Q,
  - P *D-contains* Q, written $P \supseteq_{SAT(D)} Q$, if for all trees $t \in SAT(D)$, $P(t) \supseteq Q(t)$
  - P is *D-equivalent* to Q, written $P \equiv_{SAT(D)} Q$, if $P \supseteq_{SAT(D)} Q$ and $Q \supseteq_{SAT(D)} P$

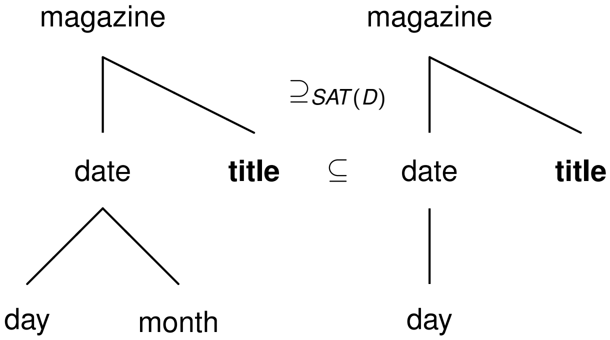
Example of *D*-Equivalence (Child Constraint)

- Every `author` must have a `last-name`



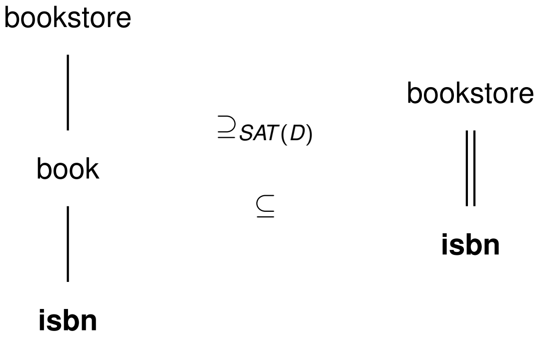
Example of *D*-Equivalence (Sibling Constraint)

- Every `date` with a `day` must have a `month`



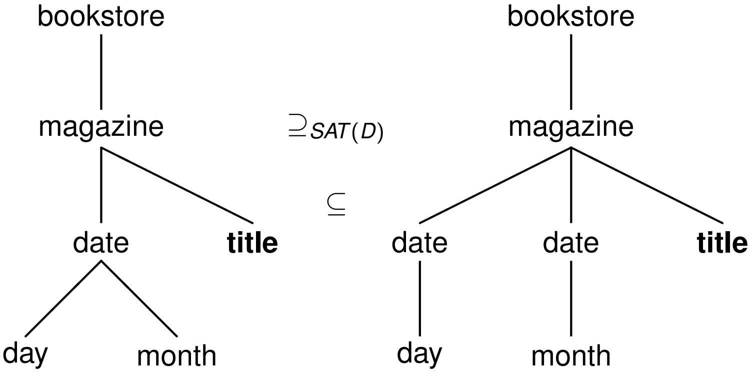
Example of *D*-Equivalence (Path Constraint)

- The only path from `bookstore` to `isbn` is through `book`



D-Equivalence Example (Functional Constraint)

- Every `magazine` has a single `date`



Summary

- We have considered logical optimisation of a fragment of XPath
- Can be used to delete redundant subexpressions from queries
- Further redundancies can be found when documents are valid with respect to a DTD
- We will consider efficient evaluation of XPath and some general physical optimisation techniques later