A Normal Form for XML Documents

- Overview of Relational Database Design Process
  - Functional Dependencies and Normalization
    - functional dependencies (FDs)
    - redundancy and update anomalies
    - third normal form (3NF) and Boyce-Codd normal form (BCNF)
    - design algorithms for 3NF and BCNF
  - Nested Normal Form for nested relations
- Normal Form for XML docuemnts
  - redundancy and update anomalies for XML docuemnts
  - functional dependencies
  - XNF: a normal form for XML documents
  - a design algorithm for XNF

This section is based on the paper A Normal Form for XML Documents by M. Arenas L. Libkin in Proceedings of ACM PODS02.

A motivation Example for Normal Form Relations

course	**title**	**student_id**	**Name**	**Major**	**Grade**
391	Database	1234	Sarah	CS	9
391	Database	4321	Tom	CS	8
391	Database	2345	Bill	CS	7
201	Program	1234	Sarah	CS	6
201	Program	2345	Bill	CS	5

Motivation Example

StudentCourse = (course, title, student_id, name, major, grade)

Student = (student_id, name, major)
Course = (Course, title)
Registration = (course, student_id, grade)

Functional Dependencies

- Functional dependencies (FDs)

  Let R be a relation scheme, and $X \subseteq R$ and $Y \subseteq R$ be sets of attributes. Then the functional dependency

  $$X \rightarrow Y$$

  holds on R if in any legal relation r, for all pairs of tuples t1 and t2 in r

  $$t_1[X] = t_2[X] \quad \Rightarrow \quad t_1[Y] = t_2[Y].$$

Example: student_id → name
course, student_id --> grade

Desirable Properties of Decomposition

- Minimizing redundancy
  - Boyce-Codd normal form
  - third normal form

Boyce-Codd Normal Form

- A relation scheme R is said to be in Boyce-Codd normal form (BCNF) if for any non-trivial FD $X \rightarrow A$ which holds in R, X is a key of R, that is, $X \rightarrow A$ holds in R.
  - no partial redundancy
  - no transitive redundancy
- Let U be a set of attributes, F be a set of FDs, and D = {R1, ..., Rn} be a decomposition of U. Then D is said to be a BCNF decomposition of U with respect to F if
  - R is a join loss-less decomposition of U wrt F, and
  - every relation scheme Ri in D is in BCNF wrt F.

- Example

  Regist (Course#, Student#, Grade, Address, Phone)

  is not in BCNF since

  Student# $\rightarrow$ Address holds but Student # is not a key

  Course (Course#, Prof, Office, Phone)

  is not in BCNF because Prof $\rightarrow$ Office holds but Prof is not a key

But

{ (Course#, Student#, Grade), (Student#, Adress, Phone)} is a BCNF decomposition of Regist.

{ (Course#, Porf), (Prof, Office, Phone)} is a BCNF decomposition of Course.

Algorithm for BCNF decomposition

Input U: a set of attributes
F: a set of FDs
Output D: a BCNF decomposition of U wrt F
Method

(1) D = {U};
(2) **while** there exists a relation scheme Q in D that is not in BCNF **do**
begin
find a nontrivial FD $X \rightarrow W$ that violates BCNF, i.e.,
$X \rightarrow W$ in F^+ and $XW \subseteq Q$ and $X \not\rightarrow Q$;
$X^* := \{ A \mid A \text{ is in } (Q - X) \text{ and } F \models X \rightarrow A \}$;
replace Q in D by two schemes $(X \cup X^*)$ and $(Q - X^*)$
end;

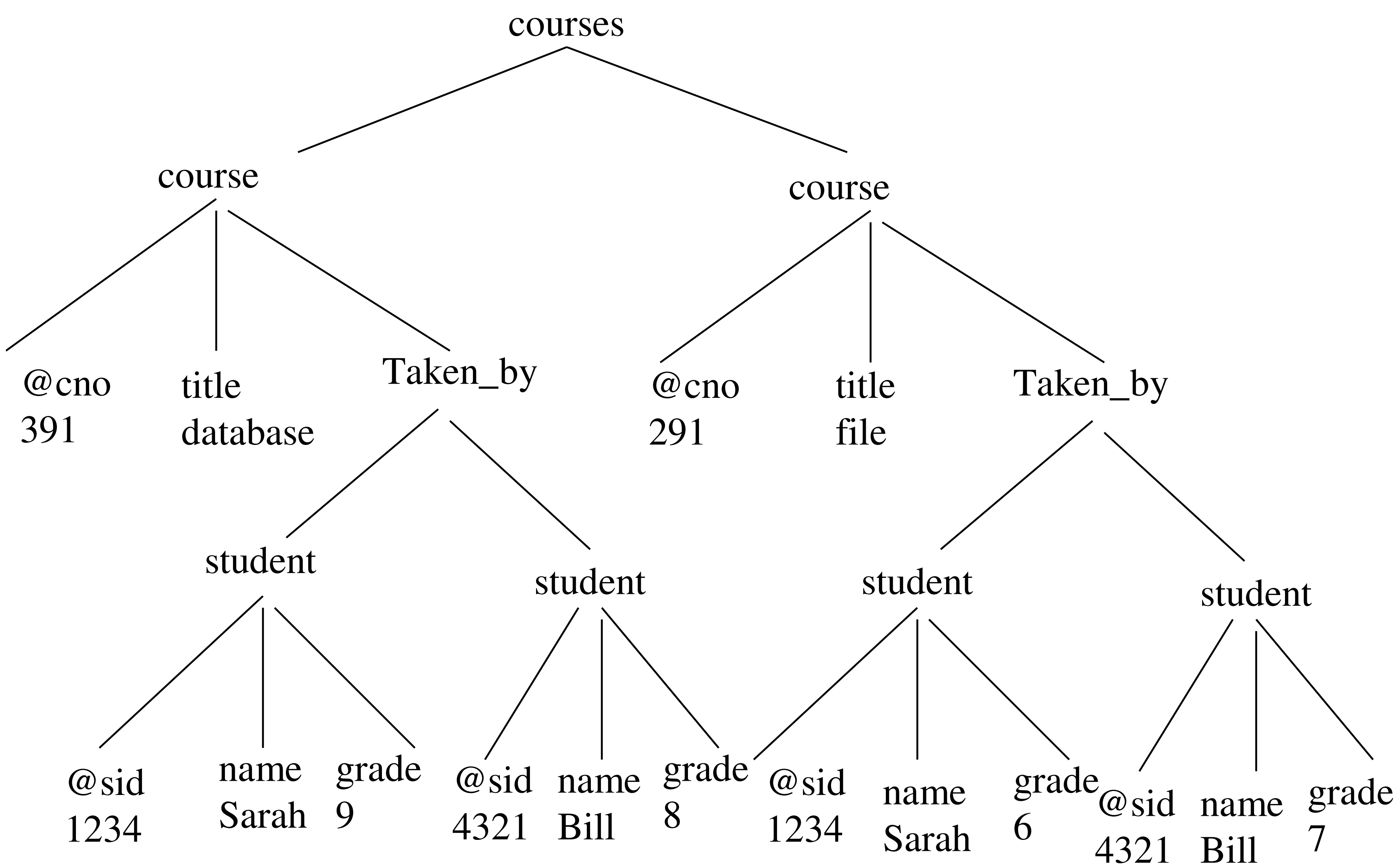
Note that it is NP-complete to determine whether a relation scheme is in BCNF wrt F.

NNF: A Normal Form for Nested Relations

- Functional dependency and multi-valued dependencies
- Path Attributes
- Minimizing redundancy and update anormalies

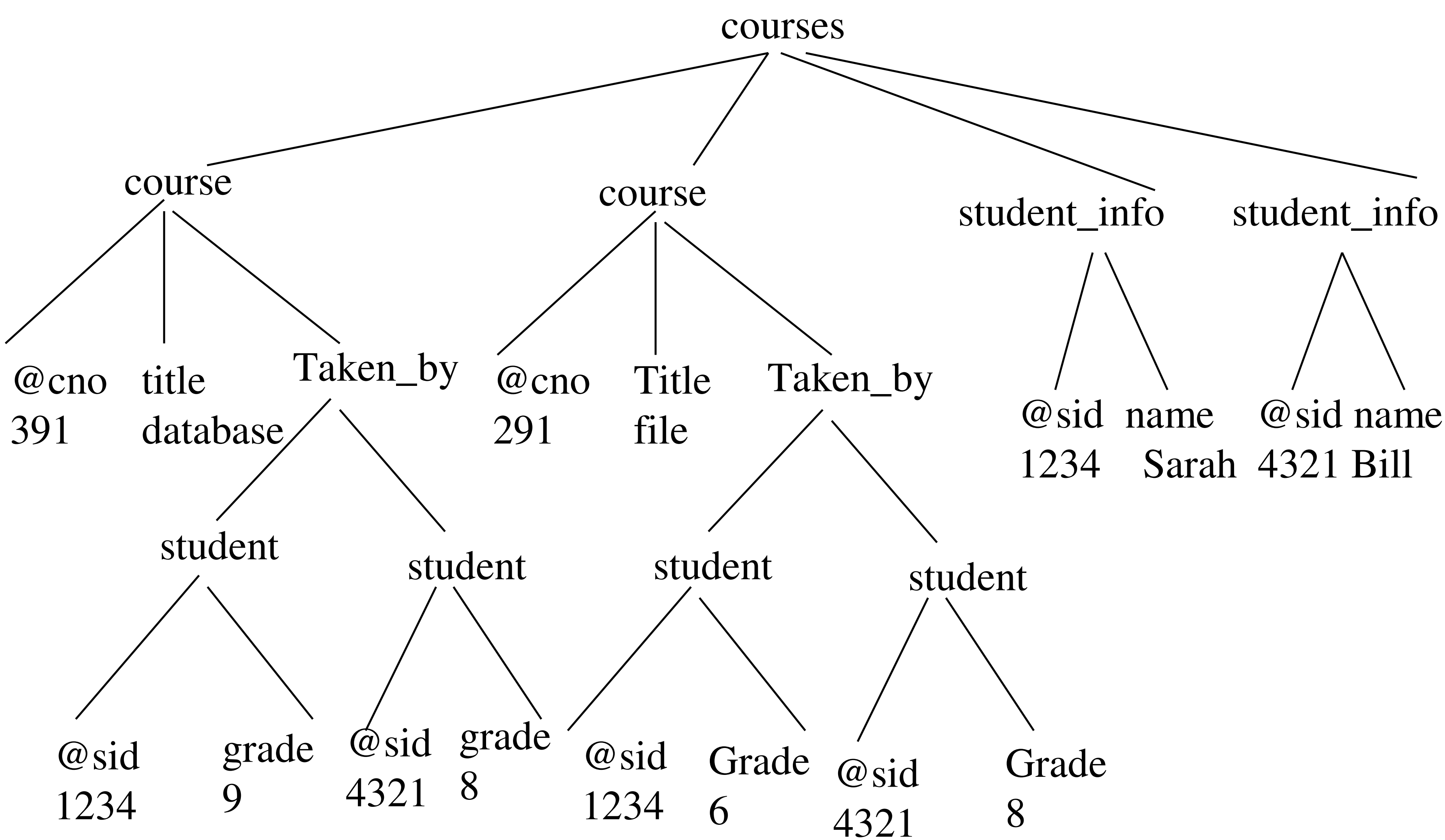
Motivation Example for XML

```
<!DOCTYPE courses [
  <!ELEMENT courses ( course*) >
  <!ELEMENT course( title, taken_by) >
  <!ATTLIST course cno CDATA #REQUIRED>
  <!ELEMENT  title (#PCDATA)>
  <!ELEMENT take_by( student*)>
  <!ELEMENT student ( name, grade)>
  <!ATTLIST student sid CDATA #REQUIRED>
  <!ELEMENT name ( #PCDATA)>
  <!ELEMENT grade (#PCDATA) >
] >
```


courses
course
course
@cno 391
title database
Taken_by
@cno 291
title file
Taken_by
student
student
student
student
@sid 1234
name Sarah
grade 9
@sid 4321
name Bill
grade 8
@sid 1234
name Sarah
grade 6
@sid 4321
name Bill
grade 7

Motivation Example for XML

```
<!DOCTYPE courses [
 <!ELEMENT courses ( course*, student_info*) >
 <!ELEMENT course( title, taken_by) >
 <!ATTLIST course cno CDATA #REQUIRED>
 <!ELEMENT  title (#PCDATA)>
 <!ELEMENT take_by( student*)>
 <!ELEMENT student( grade) >
<!ELEMENT grade (#PCDATA) >
 <!ATTLIST student sid CDATA #REQUIRED>
 <!ELEMENT student_info( sid*, name) >
 <!ELEMENT numberEMPTY>
 <!ATTLIST number sid CDATA #REQUIRED>
 <!ELEMENT name ( #PCDATA)>
] >
```


courses
course
course
student_info
student_info
@cno 391
title database
Taken_by
@cno 291
Title file
Taken_by
@sid 1234
name Sarah
@sid 4321
name Bill
student
student
student
student
@sid 1234
grade 9
@sid 4321
grade 8
@sid 1234
Grade 6
@sid 4321
Grade 8

Notations

- Assume the following disjoint sets
  - EL: the set of all element names
  - Att: the set of all attribute names, starting with @
  - Str: the set of all possible string valued attributes
  - Vert: the set of node identifies
- A DTD (Document Type Definition) is defined to be
  - D = (E, A, P, R, r), where
    - E is a finite subset of EL
    - A is a finite subset of Att
    - P is a mapping from E to element type definitions, defined as follows
      - P(t) = EMPTY or
      - P(t) ::= empty sequence | t’ in E | P(t) union P(t) | P(t) P(t) | P(t)*
    - R is a mapping from E to the power set of A
  - r is in E as the root element

- Given a DTD D = (E, A, P, R, t), a string w = w1,…, wn is a PATH in D if
  - w1 = r,
  - wi is in the alphabet of P(wi-1), for each i in [2, n-1], and
  - wn is in the alphabet of P(wn-1) or wn = @l for some @l in R(wn-1)
- Assume w is a path in D, length(w) is defined as n, and last(w) as wn.
- Given a DTD D,
  - Paths(D) stands for the set of all paths in D,
  - Epaths(D) stands for the set of all paths that ends with an element type
- DTD is recursive if Paths(D) is infinite.

Example

```
<!DOCTYPE courses [
  <!ELEMENT courses ( course*) >
  <!ELEMENT course( title, taken_by) >
  <!ATTLIST course cno CDATA #REQUIRED>
  <!ELEMENT  title (#PCDATA)>
  <!ELEMENT take_by( student*)>
  <!ATTLIST student sid CDATA #REQUIRED>
  <!ELEMENT name ( #PCDATA)>
  <!ELEMENT grade (#PCDATA) >
] >
```

The followings are paths in D

courses,
courses.course
courses.course.@cno
courses.course.title
courses.course.title.S
courses.course.taken_by
courses.course.taken_by.student
courses.course.taken_by.student.@sid
courses.course.taken_by.student.name
courses.course.taken_by.student.name.S
courses.course.taken_by.student.grade
courses.course.taken_by.student.grade.S

- An XML tree T is defined to be a tree (V, lab, ele, att, root), where
  - V is a finite subset of Vert (nodes)
  - lab: V => EL
  - ele: V => Str U V*
  - att is a partial function V x Att => Str
  - root in V is called the root of T
- Given an XML tree T, a string w1 … wn, where with wi, I< n-1, in EL, and wn is in the union of El, Att, and {S}.
  - The string is a path in T if tehre are vertices v1, …, vn-1 in V such that
    - v1 = root, vi+1 is a child of vi for I <= n-1, lab(vi) = wi for I <= n-1
    - if wn in El, then there is a child vn ofv n-1 such that lab(vn) = wn.
    - If wn = @l then att(vn-1, @l) is defined
    - if wn = S (#PCDATA) then vn-1 has a child in Str.

- T is compatible with D if and only if
  - paths(T) is a subset of paths(D)

Tree Tuples

- XML trees are defined as sets of tree tuples
- Given a DTD D = (E, A, P, R, r), a *tree tuple* t in D is defined as a function from paths(D) to Vert U Str U {null} such that
  - For p in EPaths(D), t(p) is in Vert + {null} , and t(r) =/= {null}
  - For p in paths(D) = EPahths(D), t(p) is in Str + {null}.
  - If t(p1) = t(p2) and t(p1) is in Vert, then p1 = p2
  - If t(p1) = null, and p1 is a prefeix of p1, then t(p2) = null.
  - { p in paths(D) | t(p) =/= null } is finite.
- T(D)is defined to be the set of all tree tuples in D.

Example

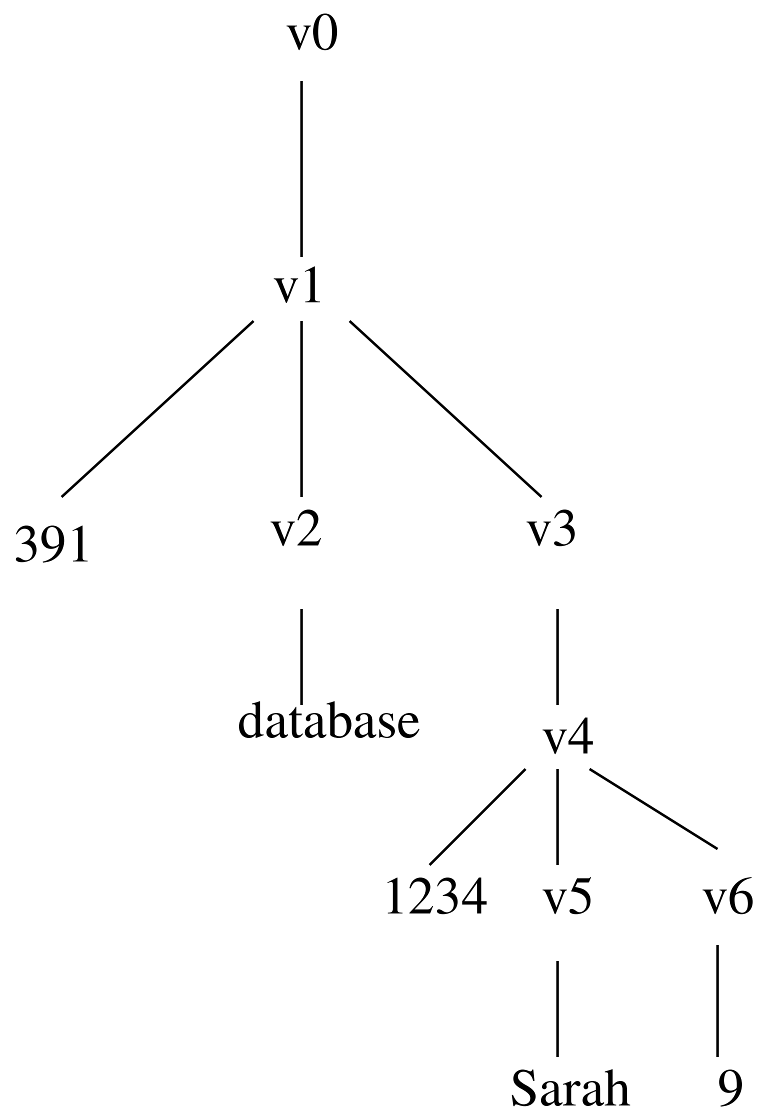
```
<!DOCTYPE courses [
  <!ELEMENT courses ( course*) >
  <!ELEMENT course( title, taken_by) >
  <!ATTLIST course cno CDATA #REQUIRED>
  <!ELEMENT  title (#PCDATA)>
  <!ELEMENT take_by( student*)>
  <!ATTLIST student sid CDATA #REQUIRED>
  <!ELEMENT name ( #PCDATA)>
  <!ELEMENT grade (#PCDATA) >
] >
```

The followings are paths in D

t(courses) = v0
t(courses.course) = v1
t(courses.course.@cno) = 391
t(courses.course.title) = v2
t(courses.course.title.S = database
t(courses.course.taken_by= v3
t(courses.course.take_by.student) = v4

t(courses.course.taken_by.student.@sid) =1234
t(courses.course.taken_by.student.name) = v5
t(courses.course.taken_by.student.name.S) = Sarah
t(courses.course.taken_by.student.grade) = v6
t(courses.course.taken_by.student.grade.S) = 9

The XML tree for this one tree tuple



- Important Results:
  - Given a DTD D and an XML tree T such that T conforms with D. Then T can be represented by a set of tree tuples, if we consider it as an unordered tree.

Functional Dependencies

- Let D be a DTD, S1 and S2 are finite non-empty subsets of paths(D).
  - A functional dependency FD over D is an expression of the form
    S1 --> S2
  - An XML tree T satisfies S1 --> S2 if for every pair of tree tuples t1, t2 in tuples(T),
    - t1.S1 = t2.S2 and t.S1 =/= null implies t1.S2 = t2.S2.

Example

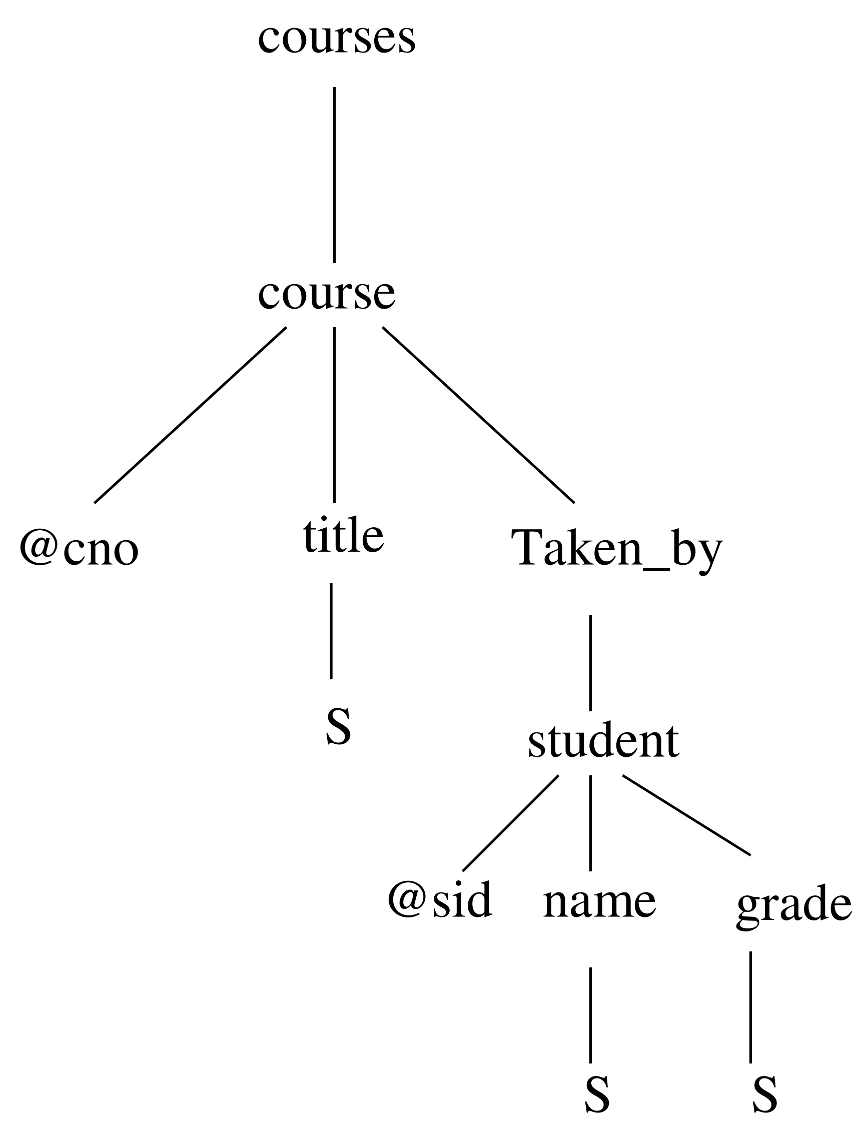
```
<!DOCTYPE courses [
  <!ELEMENT courses ( course*) >
  <!ELEMENT course( title, taken_by) >
  <!ATTLIST course cno CDATA #REQUIRED>
  <!ELEMENT  title (#PCDATA)>
  <!ELEMENT take_by( student*)>
  <!ATTLIST student sid CDATA #REQUIRED>
  <!ELEMENT name ( #PCDATA)>
  <!ELEMENT grade (#PCDATA) >
] >
```

The followings are paths in D

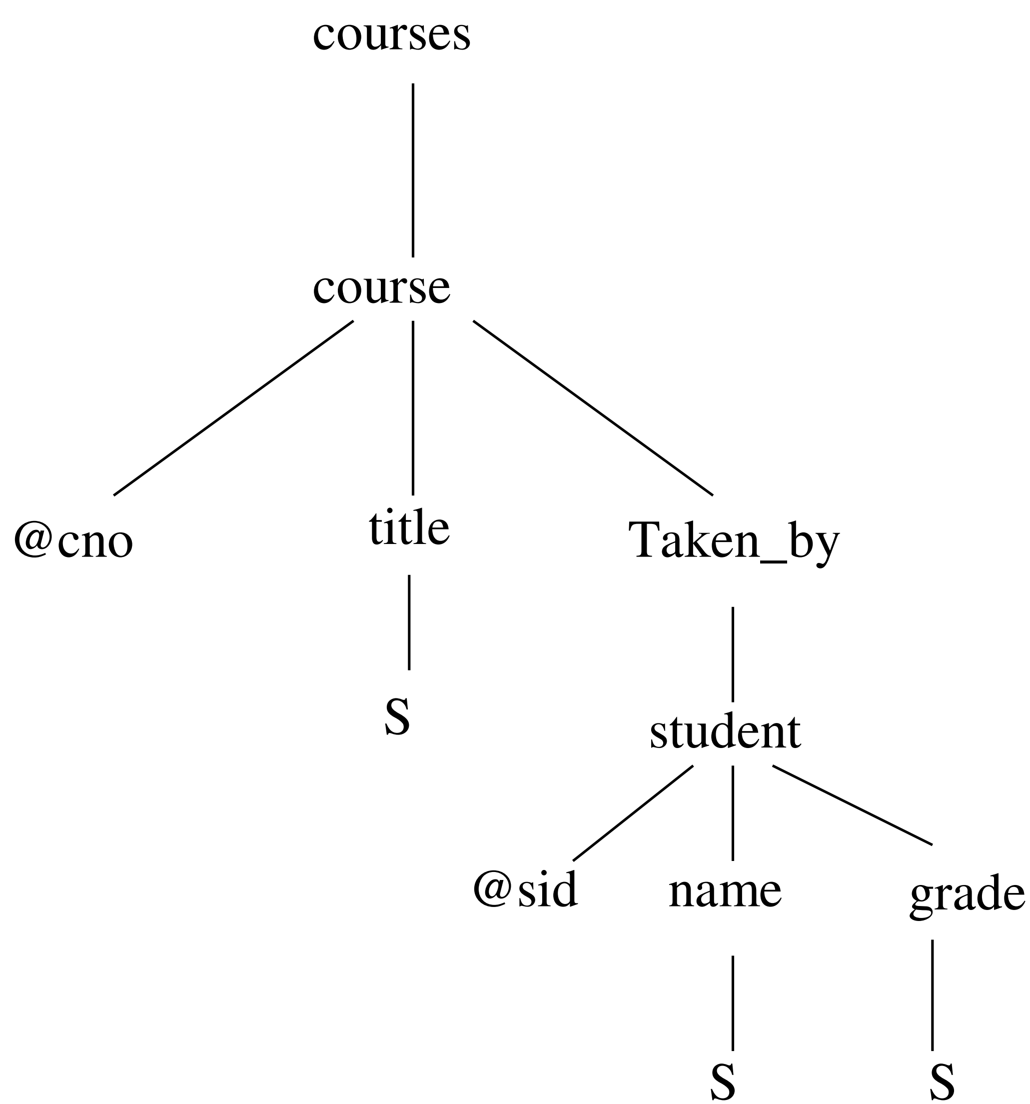
courses,
courses.course
courses.course.@cno
courses.course.title
courses.course.title.S
courses.course.taken_by
courses.course.taken_by.student
courses.course.taken_by.student.@sid
courses.course.taken_by.student.name
courses.course.taken_by.student.name.S
courses.course.taken_by.student.grade
courses.course.taken_by.student.grade.S

Example: Paths(D)

courses,
courses.course
courses.course.@cno
courses.course.title
courses.course.title.S
courses.course.taken_by
courses.course.taken_by.student
courses.course.taken_by.student.@sid
courses.course.taken_by.student.name
courses.course.taken_by.student.name.S
courses.course.taken_by.student.grade
courses.course.taken_by.student.grade.S



Example



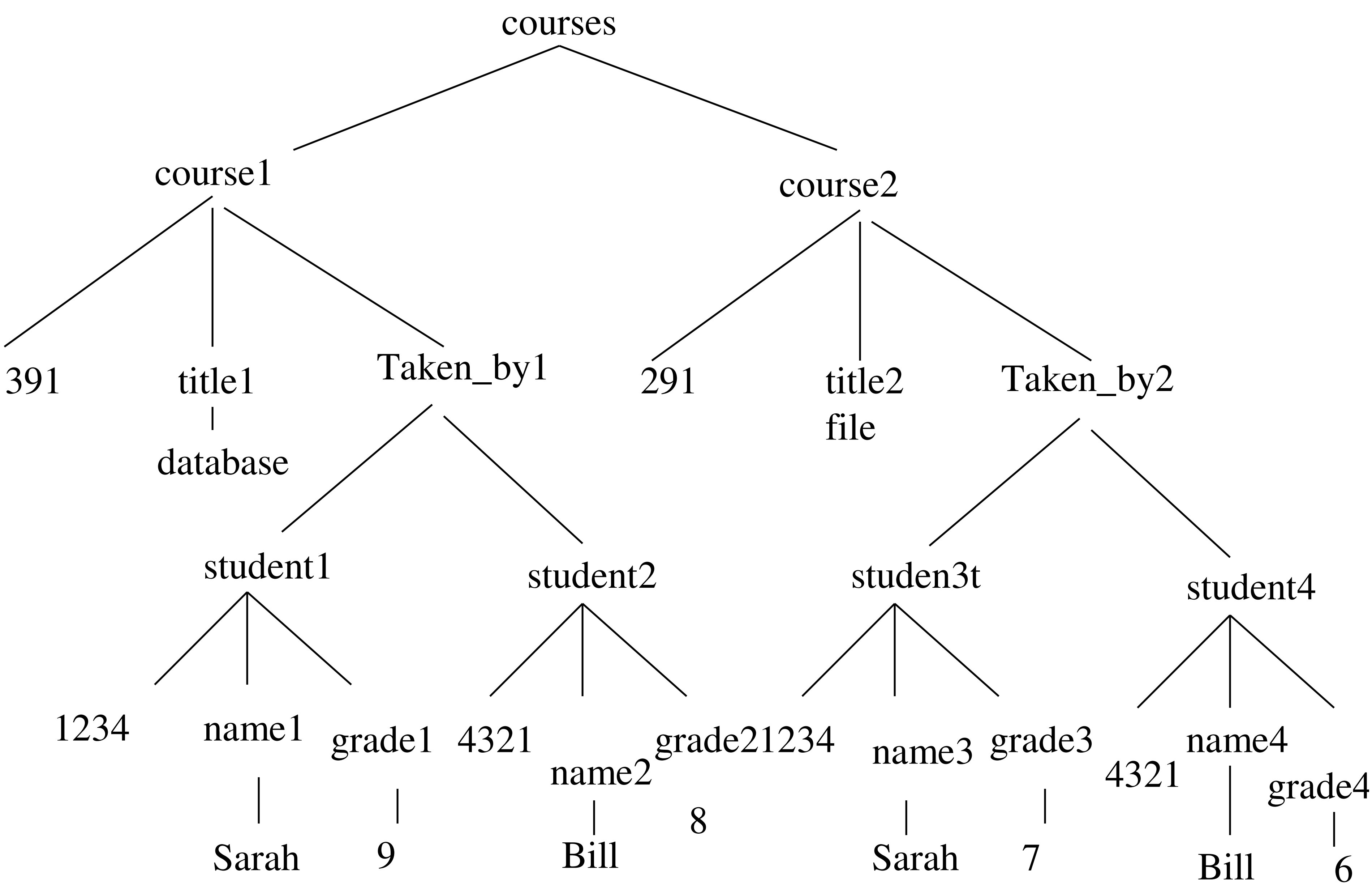
Constraint:

cno is a key of course

FD1:

courses.course.@cno --> courses.course

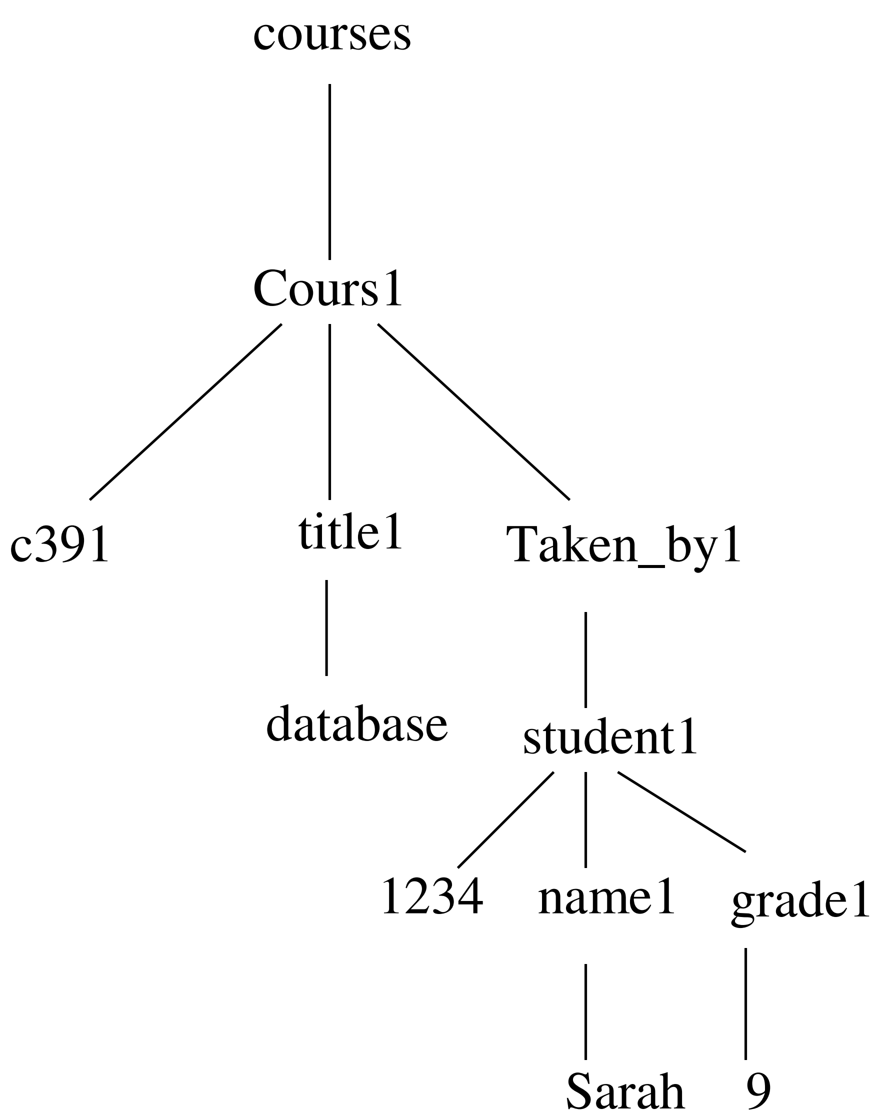
Consider a sample XML Tree T1

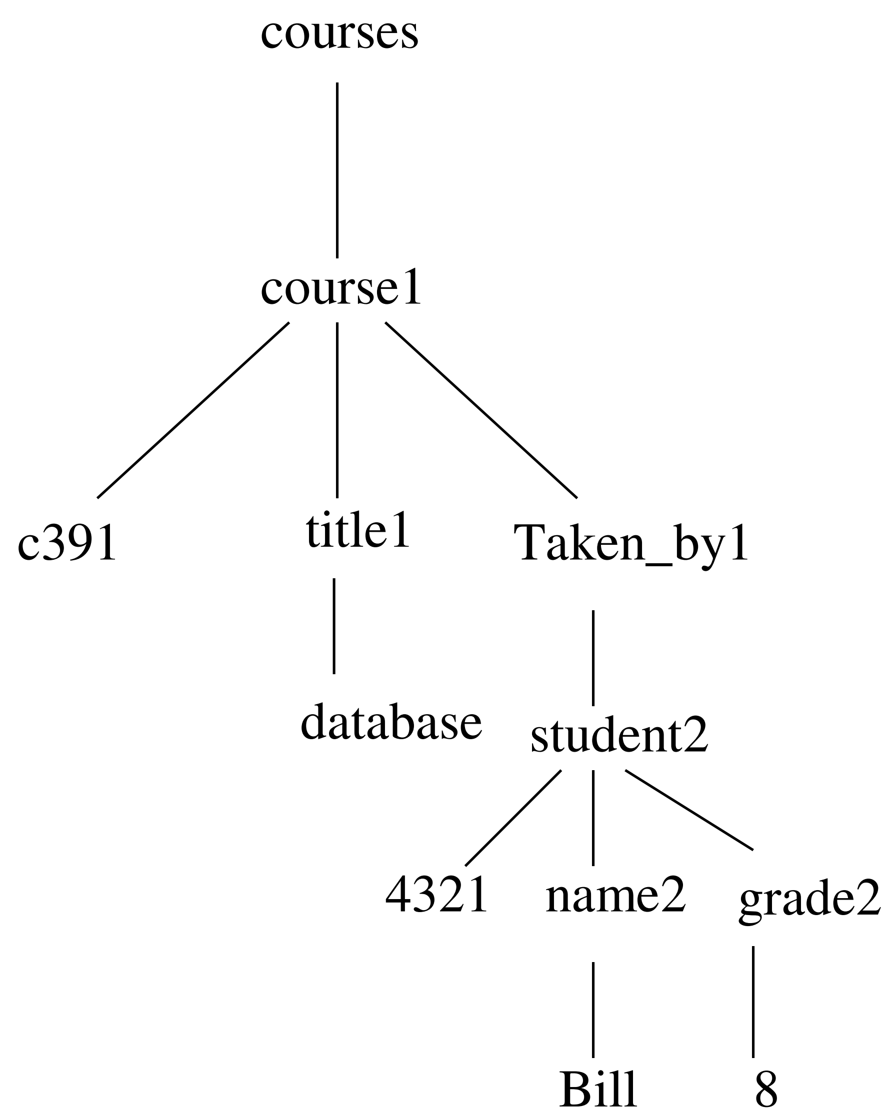


The corresponding flat table for T1

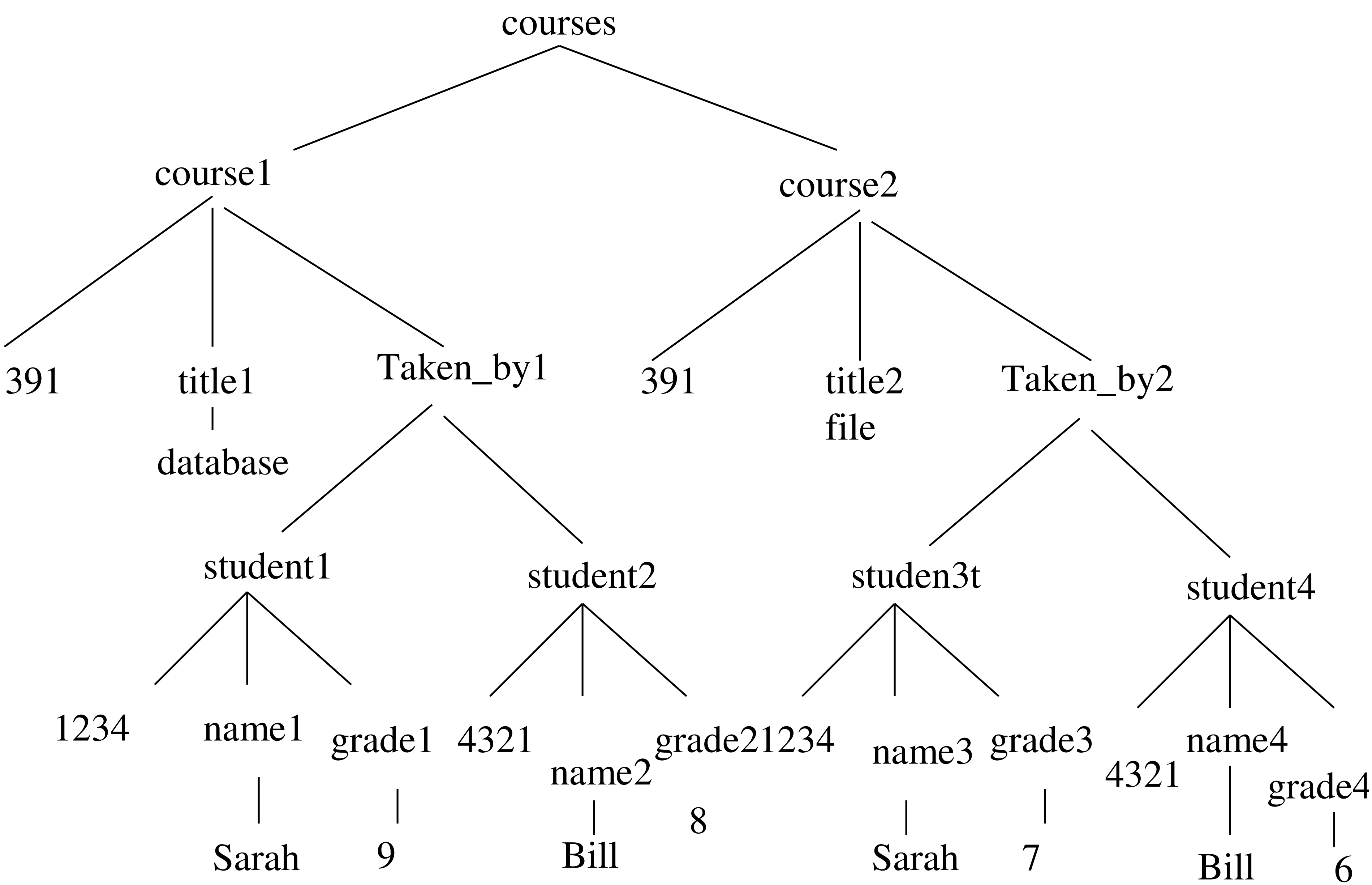
Cno	Title	SID	Name	Grade
391	database	1234	Sarah	9
391	database	4321	Bill	8
291	file	1234	Sarah	7
291	file	1234	Bill	6

The following are the only two tree typles with cno = c391 in T1





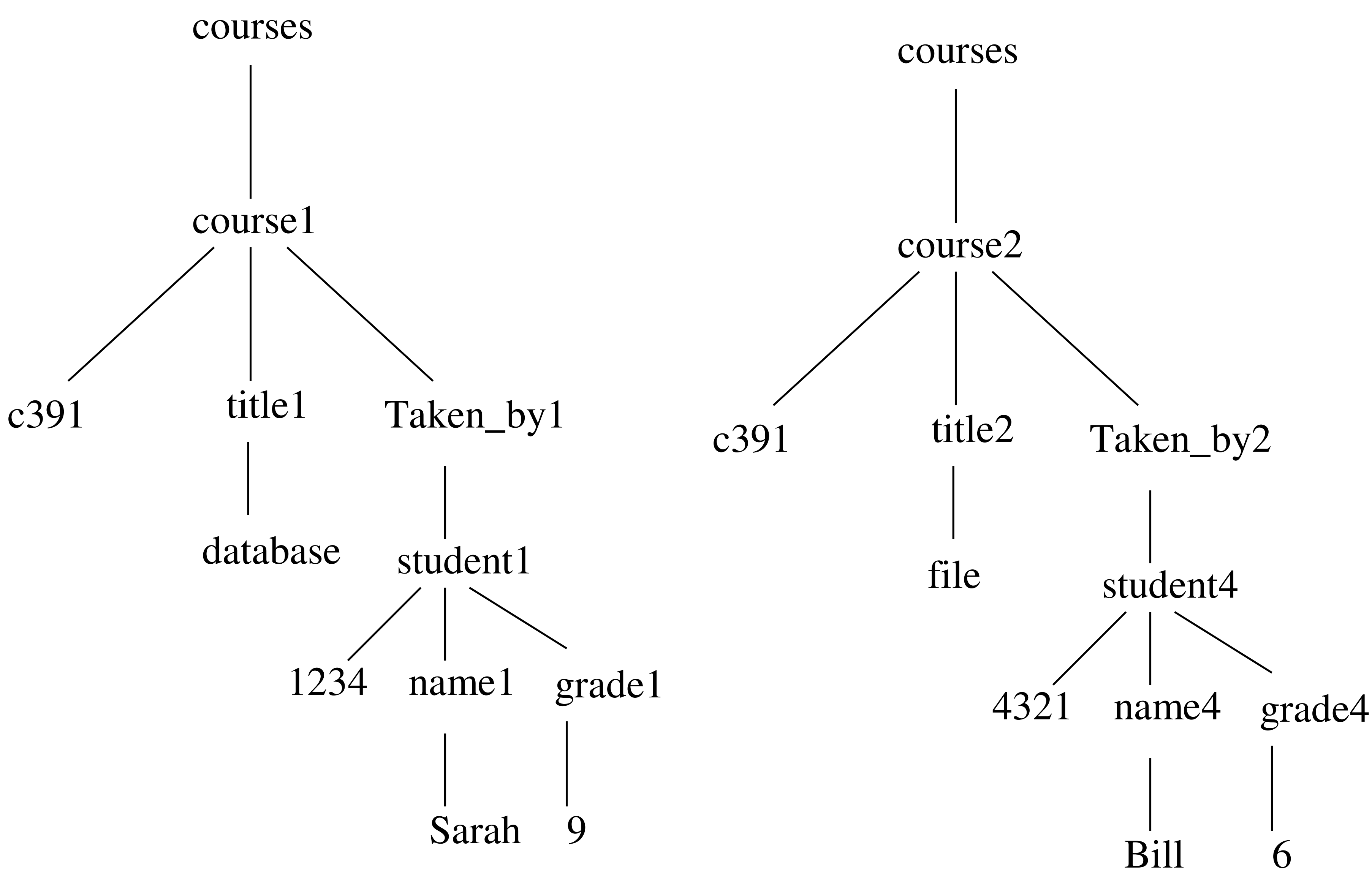
Consider another XML Tree T2



The corresponding flat table for T2

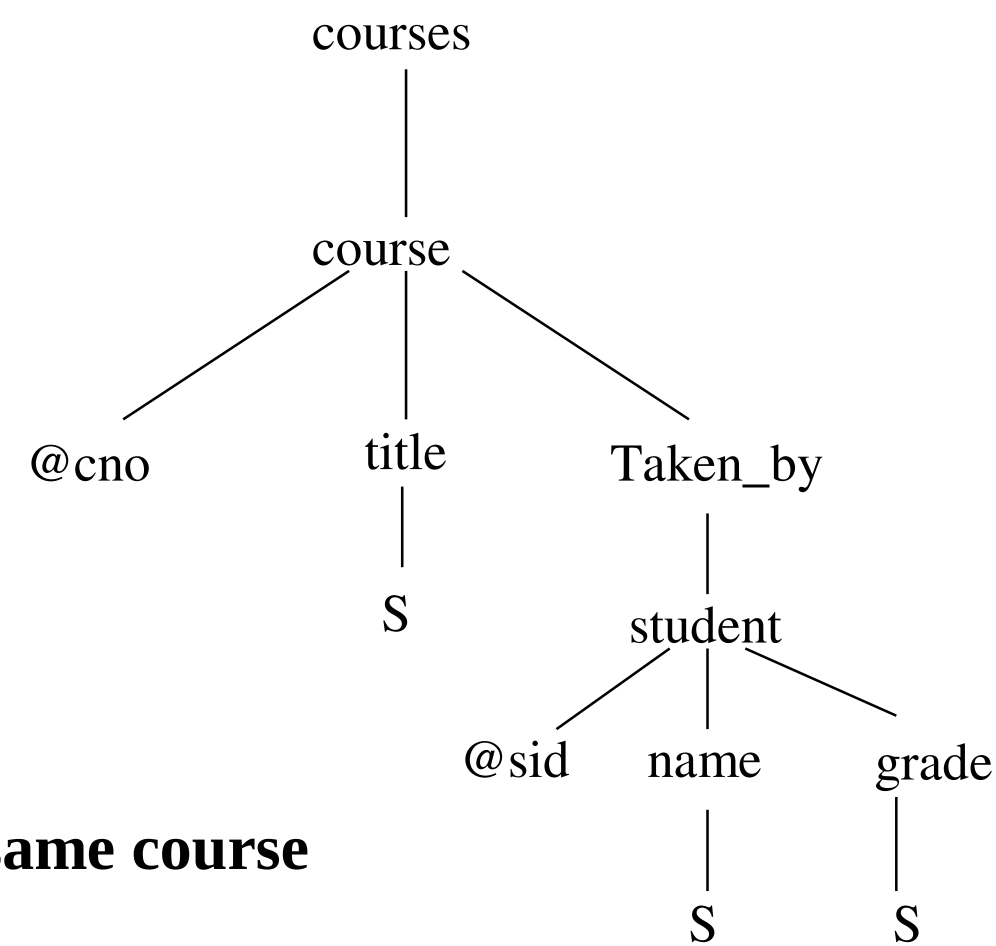
Cno	**Title**	**SID**	**Name**	**Grade**
391	database	1234	Sarah	9
391	database	4321	Bill	8
391	file	1234	Sarah	7
391	file	1234	Bill	6

The following are two tree typles with cno = c391 in T2



- Observation
  - Both T1 and T2 conform to the DTD
  - T1 satisfies the FD
    - **courses.course.@cno --> courses.course**
  - T2 does not satisfy the above FD

Example



Constraint:

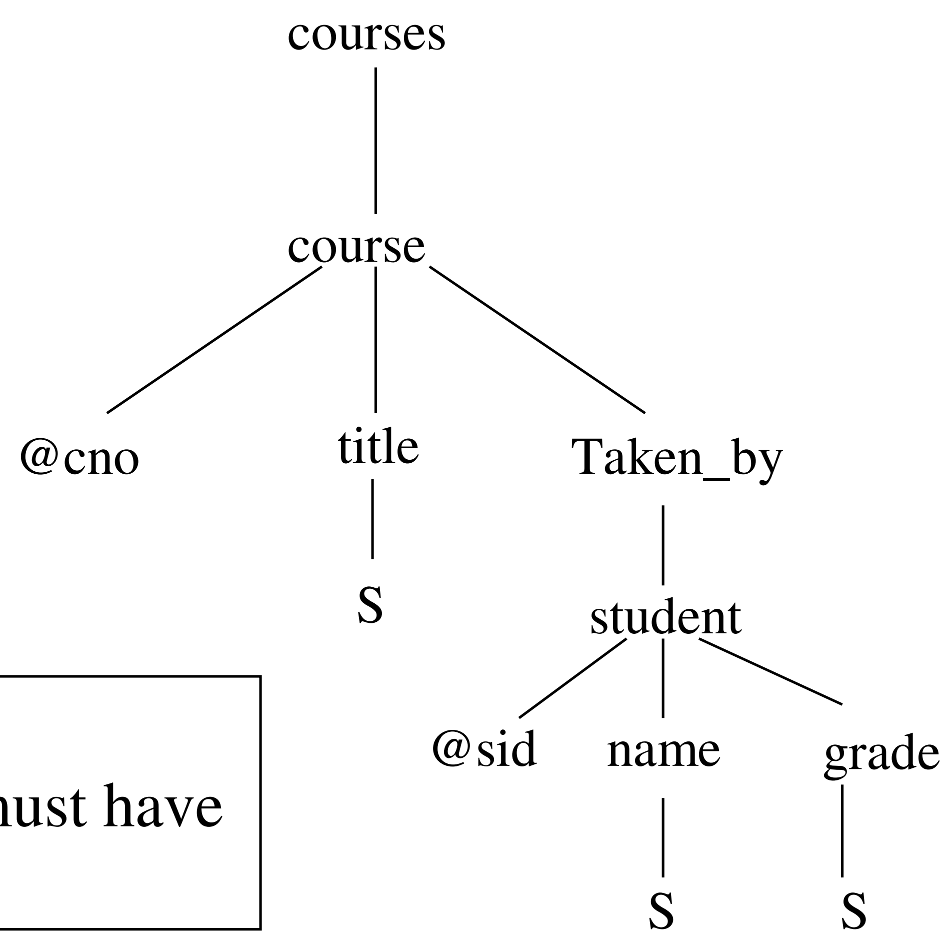
two distinct students of the same course cannot have the same sid

FD2:

{ **courses.course, courses.course.taken_by.student.@sid}**

--> courses.course.taken_by.student

Example



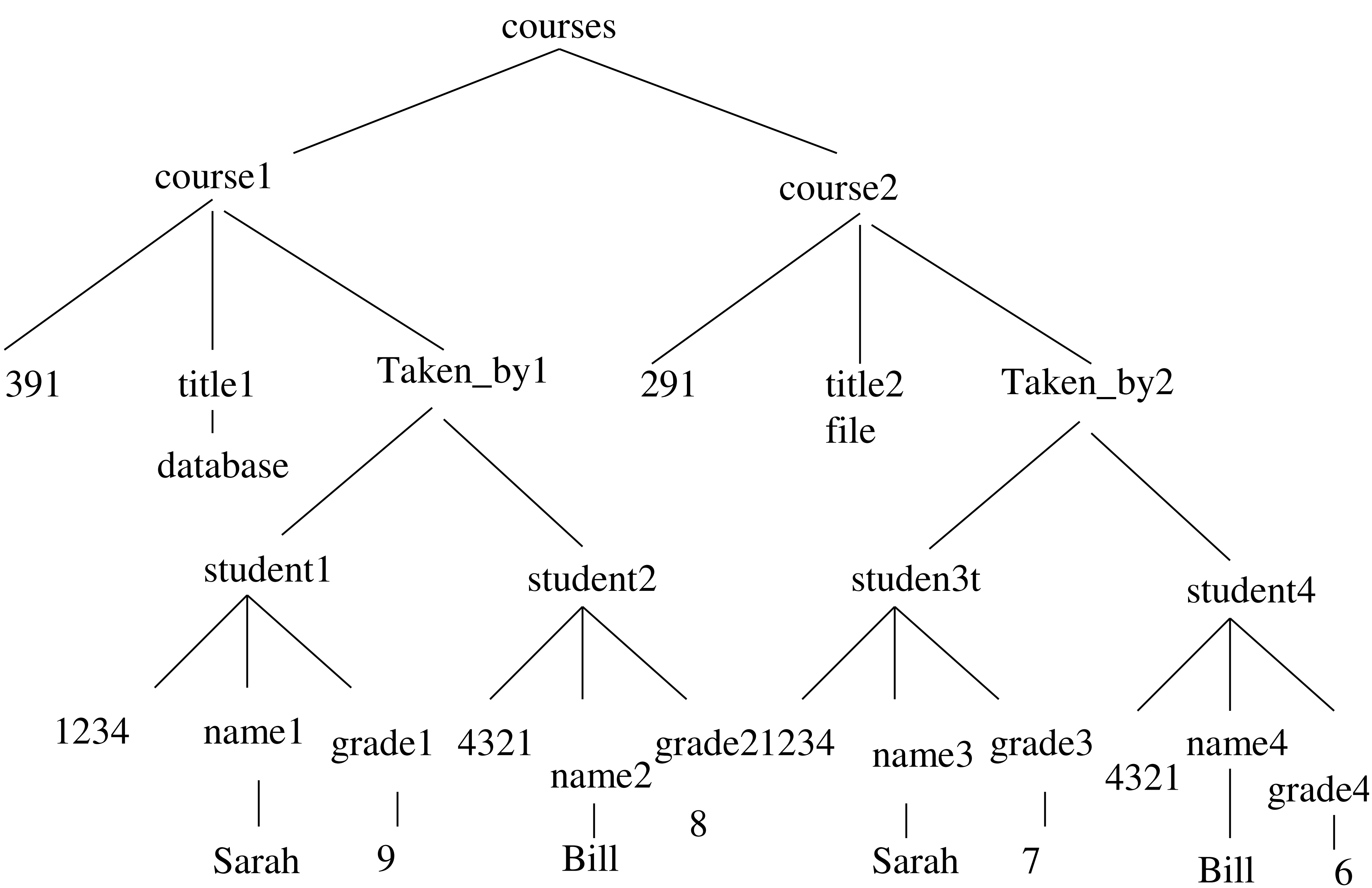
Constraint:
two students with the same sid must have the same name

FD3:
courses.course.taken_by.student.@sid -->
courses.course.taken_by.student.name.S

XNF: An XML Normal Form

- Given a DTD, and a set F of FDs, (D, F) is in XML normal form (XNF) if and only if for every nontrivial FD of the form S --> p.@l or S --> p.S, it is the case that S--> p is implied by F.
- Intuition
  - For every set values of the elements in S, we can find only one value of p.@l. Thus, we need to store the value only one.

Consider the following example again



We have FD3:

courses.course.taken_by.student.@sid -->
courses.course.taken_by.student.name.S

But the following does not held:

courses.course.taken_by.student.@sid -->
courses.course.taken_by.student.name

This implies that the student name for a given sid, the document may have multiple copies of student name.

Relationships with other normal forms

- Assume a standard coding between tables and XML documents
  - A relation schema in in BCNF if and only if its XML counter part is in XNF
- Assume a standard nesting operations and coding
  - A nested relation is in NNF if and only if its XML representation is in XNF.

Normalization Algorithm

- Two basic operations
  - Moving attributes
  - Creating new element types
- Given a DTD D and a set F of FDs
  - If (D, F) is in XNF, return
  - Otherwise find an anomalous FD and use the two basic operations to modify D to eliminate the anomalous FD,
  - Continue the above steps until (D, F) is in XNF.
- The normalization algorithm is efficient and join-lossless