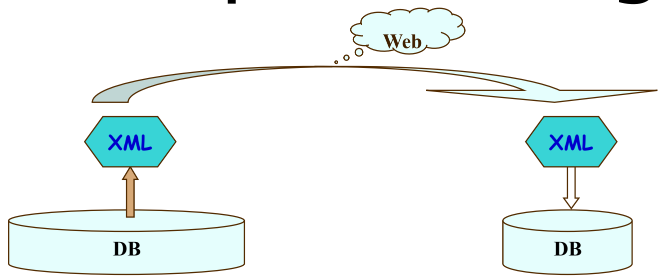
Querying and storing XML

Week 5
Publishing relational data as XML
February 12-15, 2013

XML publishing



- Exporting and importing XML data shared over Web
- Key problem: defining relational-XML views specifying mappings from relational to XML data sources
- Useful for querying shredded XML stored in RDBMSs
  - define reconstructing view, then translate queries on view to SQL

From relations to XML Views

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

Appears

mid	aid
11	1
11	2
32	2

From relations to XML Views

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

Appears

mid	aid
11	1
11	2
32	2

```
<Actor id="1">
 <LName>Maguire</LName>
 <FName>Tobey</FName>
 <Movie id="11">
  <Title>Spider-Man</Title>
  <Year>2002</Year>
 </Movie>
</Actor>
<Actor id="2">
 <LName>Dunst</LName>
 <FName>Kirsten</FName>
 <Movie id="11">
  <Title>Spider-Man</Title>
  <Year>2002</Year>
 </Movie>
 <Movie id="32">
  <Title>Elizabethtown</Title>
  <Year>1999</Year>
 </Movie>
</Actor>
```

From relations to XML Views

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

Appears

mid	aid
11	1
11	2
32	2

```
<Movie id="11">
 <Title>Spider-Man</Title>
 <Year>2002</Year>
 <Actor id="1">
  <LName>Maguire</LName>
  <FName>Tobey</FName>
 </Actor>
 <Actor id="2">
  <LName>Dunst</LName>
  <FName>Kirsten</FName>
 </Actor>
</Movie>
<Movie id="32">
 <Title>Elizabethtown</Title>
 <Year>1999</Year>
 <Actor id="2">
  <LName>Dunst</LName>
  <FName>Kirsten</FName>
 </Actor>
</Movie>
```


Commercial systems - canonical publishing

- Canonical publishing: the universal-relation approach
  - Embedding single SQL query in XSL stylesheet
  - Result: canonical XML representation of relations
- Systems:
  - Oracle 10g XML SQL facilities: SQL/XML, XMLGen
  - IBM DB2 XML Extender: SQL/XML, DAD
  - Microsoft SQL Server 2005: FOR-XML, XSD
- incapable of expressing practical XML publishing: default fixed XML document template

Canonical publishing

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

Appears

mid	aid
11	1
11	2
32	2

Canonical publishing

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

```
<Actor aid="1">
 <LName>Maguire</LName>
 <FName>Tobey</FName>
</Actor>
<Actor aid="2">
 <LName>Dunst</LName>
 <FName>Kirsten</FName>
</Actor>
```

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

```
<Movie mid="11">
 <Title>Spider-Man</title>
 <Year>2002</Year>
</Movie>
<Movie mid="32">
 <Title>Elizabethtown</title>
 <Year>2005</Year>
</Movie>
```

Appears

mid	aid
11	1
11	2
32	2

```
<Appears mid="11" aid="1"/>
<Appears mid="11" aid="2"/>
<Appears mid="32" aid="2"/>
```

Generating canonical XML view

- Goal: Push computation to DB
  - use DB sort or join to generate tuples in same order as needed in XML document
- Several approaches:
  - Redundant relation: join all relations relating parents to children
  - Unsorted path outer union: reduce redundancy
  - Unsorted outer union
  - Sorted outer union: single SQL query; best
- All approaches: require "tagging" post-processing stage to generate actual XML

Outer union query: example

```
((SELECT 1 AS tag, aid, lname, fname, NULL, ... NULL
 FROM Actors)
UNION
(SELECT 2 AS tag, NULL,..., mid, title, year, NULL...
 FROM Movies)
UNION
(SELECT 3 AS tag, NULL, ..., aid, mid
 FROM Appears))
ORDER BY tag, aid, mid, ...
```


XPERANTO
[Shanmagusundaram et al.]

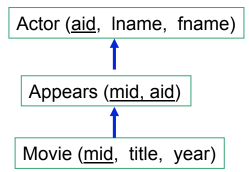
- Commercial system: IBM DB2 XML extender, SQL/XML
- Middleware (vendor-independent): XPERANTO
- Extends SQL with XML constructors:

  `select` XML-aggregation

  `from` R1, . . ., Rn

  `where` conditions
- XML constructors (XML-aggregation): functions
  - Input: tables and XML trees (forest)
  - Output: XML tree

XML publishing with XPERANTO (SQL/XML)




```
<Actor>
 <LName>Maguire</LName>
 <FName>Tobey</FName>
 <Movie>
  <Title>Spider-Man</Title>
  <Year>2002</Year>
 </Movie>
</Actor>
...
```

- Extended SQL:

```
select XMLAGG(ACTOR(lname, fname,
                    select   XMLAGG ( MOVIE(title,   year)
                    from     Appears Ap, Movies M
                    where    Ap.aid = A.aid and Ap.mid = M.mid
                    group    order by  A.lname, A.fname ))
 from      Actor  A
```

XML constructors (SQL/XML)

- Actor constructor:

```
create function ACT(lname: str, fname: str, mlist: XML)
         <Actor>
          <Lname>{lname}</Lname>
          <Fname>{fname}</Fname>
          {mlist}
        </Actor>
```

- Movie constructor (mlist)

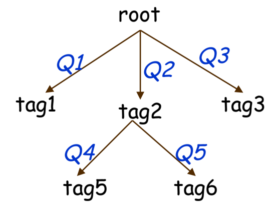
```
create function Mov(title: str, year: int)
    <Movie year="{year}">{title}</Movie>
```

- Verbose and cumbersome
    - small document: tedious
    - large documents: unthinkable

SilkRoute

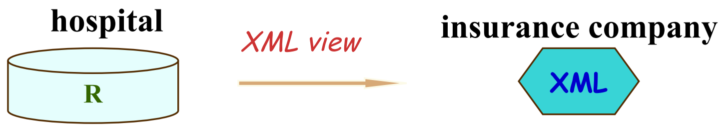
[Fernandez et al. 2002]

- Annotated template: embedding SQL in a fixed XML tree



- Middleware: SilkRoute
- Commercial: SQL Server 2005 XSD, IBM DB2 DAD
- Advantages:
    - More `modular' compared to the universal relation approach
    - Limited schema-driven: conforming to a fixed doc template

Data exchange: insurance company and hospital



- Daily report
- Relational database R at the hospital:

```
Patient (SSN, name, tname, policy#, date)
inTreatment (tname, cost)
outTreatment (tname, referral#)
Procedure (tname1, tname2)
```

- treatment
    - in hospital: composition hierarchy in Procedure
    - outside of the hospital: referral#

Example: insurance company and hospital

- DTD *D* predefined by the insurance company:

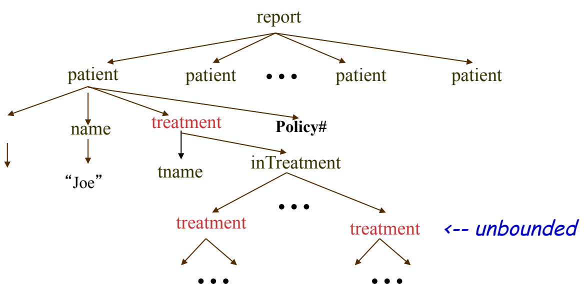
```
report       →   patient*
patient      →   SSN,  pname,  treatment,  policy#
treatment    →   tname,(inTreatment + outTreatment)
inTreatment  →   treatment*
outTreatment →   referral#
```

- How to define a mapping σ such that for any instance DB of R,
    - σ (DB) is an XML document containing all the patients and their treatments (hierarchy, referral#) from DB, and
    - σ (DB) conforms to D?

Challenge: recursive types

- XML data: unbounded depth -- cannot be decided statically

```
treatment    → tname, (inTreatment + outTreatment)
inTreatment  → treatment*    --- recursive
```

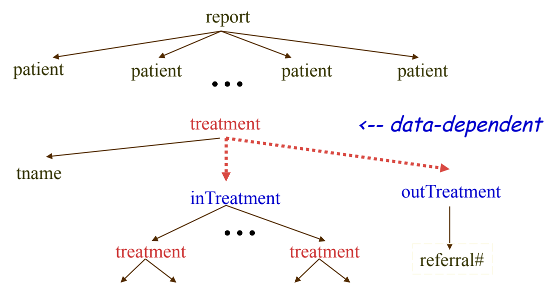


Challenge: non-determinism

- The choice of a production (element type definition)

```
treatment → tname,  (inTreatment + outTreatment)
```

  - depends on the underlying relational data



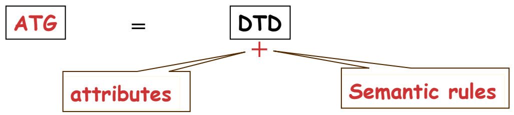
Limitations of existing systems

- uses fixed XML tree template or ignores DTD-conformance
  - middleware: SilkRoute (AT&T), XPERANTO (IBM), …
  - systems: SQL Server 2005, IBM DB2 XML extender, …
  - incapable of coping with a predefined DTD (e.g. recursion)
- type checking: define a view and then check its conformance
  - undecidable in general, co-NEXPTIME for extremely restricted view definitions (but cf. week 7)
  - no guidance on how to define XML views that typecheck
- one gets an XML view that typechecks only after repeated failures and with luck

Schema-directed XML publishing

Attribute Translation Grammar (ATG)

[Benedikt et al. 2002]



- DTD: normalized; element type definitions a → r where:

  r ::= PCDATA | ε | $a_1, \ldots, a_n$ | $a_1 + \ldots + a_n$ | a*

- Attributes: $a associated with each element type a
  - $a: tuple-valued, to pass data value as well as control info
- Rules: associated with each a → r:
  - for each b in r, define $b := Q($a)
  - SQL query Q extracts data from DB
  - tuple-valued parent attribute $a is a parameter in Q

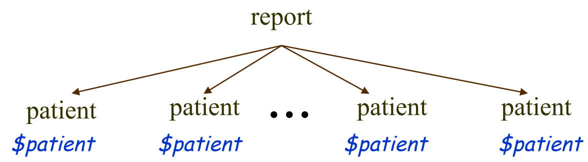
Semantics: conceptual evaluation

- Top-down

  `report → patient*`

```
$patient ← select   SSN, name, tname, policy
           from     Patient  ---  SQL query
```

- recall `Patient (SSN, name, tname, policy#)`
- Data-driven: one patient element **for each tuple** in `Patient` relation



Inherited attributes

- Inherited: $child is computed using $parent

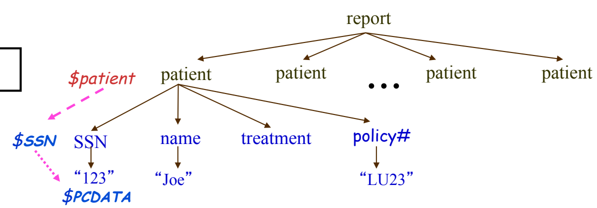
  `patient → SSN, name, treatment, policy#`

```
$SSN        ← $patient.SSN
$name       ← $patient.name
$treatment  ← $patient.tname
$policy#    ← $patient.policy
```

- recall `$patient = (SSN, name, tname, policy)`

  `SSN → PCDATA`

```
$PCDATA ← $SSN
```

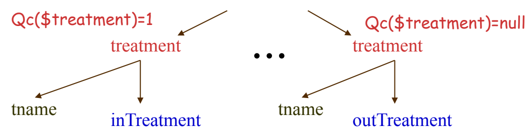


Coping with non-determinism

`treatment → tname, (inTreatment + outTreatment)`

```
$tname ← $treatment
($inTreatment, $outTreatment) ← case  Qc($treatment).tag
                                 1:     ($treatment, null)
                                 else:  (null, $treatment)
```

- Qc: `SELECT 1 as tag FROM inTreatment WHERE tname = $treatment`
  - conditional query: the choice of production
  - $parent a parameter in SQL query

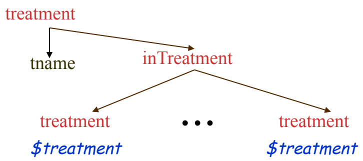


Coping with recursion

```
inTreatment  → treatment*
```

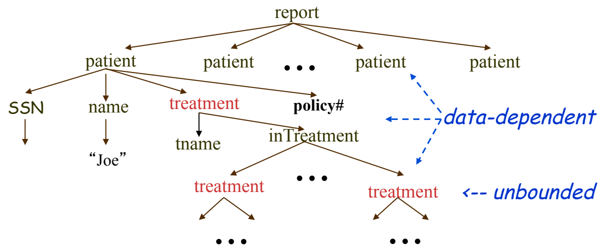
```
$treatment ← select tname2
             from  Procedure
             where $inTreatment = tname1
```

- recall `Procedure (tname1, tname2)`
  - `$parent` as constant parameter in SQL query Q
  - inTreatment is further expanded as long as Q(DB) is nonempty



DTD-directed publishing with ATGs

- DTD-directed: the XML tree is constructed strictly following the productions of a DTD
  - guaranteed DTD conformance
- Data-driven: the choice of productions and expansion of the XML tree (recursion) depends on relational data
  - static analysis to guarantee termination



ATGs vs. existing systems

- DTD-conformance:
  - ATGs: provide guidance for how to define DTD-directed publishing
  - Other systems: based on a fixed tree template
- Expressive power: strictly more expressive than others
  - ATGs: capable of expressing XML views supported by other systems
  - Other systems: cannot handle recursion/ nondeterminism

Quiz: Fill in blanks

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

Appears

mid	aid
11	1
11	2
32	2

```
doc -> Actor*
```

$Actor := ...

```
Actor -> id,lname,fname,Movies
```

$id := ... $lname := ...
$fname := ... $Movie := ...

```
Movies -> Movie*
```

$Movie := ...

```
Movie -> id,title,year
```

$id := ... $year = ...
$title := ...

Quiz: Fill in blanks

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

Appears

mid	aid
11	1
11	2
32	2

```
doc -> Actor*
```

```
$Actor := select aid,lname,fname
          from Actors
```

```
Actor -> id,lname,fname,Movies
```

$id := ... $lname := ...
$fname := ... $Movie := ...

```
Movies -> Movie*
```

$Movie := ...

```
Movie -> id,title,year
```

$id := ... $year = ...
$title := ...

Quiz: Fill in blanks

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

Appears

mid	aid
11	1
11	2
32	2

```
doc -> Actor*
```

```
$Actor := select aid,lname,fname
          from Actors
```

```
Actor -> id,lname,fname,Movies
```

```
$id := $Actor.aid       $lname := $Actor.lname
$fname := $Actor.fname $Movies := $Actor.aid
```

```
Movies -> Movie*
```

$Movie := ...

```
Movie -> id,title,year
```

$id := ... $year = ...
$title := ...

Quiz: Fill in blanks

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

Appears

mid	aid
11	1
11	2
32	2

```
doc -> Actor*
```

```
$Actor := select aid,lname,fname
          from Actors
```

```
Actor -> id,lname,fname,Movies
```

```
$id := $Actor.aid       $lname := $Actor.lname
$fname := $Actor.fname $Movies := $Actor.aid
```

```
Movies -> Movie*
```

```
$Movie := select m.mid,m.title,m.year
          from movies m, appears app
          where app.aid=$Movie.aid, app.mid=m.mid
```

```
Movie -> id,title,year
```

$id := ... $year = ...
$title := ...

Quiz: Fill in blanks

Actors

aid	lname	fname
1	Maguire	Tobey
2	Dunst	Kirsten

Movies

mid	title	year
11	Spider-Man	2002
32	Elizabethtown	2005

Appears

mid	aid
11	1
11	2
32	2

```
doc -> Actor*
```

```
$Actor := select aid,lname,fname
          from Actors
```

```
Actor -> id,lname,fname,Movies
```

```
$id := $Actor.aid       $lname := $Actor.lname
$fname := $Actor.fname $Movies := $Actor.aid
```

```
Movies -> Movie*
```

```
$Movie := select m.mid,m.title,m.year
          from movies m, appears app
          where app.aid=$Movie.aid, app.mid=m.mid
```

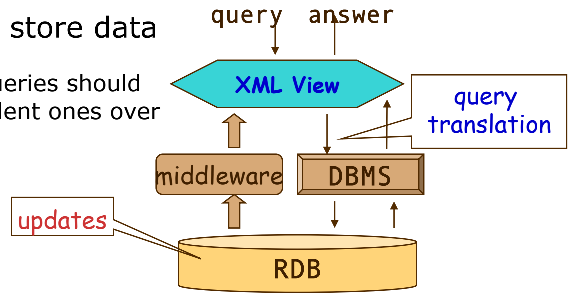
```
Movie -> id,title,year
```

```
$id := $Movie.mid
$year = $Movie.year
$title := $Movie.title
```

Querying XML views of relational data

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

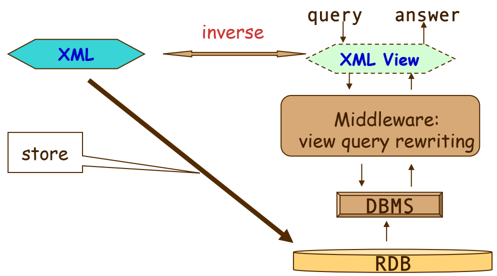


Virtual vs. materialized

- 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: multiple views of the same databases are supported simultaneously for different user groups

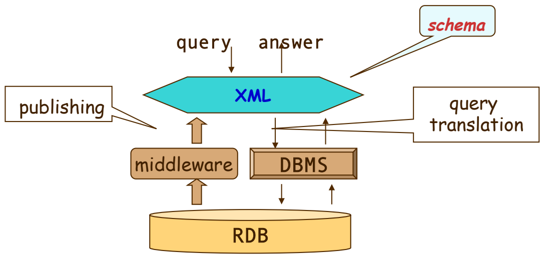
Middleware approach

- Query answering/rewriting with views has been extensively studied
  - Define publishing mapping that inverts shredding
  - Treat the inverse as a virtual view of the relational data
  - Make use of techniques for view query rewriting



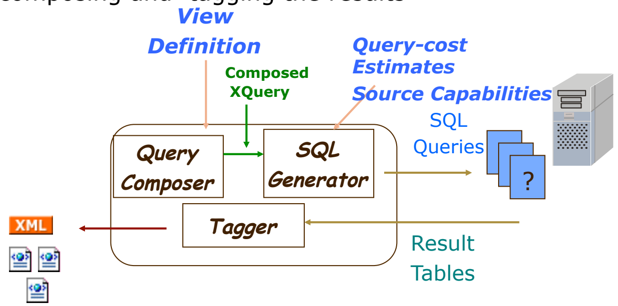
Challenging issues

- Schema-directed XML view definition: we know how to do it now
- Query translation: given a query over a virtual XML view, rewrite the query to an equivalent one over the underlying database – from views to relations



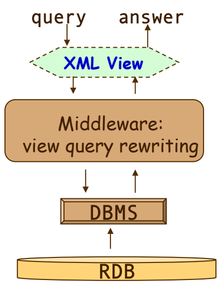
Querying a View

- The middleware must respond to view queries/ requests by
  - generating SQL queries/requests at runtime
  - composing and tagging the results



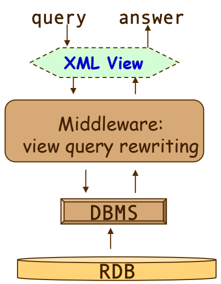
The XPERANTO approach

- Middleware:
  - A library of XQuery functions
  - Transformation rules (algebra)
  - A cost model and optimization techniques
- An intermediate language -- middleware
  - accept an XML query
  - rewrite the query with the rules and library
    - push work down to the underlying DBMS
    - conduct computations that DBMS cannot do
  - optimize the rewritten queries
  - compose query results from DBMS and middleware to build the answer to the XML query



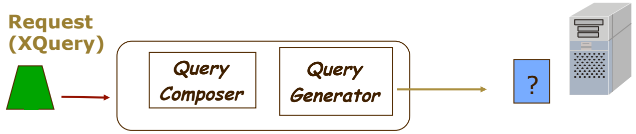
Research issues

- Optimization: can one effectively find an optimal rewriting?
- How much work should be pushed down to DBMS?
  - communication cost between DBMS and the middleware
  - leveraging the DBMS optimizer
- Multi-query optimization – hard
  - accurate cost model?
  - composition – when to tag?
  - query dependency
  - workload
- Effective generic optimization techniques are beyond reach for Turing-complete query languages
  - (NP-hard at least, can easily become undecidable)



The SilkRoute Approach

- Uses XQuery to specify view
  - Advantage: easy to compose query with view

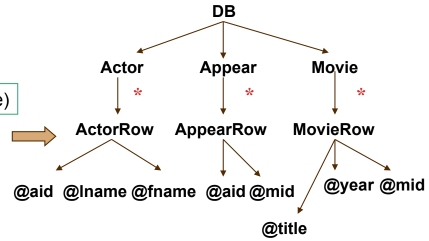


- Running example:
  - relations → canonical XML

Actor (<u>aid</u>, lname, fname)

Appearance (<u>mid, aid</u>)

Movie (<u>mid</u>, title, year)



Query Composition

View: Movies by Gibson

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return
 <Actor><Fname> Mel </Fname> <Lname> Gibson </Lname>
  for $actapp in DB//AppearRow[@aid=$aid]
   for $movie in DB//MovieRow[@mid=$actapp/@mid]
   return <Movie year="{$movie/@year}"> {$movie/@title} </Movie>
  </Actor>
```

Query: Get each Actor + Movies in 1999

```
for $act in //Actor return <Actor> {$act/Lname}
          for $movie in $act/Movie[@year=1999]
          return <Movie>{$movie}</Movie>
        </Actor>
```

Composed Query: Movies by Gibson in 1999

Query Composition

View: Movies by Gibson

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return
  <Actor><Fname> Mel </Fname> <Lname> Gibson </Lname>
  for $actapp in DB//AppearRow[@aid=$aid]
   for $movie in DB//MovieRow[@mid=$actapp/@mid]
   return <Movie year="{$movie/@year}"> {$movie/@title} </Movie>
  </Actor>
```

Query: Get each Actor + Movies in 1999

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return <Actor> {$act/Lname}
          for $movie in //Movie[@year=1999]
          return <Movie>{$movie}</Movie>
        </Actor>
```


Query Composition

View: Movies by Gibson

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return
  <Actor><Fname> Mel </Fname> <Lname> Gibson </Lname>
  for $actapp in DB//AppearRow[@aid=$aid]
   for $movie in DB//MovieRow[@mid=$actapp/@mid]
   return <Movie year="{$movie/@year}"> {$movie/@title} </Movie>
  </Actor>
```

Query: Get each Actor + Movies in 1999

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return <Actor> Gibson
  for $movie in //Movie[@year=1999]
  return <Movie>{$movie}</Movie>
     </Actor>
```

Query Composition

View: Movies by Gibson

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return
  <Actor><Fname> Mel </Fname> <Lname> Gibson </Lname>
   for $actapp in DB//AppearRow[@aid=$aid]
    for $movie in DB//MovieRow[@mid=$actapp/@mid]
    return <Movie year="{$movie/@year}"> {$movie/@title} </Movie>
  </Actor>
```

Query: Get each Actor + Movies in 1999

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return <Actor> Gibson
  for $actapp in DB//AppearRow[@aid=$aid]
  for $movie in DB//MovieRow[@mid=$actapp/@mid and @year=1999]
          return <Movie>{$movie}</Movie>
          </Actor>
```


Query Composition

Composed Query on Canonical XML:=

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return
  <Actor>Gibson
   for $actapp in DB//AppearRow[@aid=$aid]
    for $movie in DB//MovieRow[@mid=$actapp/@mid and @year=1999]
    return <Movie> {$movie/@title} </Movie>
  </Actor>
```

- Efficient query composition involves:
  - substitution
  - filtering
  - pattern matching

Generating SQL

Composed Query on Canonical XML:=

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return
  <Actor>Gibson
   for $actapp in DB//AppearRow[@aid=$aid]
    for $movie in DB//MovieRow[@mid=$actapp/@mid and @year=1999]
    return <Movie> {$movie/@title} </Movie>
  </Actor>
```


Generating SQL

Composed Query on Canonical XML:=

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return
  <Actor>Gibson
   for $actapp in DB//AppearRow[@aid=$aid]
    for $movie in DB//MovieRow[@mid=$actapp/@mid and @year=1999]
    return <Movie> {$movie/@title} </Movie>
  </Actor>
```

```
SELECT a.aid, app.mid
FROM   Actors a, Appear app
WHERE  app.aid = a.aid
  AND  a.lname = 'Gibson'
```

Generating SQL

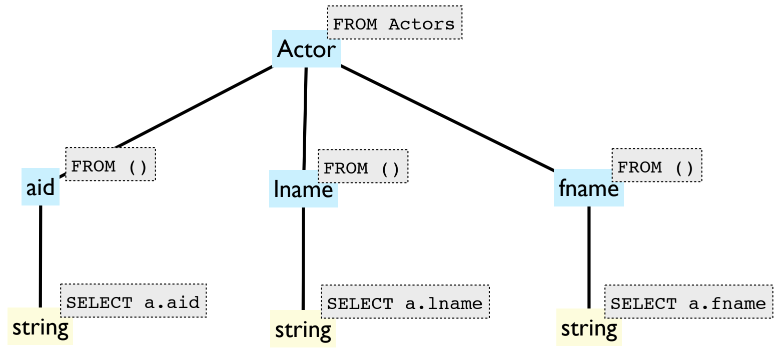
Composed Query on Canonical XML:=

```
for $aid in DB//ActorRow[@lname="Gibson"]/@aid
return
  <Actor>Gibson
   for $actapp in DB//AppearRow[@aid=$aid]
    for $movie in DB//MovieRow[@mid=$actapp/@mid and @year=1999]
    return <Movie> {$movie/@title} </Movie>
  </Actor>
```

```
SELECT a.aid, app.mid, m.title
FROM   Actor a, Appears app, Movie m
WHERE  app.mid = m.mid
   AND  m.year = 1999
   AND  a.lname = 'Gibson'
   AND  a.aid = app.aid
```


SilkRoute: Query trees

- Tree annotated with SQL clauses

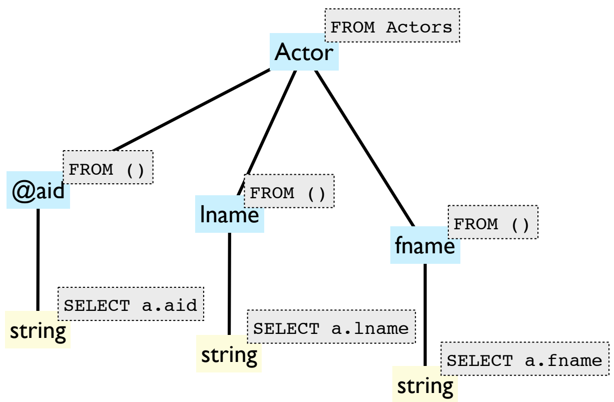


Canonical view (similar for Movies, Appears)

SilkRoute: Query composition

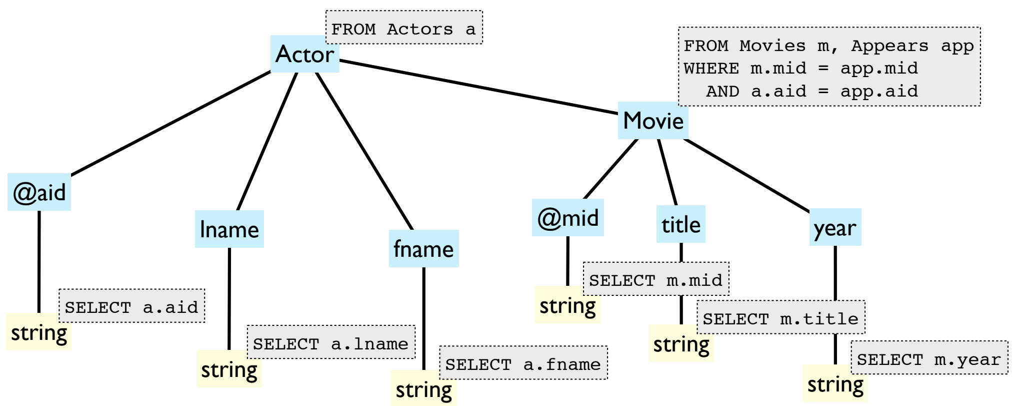
- Can *compose* XQuery queries to form new trees

```
for $a in Actor
return
 <Actors @aid={$x/aid}>
  <lname>{$a/lname}</lname>
  <fname>{$a/fname}</fname>
  {for $app in Appears[@aid=$a/aid]
   for $m in Movie[@mid=$app/mid]
   return
    <Movie mid={$m/mid}>
     <Title>{$m/title}</Title>
     <Year>{$m/year}</Year>
    </Movie>}
 </Actors>
```



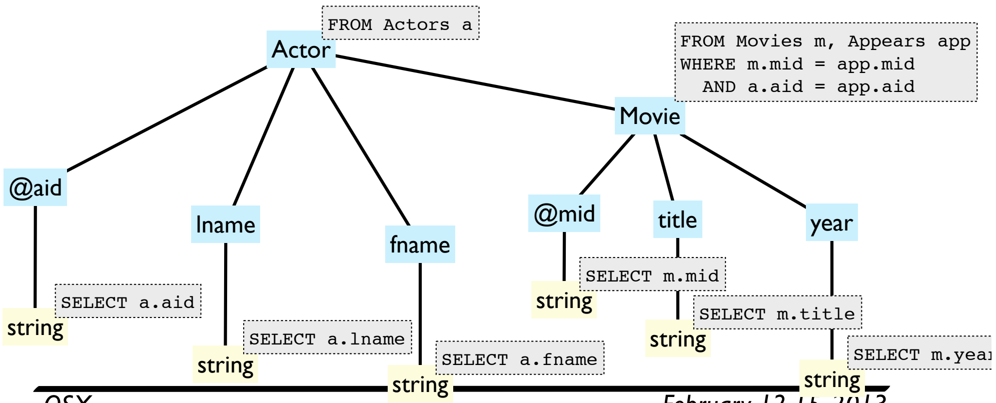
SilkRoute: Composition

- A composed query tree



SilkRoute: evaluation

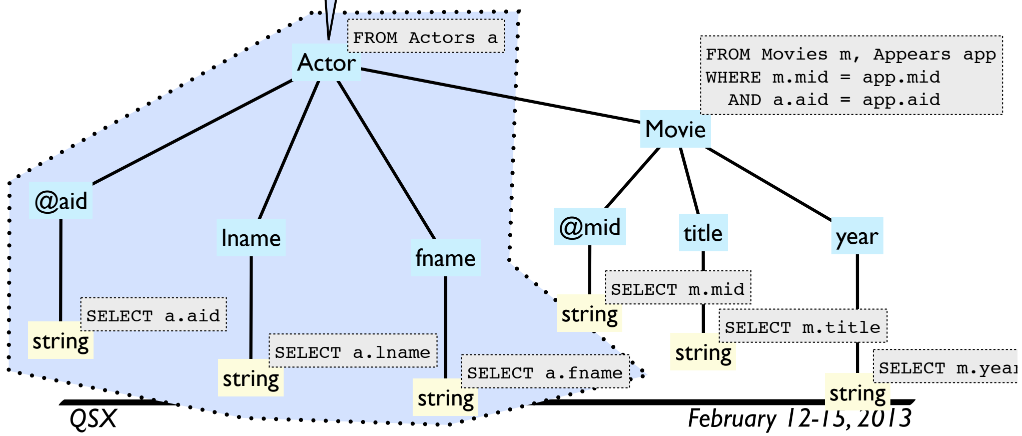
- Compose SQL fragments along path
- (Similar to / can translate to ATGs)



SilkRoute: evaluation

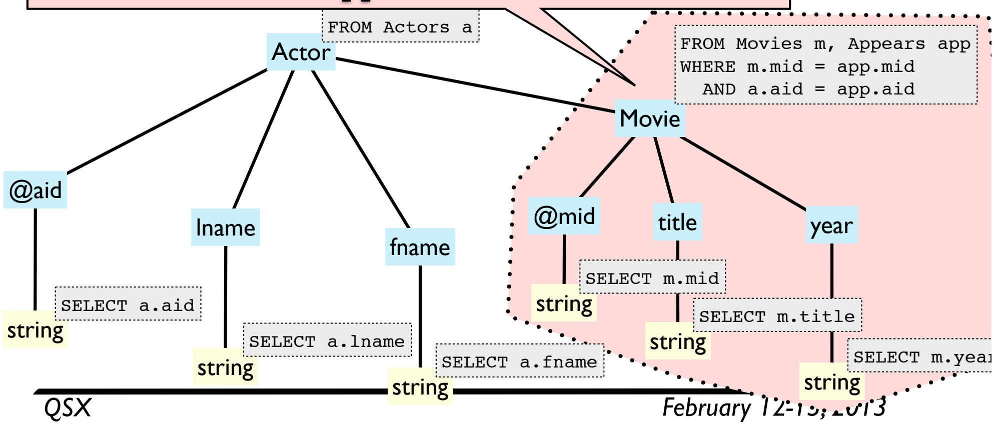
```
SELECT a.aid, a.lname, a.fname
FROM   Actors a
```

- Compose SQL fragments along path
- (Similar to / can translate to ATGs)



SilkRoute: evaluation

```
SELECT a.aid, m.mid, m.title, m.year
FROM   Actors a, Movies m, Appears app
WHERE  m.mid = app.mid
  AND  a.aid = app.aid
```



Query translation

- Works, but several challenges:
  - can we guarantee Q produces data matching D?
  - efficiency: if we materialize result, how to **recompute** when relational data updated?
  - can we translate Q ○ V to an **efficient** query/query plan?
  - how can we translate **updates** to Q(V(DB)) back to DB?
- Complications:
  - recursion (in query or DTD)
  - typechecking (XQuery typechecking intractable/ undecidable)

Commercial RDBMS support for XML storage/publishing/views

IBM DB2 XML Extender (storage)

- XML Columns: CLOBs + side tables for indexing individual elements
  - SQL/XML: an extension of SQL with XML constructors (XMLAGG, XMLELEMENT, etc) as discussed earlier
- XML Collections: Declarative decomposition of XML into multiple tables
  - Data loading: follows DAD mapping
  - Able to incrementally update existing tables (DB2)
  - Nonrecursive schema only

IBM DB2 XML Extender (publishing)

- User-defined mapping through DAD (Document Access Definition)
  - a fixed XML tree template (nonrecusive)
  - SQL mapping: a single SQL query, constructing XML trees of depth bounded by the arity of the tuples returned and group-by
  - RDB node mapping: a fixed tree template with nodes annotated with conjunctive queries
- Summary:
  - incapable of supporting schema-directed publishing
  - can't define recursive XML views

MS SQL Server 2005 (storage)

- CLOB (character large objects), XML data type
- XQuery: query(), value(), exist(), nodes(); binding relational data
  - Combine INSERT and node( ), value( ), XPath
  - OPENXML: access to XML data as a relational rowset
- selective shredding, limited recursion, can't store the entire document in a single pass

MS SQL Server 2005 (publishing)

- Annotated schema (XSD): fixed tree templates
  - **nonrecursive** schema
  - associate elements and attributes with table and column names
  - Given a relational database, XSD populates an XML elements/ attributes with corresponding tuples/columns
- FOR-XML
  - An extension of SQL with an FOR-XML construct
  - Nested FOR-XML to construct XML documents
- Summary:
  - incapable of supporting schema-directed publishing
  - can't define recursive XML views (bounded recursion depth)

Oracle 10g XML DB (storage)

- Store XML data in CLOB (character large objects) or tables
- Canonical mapping into object-relational tables
  - tag names are mapped to column names
  - elements with text-only map to scalar columns
  - elements with sub-elements map to object types
  - list of elements maps to collections
  - Indexing: standard relational
  - cannot insert into existing tables (DB2)
- Annotated schema: recursive, selective

Oracle 10g XML DB (publishing)

- SQL/XML
- DBMS_XMLGEN, a PL/SQL package
  - Supports recursive XML view definition (via linear recursion of SQL'99)
  - does not support schema-directed XML publishing

Commercial Systems: Summary

- Storage and XML-relational mappings:
  - CLOBs (or XML columns)
  - Fixed canonical mappings
  - Mappings in terms of annotated schema
- Querying:
  - SQL as the main access method to XML documents
  - "XML-aware" extensions to SQL
- Limited support for
  - recursive schema (Microsoft, IBM DB2)
  - incrementing/updating existing tables (Oracle)
  - XQuery, updates
  - context-dependent tuple construction

Update Support

- How to update?
  - Flat streams: overwrite document
  - Colonial: SQL updates?
  - Native: DOM, proprietary APIs
- But how do you know you have not violated schema?
  - Flat streams: re-parse document
  - Colonial: need to understand the mapping and translate/ maintain integrity constraints
  - Native: supported in some systems (e.g., eXcelon)
- XQuery Update Facility: relatively new

Next time

- No lectures next week!
  - Innovative learning week
- After that: XML Updates
  - XQuery Update for the impatient
  - Updating XML
  - Updating XML Views of Relations
- Reading: Monday 4pm (as usual)