
INTRODUCTION TO DATABASE
- QUERYING XML
- XPATH

Not as mature as Querying Relational Database

Sequence of development
- XPath
- XSLT
    XPath
    + transformations
    output formatinng
- XQuery
    Xpath
    Full-featured Query Language

other
- XLink
- XPointer

XPath
Think of XML as a tree
XPath = navigation down the tree

XPath = Path expression + Conditions

Path expression constructs

/	← root element
X or *	← name of the element, * match any sub element
@	← get attribute
//	← any descendant element including self
	← self = element we currently are

Condition construct

[C]	← C is a condition
[Price <50]	← sample condition
[3]	← noa condition, matches the #3 subelement !!!

Build-in functions (lots of them)

contains(s1, s2)	← returns true is s1 contains s2
name()	← returns the tag of the current element in the path

Navigation ‘axes’ (13 of them in XPath)
Used to navigate the tree

parent::	← go to parent
following-sibling::	← go to next sibling (same level) on right
descendants::	← // without self !

self:: ← current element

XPath queries operate on and return sequence of elements
* XML document
XML stream

Sometimes result can be expressed as XML, not always

INTRODUCTION TO DATABASE
- QUERYING XML
- XPATH DEMO !

doc(“BookstoreQ.xml”)/Bookstore/Book/Title
<?xml version=”1.0” encoding=”UDF-8”?>
<Title>My first title</Title>
...
<Title>My last title</Title>
← return all titles in BookstoreQ.xml file

doc(“BookstoreQ.xml”)/Bookstore/(Book|Magazine)/Title
← books and magazines

doc(“BookstoreQ.xml”)/Bookstore/*/Title
← matches books and magazines and other if any

doc(“BookstoreQ.xml”)//Title
← matches all elements Title in the TREE

doc(“BookstoreQ.xml”)//*
← huge result
← * matches any elements
← includes all the elements with CONTENTS/subelements
← element 1 ~ entire file !
← element 2 = first element of element 1

doc(“BookstoreQ.xml”)/Bookstore/Book/data(@ISBN)
<?xml version=”1.0” encoding=”UDF-8”?> ISBN1 ISBN2 …. LastISBN
← returns all ISBN numbers

!!! Attribute cannot be serialized and formated in XML ⇒ data( ) to extract data !!!

Conditions
doc(“BookstoreQ.xml”)/Bookstore/Book[@Price <90]
← Books with prices less than 90
← Book = book + sub elements

!!! There is no slash before a bracket/condition !!!

!!! Everything is case sensitive. Price <> from price !!!

doc("BookstoreQ.xml")/Bookstore/Book[@Price <90]/Title
← Return Title of Books with prices less than 90

doc("BookstoreQ.xml")/Bookstore/Book[Remark]/Title
← Matches books that have a remark
!!! Check existence !!!

doc("BookstoreQ.xml")/Bookstore/Book[@Price<90 and
Authors/Author/Last_Name = "Eullman"]/Title
← There exists an Author with Last_Name

doc("BookstoreQ.xml")/Bookstore/Book[@Price<90 and
Authors/Author/Last_Name = "Eullman" and
Authors/Author/Last_Name = "Mayssat"]/Title
← Matches books with 2 authors with specified last name

!!! Remember Authors/Author/Last_Name means 'there exists one author with last name is' !!!
!!! So above we have 2 authors with different last name, not one author with 2 last name !!!

doc("BookstoreQ.xml")/Bookstore/Book[@Price<90 and
Authors/Author/Last_Name = "Eullman" and
Authors/Author/First_Name = "Mayssat"]/Title

!!! May not match the same author. See previous case !!!

doc("BookstoreQ.xml")/Bookstore/Book[@Price<90 and
Authors/Author[Last_Name = "Eullman" and First_Name="Jeffrey"]/Title
← There exists an Author with Last_Name = … and a First_Name= ….
← Condition on same author
← condition within the condition

doc("BookstoreQ.xml")//Authors/Author[2]
← return second author of each authors element

doc("BookstoreQ.xml")//Book[contains(Remark, "great")]/Title
← return title of boosk whose remark contains greet
← 'contains' is a builtin predicate

doc("BookstoreQ.xml")//Magazine[Title =
doc("BookstoreQ.xml")//Book/Title]
← return magazines with a title same as a book
← almost a self join ;-)

!!! = is existentially qualified ;-) !!!

Axis

```
doc("BookstoreQ.xml")/Bookstore//*[name(parent::*) != "Bookstore"
        and name(parent::*) != "Book"]
                        ← All elements whose parent is not "Bookstore" or "Book"

doc("BookstoreQ.xml")/Bookstore/(Book|Magazine) [
        Title = following-sibling::*/Title]
                        ← when a latter one is =

doc("BookstoreQ.xml")/Bookstore/(Book|Magazine)
        [ Title = following-sibling::*/Title or Tittle = preceding-sibling::*/Title]
                        ← when a latter or former one
```

for all : implicit existential …

Books where EVERY author's first name include "J"

```
doc("BookstoreQ.xml")//Book
        [ count(Authors/Author[contains(First_Name, "J")]) =
          count(Authors/Author/First_Name)]
                                                ← Trick : use count !
```

!!! XPATH queries can also start with function !!!
example:: count(doc("BookstoreQ.xml")//Book) ⇒ return the number of books !!!

none / not / !=

```
doc("BookstoreQ.xml")//Book [
        Authors/Author/Last_Name = "Ullman" and
        count(Authors/Author[Last_Name = "Widom"]) = 0
        ]/Title
```

INTRODUCTION TO DATABASE
- QUERYING XML
- XQUERY

Expression language (compositional)
Data → query → result = same type of data
data → query1 → … → queryN → result

XPath is one type of expression

XQUERY: FLWOR expression

```
For $var in expr                ← iterator variables, expr is a set
Let $var := expr                ← assignment, only run once
Where condition                 ← filter ~ where in SQL
```

Order By expr ← sorts result
Return expr ← what we want to get as the result of query

!!! All clause except Return are optional !!!
!!! For and Let can repeated or interleaved !!

Mixing qeries and XML
<result> { … query goes her … } </Result>

INTRODUCTION TO DATABASE
- QUERYING XML
- XQUERY DEMO

```
for $b in doc("BookstoreQ.xml")/Bookstore/Book
where $b/@Price < 90
        and $b/Authors/Author/Last_Name ="Ullman"
return $b/Title
```

example II
Title and author first names of books whose title constains one of the author's first names

```
for $b in dec("BookstoreQ.xml")/Bookstore/Book
where some $fm in $b/Authors/Author/First_Name statisfies contains($b/Title, $fm)
return <Book>
        { $b/Title }
        { $b/Authors/Author/First_Name }
        </Book>
                        ← where some <var> in <iterator> satisfies <condition>
                        ← in return, we build elements
                        ← prints all the First_Names including those who do not match
[...]
return <Book>
        { $b/Title }
        {       for $fn in $b/Authors/Author/First_Name
                where containes($b/Title, $fn)
                return $fm
        }
        </Book>
                        ← now includes only matching first names in book title
                        ← we can match queries and expression as we wish
```

example III
Find the average book price

```
<Average>
        {
                let $plist := doc("BookstoreQ.xml")/Bookstore/Book/@Price
                return avg($Plist)
        }
</Average>
```

or

```
<Average>
        {
                let $a := avg(doc("BookstoreQ.xml")/Bookstore/Book/@Price
                return $a
        }
</Average>
```

output =

```
<?xml version="1.0" encoding="UTF-8"?>
<Average>65</Average>
```

example IV
Books whose price is below average

```
let $a := avg(doc("BookstoreQ.xml")/Bookstore/Book/@Price
for $b in doc("BookstoreQ.xml")/Bookstore/Book
where $b/@Price < $a
return <Book>
        { $b/Title }
        <Price> {  $b/data(@Price)  } </Price>
        </Book>
```

!!! Attribute Price is turned into an element !!!

example V
Return books in order of price

```
for $b in doc("BookstoreQ.xml")/Bookstore/Book
order by xs:int($b/@Price)                          ← xs:int(...) turns a string into an integer
return <Book>
        { $b/Title }
        <Price> {  $b/data(@Price)  } </Price>
        </Book>
```

!!! Without xs:int(...) the ordering is based on the string Price, so 100 before 25 !!!

example VI
Returns all the Last_Names without duplicates

```
for $n in distinct-values doc("BookstoreQ.xml")//Last_Name
return <Last_Name> { $n } </Last_Name>
                                    ← distinct values remove xml tags and duplicates!
```

!!! Use { } to evaluate $n !!!

Example VII:
Books where every author's first name include "J"

```
for $b in doc("BookstoreQ.xml")/Bookstore/Book
where every $fn in $b/Authors/Author/First_Name
        satisfies contains($fn, "J")
return $b
                                        ← where every <var> in <iterator> satisfies <condition>
```

!!! every : universal quantification !!!

example VIII
All pairs of titles containing a shared author

```
for $b1 in doc("BookstoreQ.xml")/Bookstore/Book
for $b2 in doc("BookstoreQ.xml")/Bookstore/Book
where $b1/Authors/Author/Last_Name = $b2/Authors/Author/Last_Name
        and $b1/Title != $b2/Title      ← do not match itself
        and $b1/Title < $b2/Title       ← get the pair only once, instead of twice
return
        <BookPair>
                <Title1> { data($b1/Title) } </Title1>
                <Title2> { data($b2/Title) } </Title2>
        </BookPair>
                                        ← 2 iterators
```

!!! A self-join: use 2 iterators !!!
!!! imbriqued construct !!!

example IX
Invert data elements: Authors with the books they've written

```
<InvertedBookstore>
{       for $ln in distrinct-values(doc("BookstoreQ.xml")//Author/Last_Name)
        for $fn in distrinct-values(doc("BookstoreQ.xml")//Author[Last_Name=$lm]/First_Name)
        return <Author>
                <First_Name> { $fn } </First_Name>
                <Last_Name> { $ln } </Last_Name>
                { for $b in doc("BookstoreQ.xml")/Bookstore/Book
                                [Authors/Author/Last_Name = $ln]
                  return <Book>
                        { $b/@ISBN } { $b/@Price }          ← compare this syntax
                        { $b/Title}                         ← compare with this
                        </Book>
                }
                </Author> }
</InvertedBookstore>
```

← In this solution, we are assuming author last names are unique
← Author is outer element, Book is inner element

!!! Attributes are passed as attributes !!!

INTRODUCTION TO DATABASE
- QUERYING XML
- XSLT

XSL = Extensible Stylesheet Language
XSLT = XSL (with) Transformation
← XSLT: Rule-Based Transformation

XML Document or Streem ~ Database
XSLT Specification ~ SQL Query
XSLT Processor ~ Query Processor
XML Document or Stream ~ SQL Query answer

* Match template and replace
* Recursively match templates
* Extract values
* Iteration (for-each)
* Conditionals (if)
* Strange default/whitespace behavior
* Implicit template priority scheme

INTRODUCTION TO DATABASE
- QUERYING XML
- XSLT DEMO

Hearder
<xsl:stylesheet version=”2.0” xmlns:xsl=”http://www.w3.org/1999/XSL/Transform”>
<xsl:output method=”xml” indent=”yes” omit-xml-declaration=”yes” />
← Format the output of the xslt ‘query’
← Notice is in XML format ;-)

Footer
</xsl:stylesheet>
← closing opening tag
!!! All queries have the header and footer !!!

Example I
Book and magazine title, relabeled

<xsl:template match=”Book”>

```
        <BookTitle> <xsl:value-of select="Title" /> </BootTitle>
</xsl:template>

<xsl:template match="Magazine">
        <MagazineTitle> <xsl:value-of select="Title" /> </MagazineTitle>
</xsl:template>
                                        ← replaced matched elements with template + value
```

!!! Simple template matching !!!

Example II
Extract book which cost less than 90 dollars

```
<xsl:template match="Book[@Price &lt; 90]">     ← Look at less symbol!!
        <xsl:copy-of select="." /> </BootTitle>     ←'.' means current element , like a directory
</xsl:template>
                                        ← With XPath condition
```

!!! String !!!
⇒ Element that aren't matched in the template, XSLT returns their content, so add in template

```
<xsl:template match="text()" />
```

Example III

```
<xsl:template match="Book" />       ← eliminate the match from the elements
<xsl:template match="Book">← another template that match the same elements!
        <xsl:copy-of select="." />
</xsl:template>
<xsl:template match="Magazine">
        <xsl:copy-of select="." />
</xsl:template>
```

!!! For templates that match the same element, pick:
* The most specific
* The second one takes precedence (when template are similarly specific)
!!!

Example IV:
Copy entire document

```
<xsl:template match="/">
        <xsl:copy-of select="." />
</xsl:template>
```

Using regular expression

```
<xsl:template match="*|@*|text()">
        <xsl:copy>                                          ← copy + apply template
                <xsl:apply-templates select="*|@*|text()" />
        <xsl:copy>
</xsl:template>
```

!!! Recursive template !!!
!!! * = match any element, @* = match any attributes, text() =match any text leaf !!!

Example V
Reorder and reorganize the XML file

```
<xsl:template match="*|@*|text()">
        <xsl:copy>                                          ← copy + apply template
                <xsl:apply-templates select="*|@*|text()" />
        <xsl:copy>
</xsl:template>
<xsl:template match="@ISBN">                                ← Attribute to element
        <ISBN> <xsl:value-of select="." /></ISBN>
</xsl:template>
<xsl:template match="Author">                               ← element to attribute
        <Author LN=" {Last_Name}" FN=" {First_Name}"/>
</xsl:template>
```

!!! Attribute ⇐ ⇒ Element !!!

!!! First copy is required !!!

Example VI
XML to HTML

```
<xsl:template match="/">
        <html>
        <table border=1>
                <th> Book</th>
        <th>Cost</th>
        <xsl:for-each select="Bookstore/Book">              ← for-each
        <xsl:sort select="@Price" />                        ← sort
                <xsl:if test="@Price &lt; 90">              ← if
                        <tr>
                        <td><i>xsl:value-of select="Title"/></i></td>
                        <td><xsl:value-of select="@Price"/></td>
                        </tr>
                </xsl:if>
        <xsl:for-each>
        </table>
        </html>
</xsl:template>
```

!!! Translate XML to HTML + Query language!!!

Example VII
Change Jennifer to nothing + Change Widom to Ms Widom

```
<xsl:template match="*|@*|text()">
        <xsl:copy>
                <xsl:apply-templates select="*|@*|text()" />
```

```
        <xsl:copy>
</xsl:template>

<xsl:template match="First_Name[data(.) = 'Jennifer']">     ← remove first_name and not parent
</xsl:template>                                             ← so should not match parent

<xsl:template match="Last_Name[data(.) = 'Widom']>
        <Name> Ms. Widom</Name>                             ← replace by
</xsl:template>
```

Example VIII
Same transformation, with only one template
Expunge 'Jennifer', change 'Widom' to 'Ms. Widom'

```
<xsl:template match="*|@*|text()">
        <xsl:copy>
                <xsl:apply-templates select="*|@*|text()" />
        <xsl:copy>
</xsl:template>

<xsl:template match="Author[First_Name = 'Jennifer']">      ← replace Author element by
        <Author><name>Ms. Widom</Name></Author>
</xsl:template>
```

INTRODUCTION TO DATABASE
- DATA MODELING
- UML DATA MODELING

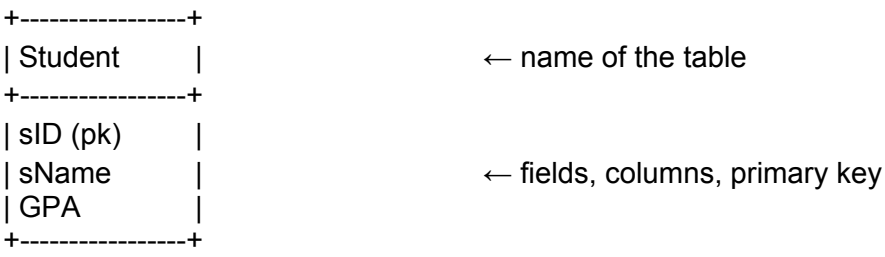
UML = Unified Modeling Language
UML is also used for programming, etc.
Here we look only at the database module.

UML → translater → relations to populate RDBMS

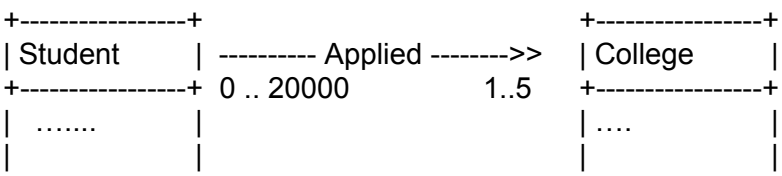
UML is a graphical composition

5 basic concepts
(1) Classes
(2) Associations
(3) Asssociation Classes
(4) Subclasses
(5) Composition & Aggregation

Classes
Name, attribute, methods
        For data modeling: add "pk" , drop methods   ← "pk" = primary key



Association



* Multiplicity of associations
Each object of class C1 (Student) is related to at least m and at most n objects of class C2 (College) is noted m .. n

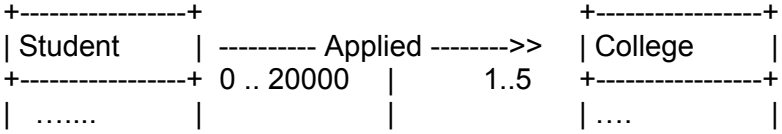
m..* ← star (*) means any number
0..n ← possibly none and up to n
0..* ← no restriction on the multiplicity
← can be abbreviated just '*'
1..1 ← default in both directions
← can be abreviated '1'

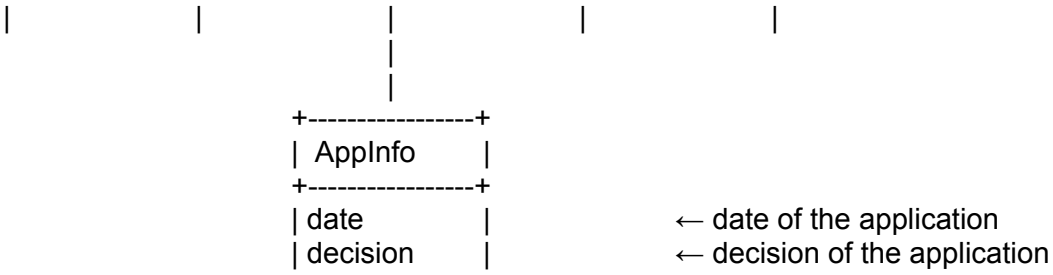
!!! Beware of the side it is positioned !!!

example
Students must apply somewhee and may not apply to more than 5 colleges ⇒ 1..5
No college takes more than 20,000 applications ⇒ 0..20000

Multiplicity of association: types of relationships
One-to-one: 0..1 -------- 0.1 ← related to exactly one
Many to one: * ---------- 0.1 ← start (*) is equivalent to 0..*
Many to many *---------- * ← no restriction on the relationship
Complete (a) 1..* ------- 1..1 ← every object must participate in the relationship
Complete (b) 1..* 1..*

Association classes
Attributes on relationships itself

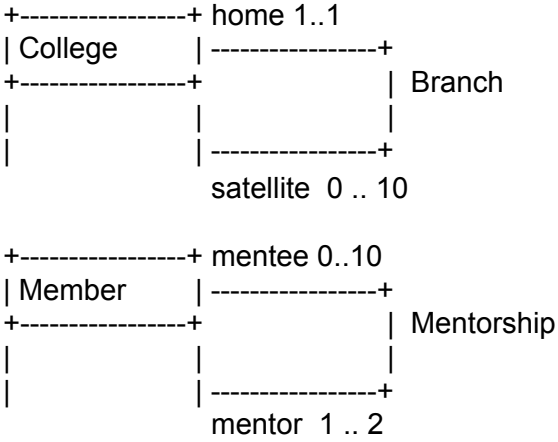




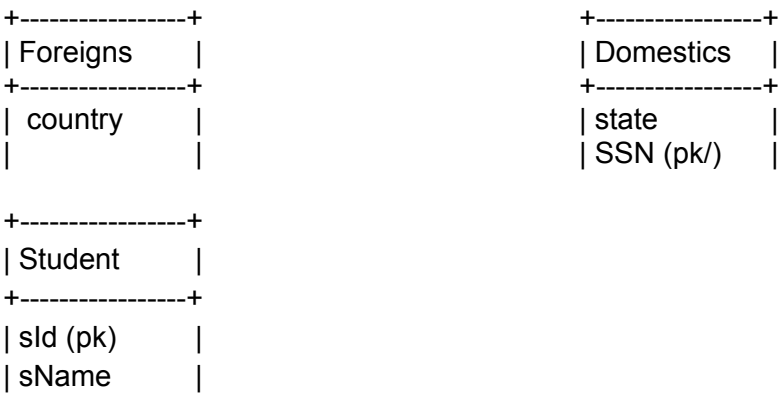
!!! In UML, there is AT MOST one association with 2 objects !!!
← one limitation of UML

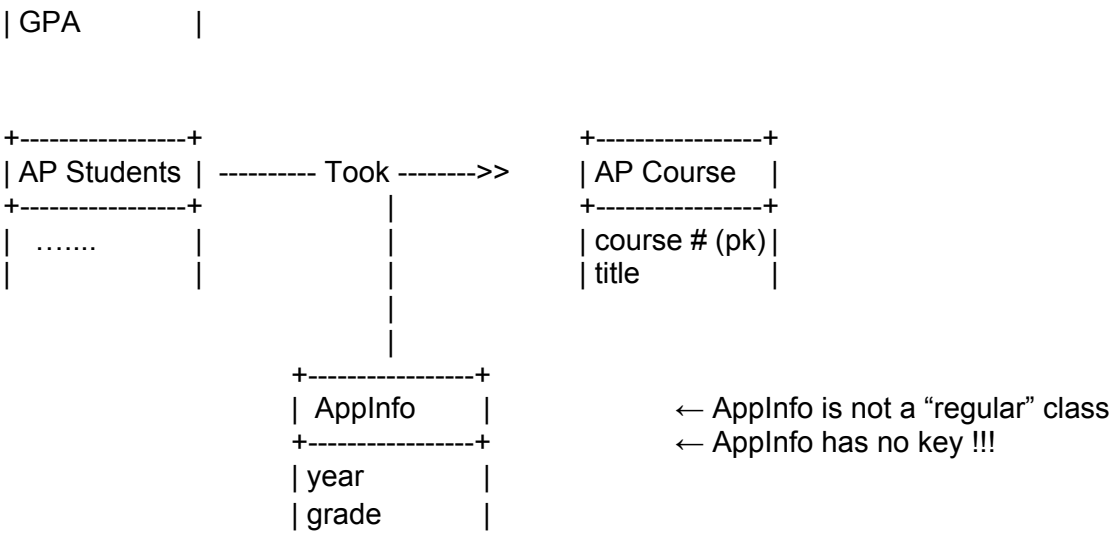
Eliminationg Association Classes
* Unnecessary if 0..1 or 1..1 multiplicity ⇒ put relation attributes in the 1..1 object
(1..1 is put on the other side of the 1..1 object)
- if 0..1, those attributes would have NULL in them

Self-associations
Associations between a class and itself



Subclasses





!!! Student parent table/ super class object of Foreign Student, domestic Student, AP Students

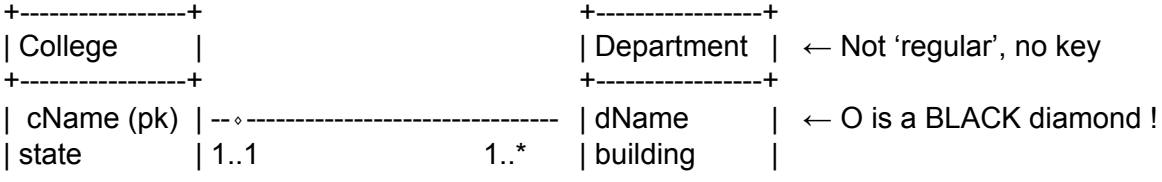
Superclass = generalization (in UML)
Subclass = Specialization (UML)

Incomplete (Partial) vs Complete
- Complete: super class is abstract
- Incomplete: super class has instance

Disjoint (Exclusive) vs Overlapping
- Disjoint: every object at most in one class
- Overlapping: object can belong to several classes

1 student can be a foreign and AP Student ← overlapping
Student is are either domestic or foreign ← complete

Composition and aggregation (diamond)
Composition is a special type of association



◇ : a department belongs to a specific college (1..1 is implicit)
College as at least one department

!!! dName may not be a key !!!

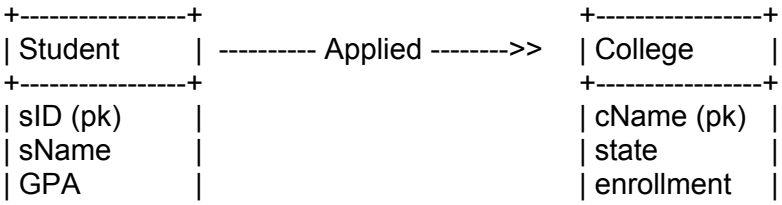
!!! black diamond is (1..1) = composition !!!
ex: each city is in exactly one state ← black diamond
!!! Open diamond is (0..1) = aggregation !!!

INTRODUCTION TO DATABASE
- DATA MODELING
- UML TO RELATIONS

UML -- (translator) ---> Relations

Design can be translated to relation automatically

Classes



To translate ⇒ turn attribute sideways ;-)
pk becomes primary keys

Student(sID, sname, GPA)
College(cName, state, enrollment)

Associations
Student --------- Applied ---------- College

Applied(sID, cName) ← a key from each side

!!! Keys for association Relations depends on multiplicity !!!

C1 (K1,O1) <- 0..1 --- A --- * -> C2 (K2,O2) ⇒ K2 is a key for A

C1 (K1,O1) <- 1..1 --- A --- * -> C2 (K2,O2) ⇒ K2 is a key for A

Association relation always needed?
Depends on multiplicity!

C1 (K1,O1) <- 1..1 --- A --- * -> C2 (K2,O2) ⇒ K2 is a key for A

can be changed to
C1(K1, O1), C2(K2,O2, K1) with no A !!
K1 in C2 cannot null

C1 (K1,O1) <- 0..1 --- A --- * -> C2 (K2,O2) ⇒ K2 is a key for A
can be changed to
C1(K1, O1), C2(K2,O2, K1) with no A !!
K1 in C2 can be NULL
example:
Student(sID, sname, GPA,cName1, cName2) ← apply to 0..2 college
College(cName, state, enrollment)

Association classes

Student(sID, sname, GPA)
College(cName, state, enrollment)
Apply(sID,cName)

Apply becomes
Apply(sID, cName, date, decision)

!!! Require a key for every "regular" class ⇒ Apply key is (sID, cName), AppInfo disappeared !!!

Self-association

Student(sID, sName, GPA)
sibling(sID1, sID2) ← no assumption on multiplicity
← (sID1, sID2) is key

College(sName, state, enrollment)
Branch(home, satellite) ← cName with <> roles
← satellite is a key here !

Subclasses

1/Subclass relations contain superclass key + specialized attributes
2/ Subclass relations contain all attributes
3/ One relation containing all superclass + subclass attributes

S (K pk, A) : superclass
S1(B) ← subclass
S2(C)

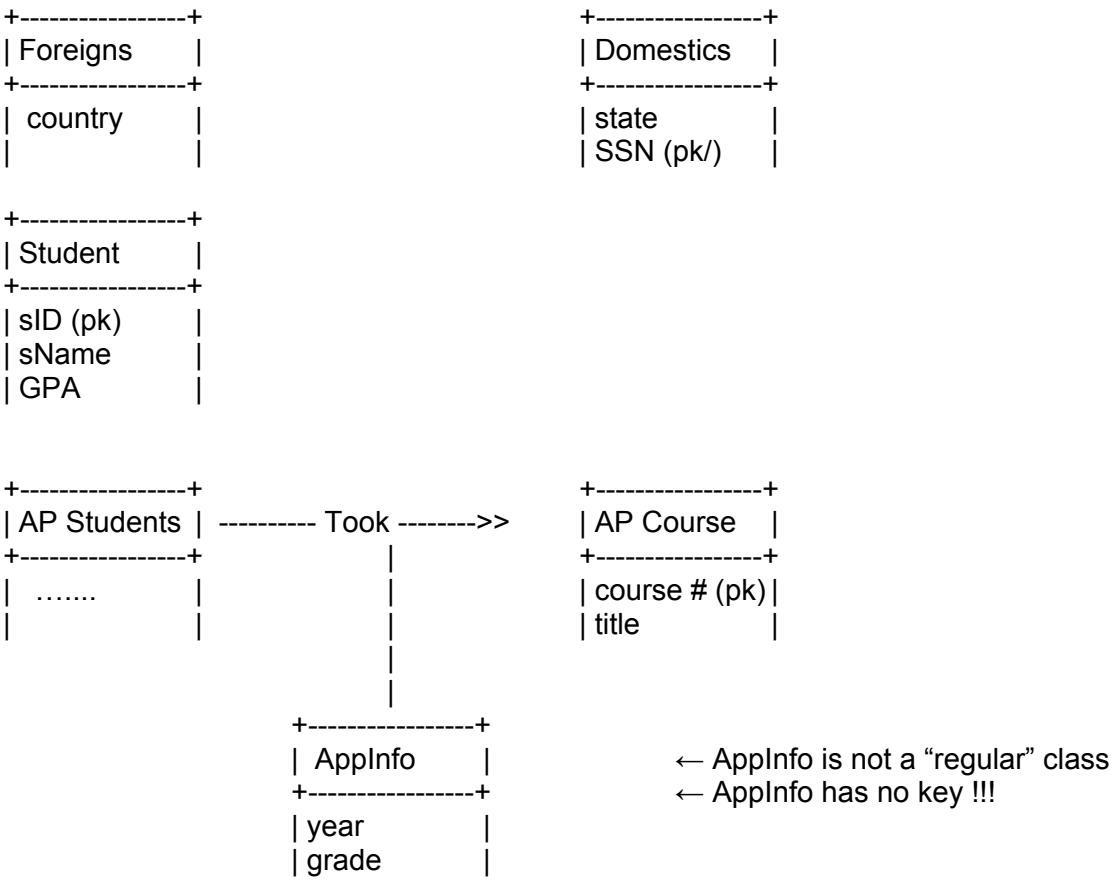
(1) S(K,A) S1(K,B) S2(K,C)
← one tuple for each object, +1 more for each time an object is in a subclass
(2) S(K,A) S1(K,A,B) S2(K,A,C)
← 1 tuple for each object + 1 more each time an object is in more than 1 subclass

(3) S(K,A,B,C) ← megarelation with NULL if not in class
← one tuple for each object regardless of subclassing

!!! Bet translation may depend on properties (incomplete/complete, overlapping/disjoint…) !!!

Heavily overlapping: use design (3)
Disjoint, complete: use design (2)
← we may not even need S(K,A) relation due to completness
Incomplete, overlapping: use design (1) or (2) or (3) ;-)

Example:



Student(sID, sName, GPA)
Foreigns(sID, country) ← Key inherited from superclass
Domestics(sID, state, SS#) ← 2 keys: sID, SS#
APStudent(sID)
APCourse(course#, title)
Took(sID, course#, year, grade)

If evey APStudent needs to take one course ….

Composition and aggregation

College -bd(1..1)---------- Department

College(cName, state) ← cName is key
Department(dName, building, cName)

INTRODUCTION TO DATABASE
- INDEXES

Primary mechanism to get improved performance from the database.
But beware of trade-off

A	B	C
cat	2	...
dog	5	…
cow	1	
dog	9	
cat	3	…
cat	8	
cow	6	
…		

Index = attached to a column or a group of column
In a query-condition, they return the row number where the condition is matched
Important if a table/query is used frequently
index = difference between full table scans and immediate location of tuples
⇒ orders of magnitude performance difference

!!! indexes are built automatically on primary keys!!!
← speed up join operations (from...)

!!! indexes are also build automatically on attributes declared as UNIQUE!!!
← that's why we label attributes as UNIQUE

Underlying data structures
- balanced trees (B trees, B+ trees) ← A = Value, A < value, v1 <= A <= v2
or - hash tables ← only for A = value !!!
← hash tables have constant loading time

Select sName
From Student
Where sID = 18942

where T.A == dog ← returns 2, 4

Example 2:

Select sID
From Student
Where sname = ‘Mary’ and GPA>3.9

(1) Index on sName ← possibly a hash or b-tree
(2) Index on GPA ← b-tree required because inequality
(3) Index on (sName, GPA)

Example 3:
Select sName, cName
From Student, Apply ← with a join !
Where Student.sID = Apply.sID

assume index on Apply.sID ….
Supposed we can get the sID in order and merge the sorted index

⇒ Query planning and optimization

Downsides of indexes
1) take extra disk space - marginal
2) index creation is time consuming (at creation) - medium
3) index maintenance - slow down transaction when tables are updated (but read/query are ok)

Picking which indexes to create
Function of
* size of table (and possible layout)
* data distributions (of values)
* query vs update load ← how often you query vs update

Physical design advisors
Input: database (statistics) and workload
Output: recommended indexes

database statistics
query or update → Query optimizer → Best execution plan with estimated cost
indexes

i.e recommend indexes where their downside is smaller than the upside !

SQL syntax
create Index <IndexName> on T(A)
Create Index <IndexName> on T(A1,A2,...,An)
Create Unique Index <IndexName> on T(A) ← check that A is unique
Drop Index <IndexName>

Is index useful?
Does any query involve a selection or join on this attribute? If so, then the index could be useful.

INTRODUCTION TO DATABASE
- CONSTRAINTS AND TRIGGERS

Constraints
Impose restrictions on allowable data, beyond those imposed by structure and types.

examples
0.0 < GPA < 4.0 ← range
decision attributes: 'Y', 'N', or null ← enumerate
size HS < 200 ⇒ not admitted in college where enrollment > 30000

Use contraints to
(1) data-entry errors (insert)
(2) correctness criteria (updates)
(3) enforce consistency throughout database (cross checking)
(4) tell system about data - values are unique - store the date to make it more efficient,
…

classifications
(1) non-null ← attribute cannot ave null values
(2) key ← uniqueness in column
(3) referential integrity (foreign key)
(4) attribute-based
(5) tuple-based constraints
(6) general assertions ← not implemented in any DBMS

declaring and enforcing constraints
Declaration
(1) with original schema (checked after bulk loading)
(2) or later (check on current db)
Enforcement
check after every 'dangerous' modifications
deferred constraint checking ← check after every transaction

Triggers = "Event-condition-ACtion rules"
When event occurs, check condition, if true, do action

examples:
(1) if enrollment > 35000 ⇒ reject all applicants
(2) insert application with GPA > 3.95 ⇒ accept automatically
(3) update sizeHS to be > 7000 ⇒ raise error

purpose are
(1) to move logic from application into DBMS
(2) to enforce complex constraints (expressiveness, constraint 'repair' logic

SQL coding
Create Trigger <name>
Before | After | Instead of <events>
[referencing-variables]
[for each row]
when (<condition>)
<action>

INTRODUCTION TO DATABASE
- CONSTRAINTS DEMO

College(cName, state, enrollment)
Student(sID, sName,GPA,sizeHS)
Apply(sID,cName,major,decision)

Null constraints

create table Student(sID int, sName text, GPA real not null, sizeHS int)

insert into Student values (123, 'Amy', 3.9,1000); ← ok
insert into Student values (234, 'Bob', 3.6, null); ← ok (null sizeHS)
insert into Student values (345, Craig',null,500); ← fails !!!

update Student set GPA = null where sID = 456 ← ok, because no match
update Student set GPA = null where sID = 123; ← fails

Key constraints

create table Student(sID int primary key, sName text, GPA real, sizeHS int)

insert into Student values (123, 'Amy', 3.9,1000); ← ok
insert into Student values (234, 'Bob', 3.6, 1500); ← ok
insert into Student values (123, Craig',3.5,500); ← fails !!!

update Student set sID = 123 where sName = 'Bob' ← fails
update Student sID = sID - 111 ← works, because order
update Student sID = sID + 111 ← fails, because order

create table Student (sID int primary key, sName text primary key, GPA real, sizeHS int);
← wrong !!! only 1 primary key allowed !!!
create table Student (sID int primary key, sName text unique, GPA real, sizeHS int);
← ok

create table College (cName text, state text, enrollment int,
primary key(cName, state))
← tuple key syntax

```
create table Apply(sID int, cName text, major text, decision text,
                   unique(sID,cName), unique(sID, major));

insert into Apply values (123, 'Stanford', 'CS', null);          ← ok
insert into Apply values (123, 'Berkeley', 'EE', null);          ← ok
insert into Apply values (123, 'Stanford', 'biology', null)      ← fails
insert into Apply values (234, 'Stanford', 'biology', null)      ← ok
insert into Apply values (123, 'MIT', 'EE', null)                      ← fails
insert into Apply values (123, 'MIT', 'biology', null);          ← ok, because previous one failed !!
insert into Apply values (123, null, null, 'Y')
insert into Apply values (123, null, null, 'N')                  ← ok, null don't count for keys !!!

update Apply set major = 'CS' where cName = 'MIT';               ← fails because 1st insert entry
```

Range

```
create table Student (sID int, sName test,
                   GPA real check (GPA <= 4.0 and GPA > 0.0),
                   sizeHS int check (sizeHS <5000));
```

Tuple-based constraints

```
create table Apply(sID int, cName text, major text, decision text,
                   check(decision = 'N' or cName <> 'Stanford' or major <> 'CS'));

insert into Apply values (123, 'Stanford', 'CS', 'N');
insert into Apply values (123, 'MIT', 'CS', 'Y');
insert into Apply values (123, 'Stanford', 'CS', 'Y');                ← fails

update Apply set decision = 'Y' where cName = 'Stanford';        ← fails
update Apply set cName = 'Stanford' where cName = 'MIT';                ← fails


create table Student (sID int, sName text,
                   GPA check (GPA is not null),                  ← key constraint
                   sizeHS int)
```

example: Try to enforce A is a key for T

```
create table T(A int check(A not in (select A from T)));         ← fails, fails, fails !
                                                                 ← check before or after insert A
```

← refers to T before creation

```
create table T(A int check ((select count(distinct A) from T) = ( select count(*) from T)));
```
← fails, refers to T before creation

referential integrity constraints

!!! Not supported yet, but in SQL standard !!!

```
create table Apply(sID int, cName text, major text, decision text,
                   check (sID in (select sID from Student)));
```
← ok? subquery in constraint !!!

```
create table College(cName text, state text, enrollment int,
                   check(enrollment >(select max(sizeHS) from Student)));
```

!!! Student table can change and as a result the constraint may not hold anymore !!!
← Changing sizeHS in student doesn't trigger the checking of this constraint
← Condition hold only at insert/update in table
!!! Maybe that's why it hasn't been implemented !!!

General assertion

!!! Not supported in an database !!!

```
create assertion Key
check ((select count(distinct A) from T ) = ( select count(*) from T)));

create assertion ReferentialIntegrity
check (not exists (select * from Apply where sID not in (select sID from Student)));
```
← not exists of bad things in subquery

```
create assretion AvgAccept
check (3.0 < (select avg(GPA) from Student
              where sID in
              (select SID from Apply where decision = 'Y')));

enforcement require monitor of
      Student.GPA changes
```

Referential integrity ← Important type of constraint
= integrity of references

= no dangling pointers
= foreign key constraint

Refential integrity from R.A to S.B:
each value in column A of table R must appear in column B of table S

exemple of referential integrity violation
Apply.sID not in Student.sID
Apply.cName not in College.cName

!!! A the referenced attribute is called the “foreign key” !!!

!!! B is usually required to be the primary key for table S or at least unique !!!

!!! Multi-attribute foreign keys are allowed !!!

Referential integrity enforcement (R.A to S.B)
Potentially violating modifications:
insert into R
delete from S
update R.A
update S.B

Delete from S
ex: delete tuple in Student
restrict(default), ← not allowed, generate an error
set null, ← set to null entries in Apply referencing to the Student
cascade ← cascade delete, i.e. delete entries in apply
← cascade using delete

Update S.B
ex: update sID in Student
restrict(default) ← not allowed
set null ← set reference to null
cascade ← update references, propagate to referencing values
← cascade using update

```
create table College (cName text primary key, state text, enrollment int);
create table Student (sID int primary key, sName text, GPA real, sizeHS int);
create table Apply (sID int references Student(sID),
                    cName text references College(cName),
```

```
                major text, decision text);


insert into Apply values(123, 'Stanford','CS','Y')      ← fails
                                                        ← no Student entries, no Stanford entries

insert into Student values (123, 'Amy', 3.9,1000);
insert into Student values (234, 'Bob', 3.6, 1500);
insert into College values('Stanford','CA',15000);

update Apply set sID = 345 where sID = 123;     ← fails, Apply.sID = 345 doesn't exist
update Apply set sID = 234 where sID = 123;     ← ok

delete from College where cName = 'Stanford';   ← fails, because Apply entries

drop table Student;                             ← error, first drop Apply table is required


create table Apply(sID int references Student(sID) on delete set null,
                cName text references College(cName) on update cascade,
                major text,
                decision text);
                        ← foreign key syntax
                        ← propagate changes in cName
                        ← delete referencing entries if a sID (Student tuple) is deleted
                        ← !!! still restrict mode for delete on cName and update on sID !!!

!!! Foreign key must refer to  a unique/key attribute !!!

delete from Student where sID > 200
                        ← Apply entries referring to sID > 200 are set to Null
update College set cName = 'Bezerkley' where cName = 'Berkeley';
                        ← Apply entries referring to 'Berkeley' are referring to 'Bezerkley'


create table T (A int, B int, C int,
                primary key (A,B),
                foreign key(B,C) references T(A,B) on delete cascade );
                        ← intra table referential integrity
                        ← delete entry with (?, B,C) if T(A,B) entry is deleted
insert into T values (1,1,1);
insert into T values (2,1,1);
insert into T values (3,2,1);
insert into T values (4,3,2);
```

… (5,4,3);
… (6,5,4);
… (7,6,5);
… (8,7,6);

delete T where A=1; ← deletes everything !!!!

INTRODUCTION TO DATABASE
- TRIGGERS

Triggers = Event-condition-action rules
when event occurs, check condition; if true, do action

(1) Move monitoring logic from apps into DBMS
(2) Enforce constraints
    Beyond what constraint system supports
    Automatic constraint “repair”

!!! Implementations vary significantly !!!

Triggers in SQL

Create Trigger name
Before | After | Instead of events ← triggered before, after instead of a specific events
[referencing-variables] ← old/new row/tables as var (4 var total max)
[For Each Row] ← executed once for each modified tuple | once if not present
When (condition)
action

!!! event = insert, delete, update of C1..Cn on T, etc. !!!
!!! old table = refers to the set of tuples that have been modified/deleted !!!
!!! new row = new inserted data with For each row’ directive !!!

!!! new, not old : new inserted data !!!
!!! no new, old: old deleted data !!!
!!! new, old : updated data !!!
Referential Integrity: R.A references S.B, cascaded delete

Row version of the trigger ← Triggers for each row
Create Trigger Cascade
After Delete On S
Referencing Old Row As O ← O = Old row that was modified in S

For Each Row ← Row level
[no condition]
Delete From R where A = O.B

Statement level ← Trigger only once
Create Trigger Cascade
After Delete On S
Referencing Old Table As OT
[For Each Row] ← Statement level instead of row level
[no condition]
Delete From R Where A in (select B from OT)
← select B because OT is complete tuple

Tricky issues
Row-level vs Statement-level
- New/Old Row and New/Old Table
- Before, Instead Of

Multiple triggers activated at same time (Which goes first?)
Trigger actions activating other triggers (chaining)
- Also self-triggering, cycles, nested invocations

Conditions in When vs. as part of action

!!! Implementations vary significantly. SQLLITE only at row level !!!

T(K,V) - K key, V value

Create Trigger IncreaseInserts
After Insert On T
Referencing New Row As NR, New Table As NT
For Each Row ← row level
When (Select Avg(V) From T) < (Select Avg(V) From NT)
Update T set V = V+10 where K=NR.K

← No statement-level equivalent
← Nondeterministic final state

INTRODUCTION TO DATABASE
- TRIGGERS
- DEMO

Postgress >
SQLite >>

- row-level only, immediate activation
- no old/new table
- no referencing clause. Old = Old Row. New = New Row
- trigger action : SQL statements in begin-end block

MySQL

- Row-level only, immediate activation
- no old/new table
-
-

College(cName, state, enrollment)
Student(sID, name, GPA, sizeHS)
Apply(sID, cName, major, decision)

Before and after; Insert, Delete, and Update

```
create trigger R1
after insert on Student
for each row
when New.GPA > 3.3 and New.GPA <= 3.6
begin
        insert into Apply values (New.sID, 'Stanford', geology', null);
        insert into Apply values (New.sID, 'MIT', 'biology', null);
end;

insert into Student values ('111', 'Kevin', 3.5, 1000);          ← trigger activates
insert into Student values ('222', 'Lori', 3.8, 1000);

insert into Student select sID+1, sName, GPA, sizeHS from Student;;
                                                                 ← trigger activates many times


create trigger R2
after delete on Student
for each row
begin
        delete from Apply where sID = Old.sID;
end;
delete from Student where sID > 500;                             ← trigger activates


create trigger R3                                                ← cascaded update !
after update of cName on College
for each row
```

```
begin
        update Apply
        set cName = New.cName
        where cName = Old.cName;
end;


update College set cName = 'The Farm' where cName = 'Stanford';        ← trigger activated
update College set cName = 'Bezerkeley' where cName = 'Berkeley';      ← trigger activated


create trigger R4
before insert on College                                               ← before insert!
for each row
when exists (select * from College where cName = New.cName)            ← key violation !?
begin
        select raise(ignore);                                  ← !!! Stop command under way !!!
end;


create trigger R5
before update of cName on College                                      ← before update!
for each row
when exists (select * from College where cName = New.cName)
begin
        select raise(ignore);                                  ← !!! Stop command under way !!!
end;

insert into College values ('Yale','CT', 10000);                   ← ok
insert into College values ('MIT', 'hello',10000)                  ← insert trigger activates

update College set cName = 'Berkeley' where cName = 'Bezerkeley';      ← ok
update College set cName = 'Stanford' where cName = 'The Farm';        ← fails, duplicate
update College set cName = 'Standford' where cName = 'The Farm';       ← ok, no duplicate


create trigger R6
after insert on Apply
for each row
when (select count(*) from Apply where cNAme = New.cName) > 10
begin
        update College set cName = cName || '-Done'                    ← concat operator
        where cName = New.cName;
end;
```

Chained trigger : R1 → R6 →

!!! Chaining can make changes very complicated and difficult to predict !!!

```
create trigger R7
before insert on Student                    ← before trigger
for each row
when New.sizeHS < 100 or New.sizeHS > 5000
begin
        raise(ignore);                      ← assume requested modification is erroneous
end;
```

```
create trigger R8
after insert on Student                             ← after trigger
for each row
when New.sizeHS < 100 or New.sizeHS > 5000
begin
        delete from Student where sID = New.sID;;;  ← assume modification was erroneous
end;
```

!!! R7 and R8 are equivalent !!!

!!! Triggers that activate at the same time …. assume both execute. Beware of inconsistency !!!

```
create trigger AutoAccept
after insert on Apply
for each row
when (New.cName = 'Berkeley' and
        3.7 < (select GPA from Student where sID = New.sID) and
        1200 < (select sizeHS from Student where sID = New.sID))
begin
        update Apply
        set decision = 'Y'
        where sID = New.sID
        and cName = New.cName;
end;
```

```
create trigger TooMany                              ← look for a threshold !!!
after update of enrollment on College
for each row
when (Old.enrollment <= 16000 and New.enrollment > 16000)
begin
```

```
        delete from Apply
                where cName = New.cName and major = 'EE";
        update Apply
                set decision = 'U'                      ← U for undecided
                where cName = New.cName
                and decision = 'Y';
end;
```

INTRODUCTION TO DATABASE
- TRIGGERS
- DEMO 2

T1(A)
T2(A)
T3(A)
T4(A)

Self-triggering

```
create trigger R1
after insert on T1
for each row
begin
        insert into T1 values (New.A+1);        ← trigger itself !!!
end;
```

!!! In SQLite, trigger can be activated only once, unless "pragma recursive_triggers = on;" !!!

!!! Need stop clause !!!

```
create trigger R1
after insert on T1
for each row
when (select count(*) from T1) < 10             ← self-triggering stop clause
begin
        insert into T1 values (New.A+1);
end;
```

Cycle

```
create trigger R1
after insert on T1
for each row
begin
        insert into T2 values (New.A+1);
```

```
end

create trigger R2
after insert on T2
for each row
begin
        insert into T3 values (New.A+1);
end

create trigger R3
after insert on T3
for each row
begin
        insert into T1 values (New.A+1);
end

pragma recursive_triggers = off;          ← R1 → R2 → R3  if 'recursive' trigerring is off
```

If 'pragma recursive_triggers = on;', put exit clause in R3

```
create trigger R3
after insert on T3
for each row
when (select count(*) from T1) < 100          ← exit clause
begin
        insert into T1 values (New.A+1);
end
```

Conflicts

= 2 triggers executed at the same time and modifying the same area

```
create trigger R1
after insert on T1
for each row
begin
        update T1 set A = 2;
end;

create trigger R2
after insert on T1                            ← triggers at same time as R1
for each row
when exists (select * from T1 where A = 2)
begin
        update T1 set A = 3;                  ← conflict with R1 !!!
```

```
end;

insert into T1 values (1);                    ← what triggers first?
```

!!! In SQLite, the trigger activate in the reverse order of their creation.!!!
!!! The last one declared first !!!

Nested trigger invocations

```
create trigger R1
after insert on T1
for each row
begin
        insert into T2 values (1);          ← activate second trigger before being done !!!
        insert into T3 values(1);
end;

create trigger R2
after insert on T2
for each row
begin
        insert into T3 values (2);                ← trigger R3 before completion
        insert into T4 values (2);
end;

create trigger R3
after insert on T3
for each row
begin
        insert into T4 values (3);
end;

insert into T1 values (0);
```

SQLite
!!! In SQLite the trigger SQL statement execute BEFORE the begin-end clause?!?!? ONly IN AFTER CLAUSE !!!

```
create trigger R1
after insert on T1
for each row
begin
```

insert into T2 select avg(A) from T2; ← function of number of 2s in the table
end;

insert into T1 select A+1 from T1

INTRODUCTION TO DATABASE
- TRANSACTIONS

Motivation:
Concurrent database access
Resilience to system failures

Concurrent database access

(1) Attribute-level inconsistency

update College
set enrollment = enrollment + 1000
where cName = ‘Stanford’

concurrent with

update College
set enrollment = enrollment + 1500
where cName = ‘Stanford’

A database is updated as follow: get tuple, modify tuple, write tuple.
⇒ if requests are sequential, X enrollment → X + 2500
if request are interleaved, X enrollment → X + 1000 or X + 1500
⇒ !!! 3 values for the same attribute are possible !!! ← attribute level

update Apply
set major=’CS’
where sID = 123

concurrent with

update Apply
set decision=’Y’
where sID = 123

A database is updated as follow: get tuple, modify tuple, write tuple.
⇒ if requests are sequential, both changes
if not, either value of the tuple is updated ← inconsistency at tuple level

update Apply
set decision='Y'
where sID in (Select sID From Student Where GPA>3.9)

concurrent with

update Student
set GPA=(1.1) * GPA
where sizeHS > 2500

Outcome function of which runs first ← table consistency

Insert into Archive Select * from Apply where decision='N';
Delete From Apply where decision='N';

concurrent with

select count(*) from Apply;
select count(*) from Archive;
← multi-statement inconsistency

Concurrency goal
= execute sequence of SQL statements so they appear to be running in isolation
→ simple solution is to execute them in isolation

!!! But we want to enable concurrency whenever safe to do so. !!!

!!! Beware of Multiprocessor, multi-threaded, asynchronous I/O !!!

Resilience to system failures
ex: crash in bulk load
ex: data is move from one table to the other
ex: lots of updates buffered in memory + power failure

We want guarantee all-or-nothing execution, regardless of failure

⇒ TRANSACTIONS

Transaction is a sequence of one or more SQL operations treated as a unit

- transactions appear to run in isolation ← atomicity
- if the system fails, each transaction's changes are reflected either entirely or not at all

SQL standard
- A transaction begins automatically on first SQL statement
- On 'commit' transaction ends and new one begins ← commit
- current transaction ends on session termination
- 'autocommit' turns each statement into a transaction

INTRODUCTION TO DATABASE
- TRANSACTIONS

ACID Properties
Atomicity
Consistency
Isolation
Durability

Isolation
= serializibility
= operations may be interleaved , but execution must be equivalent to some sequential (serial) order of all transactions
!!! property is guaranteed using locking !!!

!!! Serializability does not guarantee which transaction of concurrent requests is executed first!!!
!!! Beware the first transaction may change the result of the other one!!!
← order can be enforced in code

Durability
If a system crashes after transaction commits, all effects of transaction remain in database
← use logging (like extfs4)

Atomicity
Each transaction is "all-or-nothing" never left halt done
← use logging

!!! Transaction executed once, may need to be re-executed by client because of crash !!!

Atomicity - Transaction rollback (=Abort) ← **goes back to previous commit**
= Undoes partial effects of transaction
Can be system or client initiated

!!! a rollback doesn't return cash which has been distributed !!!

!!! Transaction should run to completion relatively quickly !!!

Consistency
=each client, each transaction (1) can assume all constraints hold when transaction begins (2) must guagantee all constraints hold when transaction end

serializability ⇒ constraints always hold

INTRODUCTION TO DATABASE
- TRANSACTIONS
- ISOLATION LEVELS

isolation level
* read uncommitted ← weakest
* read committed ←
* repeatable read ← stronger
* serializability ← strongest

Issue with strong isolation level:
The stronger the isolation level the bigger the overhead (through locking) and the worse the concurrency

!!! Isolation level is per transaction (and per client) !!!

Dirty reads ← **apparent during read uncommitted**
"Dirty" data item: written by an uncommitted transaction
~ partially written in memory but not committed to disk/database (??)

Isolation Level Read uncommitted

Update Student
Set GPA = (1.1)*GPA where sizeHS>2500 ← T1

concurrent with

(1) ← default is serializable mode
Select Avg(GPA) from Student ← T2

⇒ serialization T1,T2 or T2,T1

(2)
Set Transaction Isolation Level Read Uncommitted;
Select Avg(GPA) from Student;

⇒ We don't have T1,T2 or T2, T1. We have T2 takes result status of the memory as is. Even if T1 is half done !!!

← used if we want to have a rough idea

example
- R(A) contains the tuple {(1),(2)}
- (T1) update R set A = 2*A
- (T2) Set Transaction Isolation Level Read Uncommitted;
  select avg(A) from R;

T2,T1 ⇒ avg = 1.5
T1,T2 ⇒ avg = 3
T1 (1),T2 ⇒ avg = 2 ← the update starts with the first tuple
T1 (2), T2 ⇒ avg = 2.5 ← the update starts with the second tuple

!!! The order of an update is random !!!

← may start with first, last, middle tuple

Isolation level Read Committed
A transaction may NOT perform dirty reads
Read values which have been committed in the database.
Note that the committed values in the database may change due to concurrency.
Still does not guarantee global serializability

← would be if T2 was only one statement (?)

Update Student
Set GPA = (1.1)*GPA where sizeHS>2500 ← T1

concurrent with

Set Transaction Isolation Level Read Committed;
Select Avg(GPA) from Student; ← all of this is T2
Select Max(GPA) form Student;

⇒ Avg may read as if T2, T1, while max may read as if T1, T2

← T1 has been committed while T2 was executing

example
- R(A) = { (1),(2) }
- S(B) = { (1),(2) }
- (T1) update R set A = 2*A;
  update S set B = 2*B; ← commit happens at the end of this SQL statemt

- (T2) set transaction isolation level read committed
  select avg(A) from R;
  select avg(B) from S;

Isolation level Repeatable Read
A transaction may not perform dirty reads
An item read multiple times cannot change value
← previously we read GPA and it changed
Still does not guarante serialization

Update Student set GPA=(1.1)*GPA;
update Student Set sizeHS=1500 where sID=123;

concurrent with

Set transaction Isolation Level Repeatable Read;
Select Avg(GPA) From Student;
Select Avg(sizeHS) From Student;

!!! Not equivalent to serialization because GPA can be value before and sizeHS value after T1 !!!

!!! Values (tuples) which have been read once are locked.!!!
!!! If new tuples are inserted they are not locked !!!
← locked means not transaction can change those tuples
← beware of inserts
← beware of phantom tuples !

!!! if we have a delete of LOCKED TUPLES in T1, instead of insert, then T1 and T2 are serialized due to isolation level !!!
← delete of unlocked tuples is ok (?)
← other non delete statement in commit are executed until blocking statement (?)
← commit may be blocked half way ! (?)

Read only transactions
Use to optimize performance
Independent of isolation level

example:
Set Transaction Read Only;
Set Transaction Isolation Level Repeatable Read;
Select Avg(GPA) From Student;

Select Max(GPA) From Student;

	dirty reads	non-repeatable reads	phantoms
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeat Read	N	N	Y
Serializable (strongest)	N	N	N

Weaker isolation levels
+ increased concurrency
+ decrease overhead
+ increase performance
- weaker consistency guarantees
- some systems have default Repeatable Read

Isolation level
per transaction
per “eye of the beholder” (client?)

INTRODUCTION TO DATABASE
- VIEWS

Three-level vision of the database

Physical - conceptual - logical vision of database
physical: pages on disk
conceptual: tables, relations, etc
logical: query over relation

Why use views?
- Can hide some data on some users
- Make some queries easier / more natural
- Modularity of database access

!!! Real application tends to use lots and lots of views !!!

Defining and using views
View V = ViewQuery(R1, …., Rn)

Schema of V is schema of query result
Query Q involving V, conceptually: ← Q is an ad-hoc query
V := ViewQuery(R1,...,Rn) ← populate V, a temporary table
Evaluate Q

!!! In reality, Q rewritten to use R1, ..,Rn instead of V !!!

← the query is executed each time

SQL syntax
Create view Vname(A1,...,An) as ← optional attribute names for View
<Query> ← SQL

example:
College(cName, state, enrollment)
Student(sID, name, sizeHS)
Apply(sID,cName, major, decision)

```
create view CSaccept as
select sID, cName
from Apply
where major = 'CS' and decision = 'Y';

select Student.sID, sName, GPA
from Student, CSaccept
where Student.sID = CSaccept.sID and cName = 'Stanford' and GPA < 3.8;
```

← refers to CSaccept as if a table

equivalent to

```
create temporary table T as
select sID, cName
from Apply
where major = 'CS' and decision = 'Y';

select Student.sID, sName, GPA
from Student, T
where Student.sID = T.sID and cName = 'Stanford' and GPA < 3.8;

drop table T
```

equivalent to

```
select Student.sID, sName, GPA
from Student, (select sID, cName from Apply
               where major = 'CS' and decision = 'Y') as CSaccept
```

where Student.sID = CSaccept.sID and cName = ‘Stanford’ and GPA < 3.8;

equivalent to

select Student.sID, sName, GPA
from Student, Apply
where major = ‘CS’ and decision = ‘Y’
and Student.sID = Apply.sID and cName = ‘Stanford’ and GPA < 3.8;

!!! This is exactly how the view is being computed, or the query rewritten !!!

view that references other views

create CSberkeley as
select Student.sID, sName, GPA
from Student, CSaccept
where Student.sID = CSaccept.sID and cName = ‘Berkeley’ and sizeHS > 500;

select * from CSberkeley where GPA > 3.8;

equivalent to

select * from
(select Student.sID, sName, GPA
from Student, (select sID, cName from Apply
where major = ‘CS’ and decision = ‘Y’) as CSaccept
where Student.sID = CSaccept.sID and cName = ‘Berkeley and sizeHS > 500) CSberkely
where GPA > 3.8;
← CSaccept is expended to its view definition

deletion of views
drop view CSaccept;

!!! beware in most database, this operation breaks views that are using this view !!!

Neat view trick (Mega view)

create view Mega as
select College.cName, state, enrollment, Student.sID, sName, sizeHS, major, decision
from College, Student, Apply
where College.cName = Apply.cName and Student.sID = Apply.sID;
← join info from all 3 tables

```
select sID, sName, GPA, cName
from Mega
where GPA > 3.5 and major = 'CS' and enrollment > 15000;
                                    ← query without join
```

equivalent to

```
select Student.sID, sName, GPA, College.cName
from College, Student, Apply
where College.cName = Apply.cName and Student.sID = Apply.sID
        and GPA > 3.5 and major = 'CS' and enrollment > 15000;
```

!!! With views, complex queries can be written in a simple way !!!

INTRODUCTION TO DATABASE
- VIEW
- VIEW MODIFICATIONS

Querying views:
* Once V is defined, we can reference V like any table
* Queries involving V rewritten using the tables

Modifying views
* Once V defined, can we modify V like any table?
* Doesn't make sense: V is not stored
* Has to make sense: views are some users' entire "view" of the database
⇒ Solution: Modifications to V rewritten to modify base tables

!!! Not as straight forward !!!

V → (modification) → V'
R1,...,Rn → (modification) → R1',...., Rn'

Usually we can do the translation, but sometimes we don't know

example
R(A,B) → (1,2)
V = \project_{A}(R) → 1
insert (3) in V
But what tuple (3,?) do we insert in R?

example
R(A) → 1,3,5

V = avg(A) =7
insert (7) in V
But what tuples do we insert in R?

2 modification methods

Modifications to V rewritten to modify base tables
(1) rewriting process specified explicitly by view creator
+ Can handle all modifications
- No guarantee of correctness (or meaningful)
← instead-of triggers
← rules

(2) Restrict views and modifications so that translation to base table modifications is meaningful and unambiguous
+ No user intervention
- Restrictions are significant
← SQL standard
← automatic view modifications

INTRODUCTION TO DATABASE
- VIEW
- VIEW MODIFICATIONS
- USING TRIGGERS

deletion

```
create view CSaccept as                          ← view creation
select sID, cName
from Apply
where major = 'CS' and decision = 'Y';

delete from CSaccept where sID = 123;            ← doesn't work

create trigger CSacceptDelete                    ← instead-of trigger on deletion
instead of delete on CSaccept                    ← intercept deletion on view
for each row
begin
        delete from Apply                        ← translation
        where sID = Old.sID                      ← Old = tuple to be deleted
        and cName = Old.cName                    ← other conditions based on Old
        and major = 'CS' and decision = 'Y';     ← conditions defined in view
end;
```

```
delete from CSaccept where sID = 123;                ← works
```

update

```
update CSaccept set cName = 'CMU' where sID = 345;          ← doesn't work

create trigger CSacceptUpdate
instead of update of cName on CSaccept          ← intercept update on view
for each row
begin
        update Apply
        set cName = New.cName
        where sID = Old.sID
        and cName = Old.cName
        and major = 'EE" and decision = 'N';        ← incorrect translation, should be 'CS' + 'Y'
end;

update CSaccept set cName = 'CMU' where sID = 345;      ← works
```
!!! The translation needs to be correct. !!!

insert

```
create view CSEE as
select sID, cName, major
from Apply
where major ='CS' or major = 'EE';

insert into CSEE values (111, 'Berkeley', 'CS')          ← doesn't work !

create trigger CSEEinsert
instead of insert on CSEE
for each row
begin
        insert into Apply values (New.sID, New.cName, New.major, null);
end;

insert into CSEE values (111, 'Berkeley', 'CS')          ← works

insert into CSEE values (222,'Berkeley','biology');      ← works, but shouldn't
                                                         ← doesn't show in the view

drop trigger CSEEinsert;                                 ← drop previous trigger
```

```
create trigger CSEEinsert
instead of insert on CSEE
for each row
when New.major = 'CS' or New.major = 'EE'          ← condition added to trigger
begin
        insert into Apply values (New.sID, New.cName, New.major, null)
end;

insert into CSEE values (333,'Berkeley','biology');      ← now no effect = works correctly

insert into CSEE values (333,'BErkeley','EE')            ← ok, modify Apply table
```

Ambiguous modifications

← When modification should not be done of the view

example 1:

```
create view HSgpa as
select sizeHS, avg(gpa) as avgGPA          ← aggregation
from Student
group by sizeHS;                           ← aggregation
```

!!! Aggregation operation = should prevent update/insert in view !!!

```
update HSgpa set avgGPA = 3.6 where sizeHS = 200;      ← doesn't make sense?!!!
                                                       ← how to modify underlying table?
```

example 2:

```
create view Majors as
select distinct major from Apply;

insert into Majors values ('chemistry');          ← Which student should be in chemistry?
```

!!! ambiguous because distinct, 1 column !!!

example 3:

```
create view NonUnique as
select * from Student S1
where exists ( select * from Student S2
               where S1.sID <> S2.sID
               and S2.GPA = S1.GPA and S2.sizeHS = S1.sizeHS);
```

```
delete from NonUnique where sName = 'Amy';
                                    ← if you delete Amys, you would delete matching entries too!
```

!!! Deleting an entry in the view, delete an entry in the inner query ⇒ modification impossible !!!

view with join

```
create view Berkeley as
select Student.sID, major
from Student, Apply                                    ← join of 2 tables
where Student.sID = Apply.sID and cName = 'Berkeley';


create trigger BerkeleyInsert
instead of insert on Berkeley                          ← instead-of-insert trigger
for each row
when New.sID in (select sID from Student)              ← condition due to join !!!
begin
        insert into Apply values (New.sID, 'Berkeley', New.major, null);
end;

insert into Berkeley                                   ← batch insert
        select sID, 'psychology' from Student
        where sID not in (select sID from Apply where cName = 'Berkeley');

create BerkeleyDelete                                          ← delete application, not
student !
instead of delete on Berkeley                          ← instead-of-delete trigger
for each row
begin
        delete from Apply
        where sID = Old.sID and cName = 'Berkeley' and major = Old.major;
end;


delete from Berkeley where major = 'CS';               ← delete application, not student!!!

create trigger BerkeleyUpdate
instead of update of major on Berkeley                 ← intercept updates on Berkeley
for each row                                           ← intercept on major
begin
        update Apply
        set major = New.major
```

```
        where sID = New.sID and cName = 'Berkeley' and major = Old.major;
end;

update Berkeley set major = 'physics' where major = 'psychology';          ← works!
```

!!! If you try to modify another attribute sID, then error !!!

Constraint violations

```
drop trigger CSEEinsert;

create trigger CSEEinsert
instead of insert on CSEE
for each row
when New.major = 'CS" or new.major ='EE'
begin
        insert into Apply values (New.sID, New.cName, New.major, null)   ← null
end;

insert into CSEE values (123, 'Berkeley','CS');
                                ← 'not null' constraint kicks in at execution
```

!!! Beware of 'not null' constraints, key constraints, …. !!!

INTRODUCTION TO DATABASE
- VIEW
- VIEW MODIFICATION
- AUTOMATIC VIEW MODIFICATIONS

Supported on mysql, but no postgresql, or sqlite

Restrictions in SQL standard for updatable views

1) Select (no DIstinct) on sigle table T
2) Attributes not in view can be NULL or have a default value
3) Subqueries must not refer to T ← refer to other table ok
4) No GROUP BY or aggregation

example 1:

```
create view CSaccept as                                    ← updatable view
select sID, cName
from Apply
```

```
where major = 'CS' and decision = 'Y';
delete from CSaccept where sID = 123;                    ← works immediately
```

insert

```
create view CSEE as
select sID, cName, major
from Apply
where major = 'CS' and decision = 'Y';

insert into CSEE values (111,'Berkeley','CS')                    ←works immediately
insert into CSEE values (111,'Berkeley','psychology')            ← works when shouldn't !!!
```

```
insert into CSaccept values(333,'Berkeley');        ← put null in other fields !!! (Not good)
```

⇒ '...with check option;'

```
create view CSaccept2 as                            ← updatable view
select sID, cName
from Apply
where major = 'CS' and decision = 'Y'               ← should be fields values
with check option;                                  ← get condition field values? NO !
                                                    ← check data is inserted into the view

insert into CSaccept2 values(444,'Berkeley');       ← put null in other fields !!! (Not good)
```

!!! Return a check option failed error, because the view has not changed !!!

```
create view CSEE2 as
select sID, cName, major
from Apply
where major = 'CS' and decision = 'Y'
with check option;

insert into CSEE2 values(444,'Berkeley','psychology');     ← generate error
insert into CSEE2 values(444,'Berkeley','CS');             ← ok insert
```

Views not allowed to be modified

example 1/3:

```
create view HSgpa as
select sizeHS, avg(gpa)                             ← aggregate
from Student
group by sizeHS;                                    ← aggregate
```

```
delete from HSgpa where sizeHS < 500;          ← returns an error
insert into HSgpa values (3000, 3.0)           ← returns an error
```

example 2/3:

```
create view Majors as
select distinct major from Apply;

insert into Majors values ('chemistry');       ← returns an error
delete from Majors …                           ← returns an error
```

example 3/3:

```
create view NonUnique as
select * from Student S1
where exists ( select * from Student S2        ← Student S1 is referenced in subquery
               where S1.sID <> S2.sID
               and S2.GPA = S1.GPA and S2.sizeHS = S1.sizeHS);

delete from NonUnique where sName = 'Amy';     ← returns an error
                                               ← they are several ways to implement this
                                               ← none would be correct!

create view Bio as
select * from Student
where sID in (select sID from Apply where major like 'bio%');
                                               ← Student is not referenced in subquery
                                               ← This view is allowed to be modified
delete from Bio where sName = 'Bob';           ← delete tuple in Student
                                               ← delete student not application (Apply)
```

!!! Student tuple is erased, but Apply tuple referring to that student is still present !!!

```
                                               ← could have used a cascaded delete

insert into Bio values (555, 'Karen', 3.9, 1000);   ← runs correctly but … no tuples in view
                                                    ← use check option in view!
```

!!! delete from view ok, but insert in view nok in automatic view modification !!!

```
create view Bio2 as
select * from Student
where sID in (select sID from Apply where major like 'bio%')
which check option;                            ← ok,
```

```
insert into Bio2 values (666,'Lori',3.9,1000);          ← returns an error
```

!!! with check option has an impact on performance ⇒ not included always !!!

```
create view Stan(sID,aID,sName, major) as          ← attribute names  set explicitly (sID→aID)
select Student.sID, Apply.sID,sNAme, major
from Student, Apply
where Student.sID = Apply.sID and cName = 'Stanford';

update Stan set sName = 'CS major' where major = 'CS';
                        ← for update, use correct underlying attribute names on base tables

update Stan set aID = 666 where aID = 123;
                        ← aID maps to Apply table, but not Student (where clause)

create view Stan2(sID,aID,sName, major) as                ← attribute names  set explicitly
(sID→aID)
select Student.sID, Apply.sID,sNAme, major
from Student, Apply
where Student.sID = Apply.sID and cName = 'Stanford'
with check option;                                        ← … with check options
                                                          ← check view is modified

update Stan2 set aID = 777 where aID = 678;               ← return check option failed

insert into Stan(sID,sName) values (777,'Lance')          ← with Stan view, not Stan2
                                                          ← insert into Student table
                                                          ← no insert in view

insert into Stan2(sID,sName) values (888,'Mary')          ← return check option error

insert into Apply values(888,'Stanford','history','Y');
insert into Stan2(sID,sName) values (888,'Mary');         ← works !!!

insert into Apply values(999,'MIT','history','Y');
insert into Stan2(sID,sName) values (999,'Nancy');        ← return check option error (MIT)

delete from Stan where sID = 678;                         ← error, join view = ambiguous
```

INTRODUCTION TO DATABASE
- VIEWS
- MATERIALIZED VIEWS

!!! Used for performance improvement !!!

Virtual views

View V = ViewQuery(R1,...,Rn)
Schema of V is schema of query result
Query Q involving V, conceptually:
    V := ViewQuery(R1,...,Rn);
    Evaluate Q;
In reality, Q rewritten to use R1, …, Rn (base tables) instead of V

Materialized views
View V = ViewQuery(R1, …,Rn)
Create table V with schema of query result
Execute ViewQuery and put results in V
Queries refer to V as if it is a table
but
V could be very large ← beware of memory space
Modifications to R1, ...Rn ⇒ recompute or modify V

Create Materialized View CA-CS as
select C.cName, S.sName
From College C, Student S, Apply A ← 3 way join
where C.cName = A.cName and S.sID = A.sID and C.state = 'CA' and A.major = 'CS'

+ CA-CS is a table
- CA-CS is not updated when C,S, or A are changed
    ⇒ Modifications to base data invalidate view
    ex: College: inserts X, deletes X, updates(cName,state)
        Student: inserts X, deletes, updates(sNAme,sID)
        Apply: inserts, deletes, updates(cName,sID,major)

!!! Generate assertions ~ Materialized view !!!

Modifications on materialized views?
    Good news: just update the stored table
    Bad news: base tables must stay in sync
        ← same issues as with virtual views

Picking which materialized views to create

(Efficiency) benefits of materialized view depend on:
    Size of database/data
    Complexity of view
    Number of queries using view ← the more the better for materialized
    Number of modifications affecting view
    Also "incremental maintenance" versus full re-computation

!!! Query-update trade-off !!!
← speed up queries, but slow down updates

Automatic query rewriting to use materialized views

```
create materialize view CA-Apply as
select sID, cName, major
from Apply A
where cName in
        (select cName From College where state='CA")


select distinct S.sID, S.GPA
from College C,Student S, Apply A
where C.cName= A.cName and S.sID=A.sID
and S.GPA>3.5 and C.state = 'CA" and A.major = 'CS'
```
←- use materialized view to speed up query

```
select distinct S.sID, S.GPA
from Student S, CA-Apply A
where S.sID = A.sID and S.GPA>3.5 and A.major = 'CS';
```
←- the view has preprocessed part of the task

Why use materialized views?
Hide some data from some users
make some queries easier/more natural
modularity of database access
Improve query performance ← pre-processing of parts of the query

---

INTRODUCTION TO DATABASE
- DATABASE AUTHORIZATION

!!! Make sure users see only the data they are supposed to see !!!

!!! Guard the database against modifications by malicious users !!!

Users have privileges; can only operate on data for which they are authorized
select on R or select (A1,...,An) on R ← read privilege
insert on R or insert(A1,...,An) on R
update on R or update(A1,...,An) on R
delete on R

!!! Delete privilege never has attributes attached to it, because we only delete entire tuples !!!

example

```
update Apply
set dec = 'Y'
where sID in ( select sID
               from student
               where GPA > 3.9)
```

⇒ Required privileges are
Apply: update(dec), select(sID)
Student: select(GPA, sID)

example 2:

```
delete from Student
where sID not in (      Select sID from Apply)
```

⇒ Required privileges are:
Student: delete, select(sID)
Apply: select(sID)

example 3:
Select student info for Stanford Applicants only

```
create view SS as
select * from Student
where sID in ( select sID from Appy
               where cName = 'Stanford')
```

⇒ Required privileges are:
SS : select(*)

!!! Authorization is one of the main usage for views !!!

example 4:
Delete Berkeley applications only

```
create view BA as
select * from Apply
where cName = 'Berkeley'
```

⇒ Required privileges are:
BA: delete

!!! You must have updatable views to provide modification privileges through view !!!

Obtaining privileges
Relation creator is owner
Owner has all privileges and may grant privileges to other users

grant privs on R to users
[with grant option] ← can grant to other users less privilege

example:

grant select(SID), delete on R public

Revoking privileges

revoke privs on R from users
[cascade | restrict] ← restrict is the default

U1 → (grant privileges to) → U2 → (grant privileges to) → U3
← grant diagram

!!! Grant diagram is used to cascade the revoke command correctly !!!

cascade: also revoke privileges granted from privileges being revoked (transitively), unless also granted from another source

!!! Users can be granted the same privilege by different administrators !!!
!!! cascade does not always revoke any privileges !!!

restrict: disallow (privileges?) if cascade would revoke any other privileges
← restrict is the default
← manually revoke the privileges BOTTOM UP in the privilege graph
← If multi source for privilege, then you can restrict at one level higher without error

!!! Each users are associated with a list of which parent granted which rights !!! ← really?
!!! If revoke on parent with cascade option, then propagated to child !!! ← really?

INTRODUCTION TO DATABASE
- RECURSION IN SQL
- BASIC RECURSIVE <WITH> STATEMENT

SQL is not a “turing complete” language ← any computation can be computed in language
* Simple, convenient, declarative
* expressive enough for most database queries
* but basic SQL can’t express unbounded computations

example 1: ancestors
ParentOf(parent, child)
Find all of Mary’s ancestors

parents ok
grantparents = two instances of parentOf
Three

example 2: company hierarchy
employee(ID,salary)
manager(mID,eID) ← eID = employee ID
project(name,mgrID)
Find total salary cost of project ‘X’

Q: How deep is the tree of the hierarchy?

example 3: airline flights
Flight(orig,dest, airline,cost)
Find cheapest way to fly from ‘A’ to ‘B’

SQL WITH statement

```
with    R1 as (query-1),
        …
        Rn as (query-n)
<query involving R1,..,Rn (and other tables)>
```

SQL WITH Recursive statement

```
with recursive                                  ← any of the Ri can be recursive
        R1 as (query-1),
        …
        Rn as (query-n)
<query involving R1,..,Rn (and other tables)>

With recursive
        R as (base query                        ← base query no reference toR
                union                           ← duplicate eliminating union (<> union all)
```

recursive query) ← reference R
<query involving R (and other tables)>

!!! union eliminate duplicate and make the recurrence converge !!!

!!! The recursion stops when there is nothing new to add !!!

INTRODUCTION TO DATABASE
- RECURSION IN SQL
- BASIC RECURSIVE <WITH> STATEMENT
- DEMO

Supported by postgres and not mysql or sqlite

example 1/3:
ParentOf(parent,child)
Find all of Mary's ancestor

```
with recursive
        Ancestor(a,d) as (      select parent as a, child as d from ParentOf ← seed of recursion
                                union
                                select Ancestor.a, parentOf.child as d          ← A from 1st iteration
                                from Ancestor, ParentOf
                                where Ancestor.d = ParentOf.parent)
                                ← a and d  are attribute names in Ancestor table/view
                                ← a = ancestor, d = descendant
```

!!! Goes down in Ancestor tree !!!

```
select a from Ancestor where d = 'Mary'
```

example 2/3: company hierarchy
employee(ID,salary)
manager(mID,eID) ← eID = employee ID
project(name,mgrID)
Find total salary cost of project 'X'

```
with recursive
        Superior as (   select * from Manager               ← ~ Ancestor
                        union                               ← recursive here
                        select S.mID, M.eID
                        from Superior S, Manager M
                        where S.eID = M.mID)
select sum(salary)
from Employee
where ID in ( select mgrID from Project where name = 'X'
                union                                       ← nothing recursive here
                select eID from Project, Superior
                where Project.name - 'X' and project.mgrID = Superior.mID)
```

…. better yet …

```
with recursive
        Xemps(ID) as (      select mgrID as IF from project where name = 'X'    ← seed of rec
                            union
                            select distinct eID as ID
                            from Manager M, Xemps X
                            where M.mID = X.ID)
        Yemps(ID) as …
        Zemps(ID) as ...
                                                ← we could create the views/tables for each project
select sum(salary)
from employee
where ID in (select ID from Xemps)

select 'Y-total', sum(salary)
from Employee
where ID in (Select ID from Yemps)
union
select 'Z-total', sum(salary)
from Employee
where ID in (select ID from Zemps)
```

example 3/3: Airline flights

Flight(orig, dest, airline, cost)
Find cheapest way to fly from 'A' to 'B'

```
with recursive
        Route(orig, dest, total) as
                (       select orig, dest, cost as total from Flight
                        union
                        select R.orig, F.dest, cost+total as total
                        from Route R, Flight F
                        where R.dest = F.orig)

select min(total) from Route
where orig = 'A' and dest = 'B'
```

… better yet …

```
with recursive
        FromA(dest, total) as
                (       select dest, cost as total from Flight where orig = 'A'
```

```
                union
                select F.dest, cost+totla as total
                from FromA FA, flight F
                where FA.dest = F.orig)
select min(total)  from FromA where dest = 'B'
```

…. or again …

```
with recursive
        ToB(orig, total) as
                (       select orig, cost as totla
                        from Flight
                        where dest = 'B'
                        union
                        select F.orig, cost + total as total
                        from Flight F, ToB TB
                        where F.dest = TB.orig)
select min(total) from ToB where orig = 'A'
```

Infinite loop/recursion

In example 3 above, assume we have a loop in the route

```
with recursive
        Route(orig, dest, total) as
                (       select orig, dest, cost as total from Flight
                        union
                        select R.orig, F.dest, cost+total as total
                        from Route R, Flight F
                        where R.dest = F.orig
                        and cost+total < all(select total from Route R2
                                        where R2.orig = R.orig and R2.dest = F.dest))

select min(total) from Route
where orig = 'A' and dest = 'B'
                                ←- doesn't work !!!

with recursive
        Route(orig, dest, total) as
                (       select orig, dest, cost as total from Flight
                        union
                        select R.orig, F.dest, cost+total as total
                        from Route R, Flight F
                        where R.dest = F.orig
                                )
```

```
select min(total) from Route
where orig = 'A' and dest = 'B' limit 20                ← limit 20
```

… better solution …

add the number of hops/length

```
with recursive
        Route(orig, dest, total,length) as
                (       select orig, dest, cost as total, 1 from Flight
                        union
                        select R.orig, F.dest, cost+total as total, R.length+1 as length
                        from Route R, Flight F
                        where R.dest = F.orig and R.length < 10
                                                )

select min(total) from Route
where orig = 'A' and dest = 'B'
```

INTRODUCTION TO DATABASE
- RECURSION IN SQL
- BASIC RECURSIVE <WITH> STATEMENT
- NONLINEAR AND MUTUAL RECURSION

Example : Ancestor

```
with recursive
        Ancestor(a,d) as (select parent as a, child as d from ParentOf
                                union
                                select A1.a, A2.d
                                from Ancestor A1, Ancestor A2        ← 2 times the same table !!!
                                where A1.d = A2.d)

select a from Ancestor where d = 'Mary';
```

!!! 2 times the same table ⇒ non linear version !!!

Nonlinear (versus linear)
+ Query looks cleaner
+ converges faster ← logarithmic, not linear progression log_2(X) instead of X iter
- Harder to implement

SQL standard only requires linear

Mutual recursion
R1 refers to R2
R2 refers to R1

example: hubs and authorities
link(src, dest)
HubStart(node)
AuthStart(node)
Hub node: points to >=3 authority node
Authority point to >= 3 hub
Normal nodes: all other

```
with recursive
        Hub(node) as (      select node from HubStart
                            union
                            select src as node from Link L
                            where src not in(select node from Auth)      ← Hub xor Auth
                            and dest in (select node from Auth)  ← refers to Auth
                            group by src having count(*) >= 3),
        Auth(node) as (     select node from AuthStart
                            union
                            select dest as node from link L
                            where dest not in (select node from Hub)     ← Hub xor Auth
                            and src in (select node from Hub)            ← refers to Hub
                            group by dest having count(*) >= 3)
select * from Hub;
```

!!! not in … = negative recursive subqueries ⇒ not supported by SQL standard !!!

Example: recursion with aggregation

```
with recursive
        R(X) as (select x from P
                        union
                        select sum(x) from R)
select * from R;
```

?????

!!! disallowed: recursive subqueries (negative), aggregation !!!

INTRODUCTION TO DATABASE
- ONLINE ANALYTICAL PROCESSING (OLAP)
- INTRO

OLTP: Online Transaction Processing
- short transactions
- simple queries
- touch small portions of data
- frequent updates

ex: find GPA of a student

OLAP: Online Analytical Processing ← other extreme to OLTP
- Long transactions
- complex queries ← datamining
- touch large portion of the database
- infrequent updates ←- sometimes no updates at all

Data warehousing

Bring data from operation (OLTP) sources into a single “warehouse” for OLAP analysis

ex: OLTP database at points of sales

ex: OLAP : duplicating data from operational sources (point of sales) and do analysis

Decision support system (DSS)

Infrastructure for data analysis

e.g. data warehouse tuned for OLAP

Star schema

* Fact table ← 1 table with several foreign keys

updated frequently, often append-only, very large

example: sales transactions, course enrollments, page views

← ~ logging

* Dimension tables

updated infrequently , not as large

example: stores, items, customers, students ← ~ objects

courses, web pages, users, advertisers

example of star schema

Sales(storeID, itemID, custID, qty, price) ← fact table
Store(StoreID, city, state) ← dimension table
Item(itemID, category, brand, color, size) ← dimension table
Customer(custID,name,address) ← dimension table

OLAP query over a star schema

Join → filter → group → aggregate

Worried about performance?

inherently very slow

special indexes, querey processing techniques

extensive use of materialized views ← useful when lots of queries, not many updates

Data cube (a.k.a. multidimensional OLAP) ← different way to look at data

Dimension data forms axes of "cube" ← any number of dimension is ok

Fact (dependent) data in cells

Aggregated data on sides, edges, corner ← ~ aggregation of col in excell

example (continued):
3D axis of cube = (items, customers, stores)
@ (item72, customer4, store17), we have qty and price ← free floating cell
@ cell at face of cube = aggregate over all items
@ corner of cube (0,0,0), we have full aggregation
@ edge of cube (customer4, store17), 2 by 2 aggregates

If we have 2 stores, 5 items, and 10 customers, number of potential entries =
2 * 5 * 10 ← number of cells in graph
+ 2* 5 + 5 * 10 + 2 * 10 ← 2 by 2 aggregates
+ 17 ← 1 per object
+ 1 ← total aggregate

fact table uniqueness for data cube

sales (storeID, itemID, custID, qty, price, date?)
* Date

Drill-down and roll-up

```
select state, brand, sum(qty*price)
from sales F, store S, item I
where F.storeID = S.storeID and F.itemID = I.itemID
group by state, brand
```
← broken down by state and brand

(1) drill-down: get high level data and drill down ← add a group-by attribute
(2) roll-up: ← remove a group-by attributes

(1) For drill down, add another group by (ex: category)

```
select state, brand, category, sum(qty*price)
from sales F, store S, item I
where F.storeID = S.storeID and F.itemID = I.itemID
group by state, brand,category
```
← add category

← add category

(2) Roll-up is exactly the opposite = remove one

```
select state, brand, sum(qty*price)                    ← remove state
from sales F, store S, item I
where F.storeID = S.storeID and F.itemID = I.itemID
group by state, brand                                  ← remove state
```

SQL constructs

'With cube' and 'with rollup' added to group by clause

```
select dimension-attrs, aggregates
from tables
where conditions
group by dimension-attrs with cube

….
group by dimension-attrs with rollup
                                   ← for hierarchical dimensions, portion of 'with cube'
                                   ← can be used to simulate 'with cube'
```

OLAP review

* star schemas
* data cubes
* 'with cube' and 'with rollup'
* Special indexes and query processing techniques

INTRODUCTION TO DATABASE
- ONLINE ANALYTICAL PROCESSING (OLAP)
- DEMO

Use mysql

```
Sales(storeID, itemID, custID, qty, price)     ← fact table
Store(StoreID, city, state)                    ← dimension table
Item(itemID, category, brand, color, size)     ← dimension table
Customer(custID,name,address)                  ← dimension table
```

star-join

```
select *
from Sales F, Store S, Item I, Customer C
where F.storeID = S.storeID and F.itemID = I.itemID and F.custID = C.custID;
```

selection and projection on star-join

```
select S.city, I.color, C.cName, F.price
from Sales F, Store S, Item I, Customer C
where F.storeID = S.storeID and F.itemID = I.itemID and F.custID = C.custID
and S.state = 'CA' and I.category = 'Tshirt' and C.age<22 and F.price<25;
```

with grouping and aggregation ← to analyze the data

```
select storeID, custID, sum(price)
from Sales
group by storeID, custID;                    ← dimension attributes
```

… drilling down …

```
select storeID, custID, itemID,  sum(price)
from Sales
group by storeID, custID, itemID;            ← dimension attributes
```

… slicing query … ← slicing cube

```
select storeID, custID, itemID,  sum(price)
from Sales F, Store S                        ← join
where F.storeID = S.storeID and state = 'WA' ← subset is slice of data cube
group by storeID, custID, itemID;
```

… dicing the cube query …

```
select F.storeID, custID, F.itemID,  sum(price)
from Sales F, Store S, Item I                ← join
where F.storeID = S.storeID and state = 'WA' ← subset is slice of data cube
        and F.itemID = I.itemID and color = 'red'
group by F.storeID, custID, F.itemID;
                                             ← constraint table to 'red'
```

!!! constraining in 2D =dicing !!!

```
select itemID, sum(price)                    ← much more summarized data
from sales F
group by itemID;


select state, category,  sum(price)
from Sales F, Store S, Item I                ← join
where F.storeID = S.storeID and F.itemID = I.itemID
group by state, category;
```

…. drill down by county ...

```
select state, county, category,  sum(price)
from Sales F, Store S, Item I
where F.storeID = S.storeID and F.itemID = I.itemID
group by state, county, category;
```

… drill down by gender …

```
select state, county, category, gender, sum(price)
from Sales F, Store S, Item I
where F.storeID = S.storeID and F.itemID = I.itemID
group by state, county, category, gender;
```

… roll-up to state-gender …

```
select state, gender, sum(price)
from Sales F, Store S, Item I
where F.storeID = S.storeID and F.itemID = I.itemID
group by state,  gender;
```

with cube

```
select storeID, itemID, custID, sum(price)
from Sales
group by storeID, itemID, custID with cube;                    ← not supported in mysql
```

!!! 2x2 aggregates = 3rd column = null !!! ← null means ‘at any’
← null = edge of cube

!!! 3 null values = total revenues for all stores for all item for allcustomers

```
select sum(price)
from Sales ;
```

Queries on cubes ← preaggegated data

```
select C.*                                  ← which tuples are being used for aggregation
from Cube C, Store S, Item I
where C.storeID = S.storeID and C.itemID = I.itemID
and state = ‘CA’ and color = ‘blue’ and custID is null;


select sum(p)
from Cube C, Store S, Item I
where C.storeID = S.storeID and C.itemID = I.itemID
and state = ‘CA’ and color = ‘blue’ and custID is null;
```

… same result with less efficient processing ...

```
select sum(p)
from Cube C, Store S, Item I
where C.storeID = S.storeID and C.itemID = I.itemID
```

and state = ‘CA’ and color = ‘blue’ and custID is not null; ← give same result, but less efficient

… even less efficient processing …

select sum(price)
from Sales F, Store S, Item I
where C.storeID = S.storeID and C.itemID = I.itemID
and state = ‘CA’ and color = ‘blue’; ← give same result, but less efficient

variation on ‘with cube’

select storeID, itemID, custID, sum(price)
from Sales F
group by storeID, itemID, custID with cube(storeID, custID);
← subset of cube query, with no null in item column

rollup

select storeID, itemID, custID, sum(price)
from Sales F
group by storeID, itemID, custID with rollup;

!!! based on order of atribute in group-by directive !!!

All null
2 right most are null
or 3 attribute is null
← good for hierarchy

!!! When null attribute = summary or sum of all !!!

select state, county, city, sum(price)
from Sales F, Store S
where F.storeID = S.storeID
group by state, country, city with rollup; ← notice the hierarchy here

!!! You won’t see the tuple (CA,null, palo alto), but you will see (CA, Santa Clara, null) !!!
!!! You won’t see the tuple (null, null, palo alto) either, but you will (null,null,null) !!!

!!! rollup is a subset of the cube !!!

INTRODUCTION TO DATABASE
- NOSQL SYSTEMS
- MOTIVATION

NoSQL : the name
SQL = traditional relational DBMS
* Recognition over past decade or so:

Not every data management/analysis problem is best solved EXCLUSIVELY using a traditional relational DBMS

* “NoSQL” = “No SQL” = not using traditional relational DBMS
* “No SQL” <> Don’t use SQL language
* “No SQL” = “Not only SQL”

* convenient
  - Simple data model
  - Declarative query language
  - Transaction guarantees ←
* Multi-user
  - Transaction guarantees
* Safe
* Persistent
  - Files are OK
* Reliable
  - Redo ok
* Massive
  - Much much more massive that DBMS
* Efficient
  - Efficiency requirement extremely high

NoSQL: compromise earlier ones to boost latter ones

Alternative to traditional relational DBMS

+ Flexible schema
+ Quicker/Cheaper to set up
+ Massive scalability
+ RElax consistency → higher performance and availability

- No declarative language → more programming
- Relaxed consistency → fewer guarantees

Example:
Web log analysis
Each record: UserID, URL, timestamp, additional-info

RDBMS Task: Load into database system
- data cleaning
- data extraction
- verification (all URLs are valid)
- schema specification

NoSQL Task:
- Nothing!

Task: Find al records for …
- … a given UserID
- … a given URL
- … given timestamp
- … certain construct appearing in additional-info

Task: Find all pairs of UserIDs accessing same URL

Separate records: UserID, name, age, gender, …

Task: Find average age of user accessing given URL
← SQL-like query
!!! consistency for average age is not that important ⇒ NoSQL still ok !!!
← exact snapshot of database not required as changing all the time anyway
Example 2:
Social-network graph
EAch record: UserID1, UserID2
Separate records: UserID, name, gencer, age, …

Task: Find all friends of a given user
Task: Find all friends of friends of a given user
Task: Find all women friends of men friends of given user
Task: Find all friends of friends of friends of … friends of given user

Example 3:
Wikipedia pages
Large collection of documents
Combination of structured and unstructured data

Task : retrieve introductory paragraph of all pages about US president before 1980

INTRODUCTION TO DATABASE
- NOSQL SYSTEMS
- PRODUCT OVERVIEW AS OF NOV 2011

Several incarnations
* MapReduce framework ← Good for OLAP type of transaction
* Key-value stores ← Good for OLTP type of transaction
* Document stores
* Graph database systems

MapReduce Framework
No data model, data stored in files
GFS
HDFS
User provides specific functions
map()
reduce()
reader()
writer()
combiner()
System provides data processing “glue”, fault tolerance

Map and Reduce functions
Map: Divide problem into subproblems
map(item) ⇒ 0 or more <key,value> pairs
Reduce: Do work on subproblems, combine results
reduce(Key, list-of-values) ⇒ 0 or more records

input records → [MAP] → K1 → [REDUCE] ⇒ output records
\--> K2 → [REDUCE] ⇒ output records
\-----> K3 ….

!!! input records are read from files by the reader function !!!

!!! output records are written in files by the writer function !!!

!!! Combiner ~ pre-reduce phase !!!

!!! A lot of stuff can go on in parallel !!! ← fast processing

Task: Count number of accessses for each domain (inside URL)

map(record) → <DOMAIN, null> ← upper case = key
reduce(DOMAIN, list of null) → <DOMAIN, count>
that's all !

combiner(domain, list-of-null) → <domain, count>
reduce(domain, list-of-counts) → <domain, sum-of-counts>

Task: Total "value" of accesses for each domain based on additional info

map(record) → <domain, score> ← domain extracted fro mURL
← score based on info in additional info
← score based on what it sees
reduce(domain,list-of-scores) → <domain, sum-of-scores>

Separate records: UserID, name, age, gender
Task: Total "value" of accesses for ….

Map reduce framework Summary
* No data model, data stored in files
* user provides specific functions
* system provides data processing "glue", fault-tolerance, scalability

!!! Schemas and declarative queries are missed !!!

Hive - schemas, SQL-like query language ← 2 higher programming languages

Pig - more imperative but with relational operators
Both compile to “workflow” of hadoop (MapReduce) jobs

Dryad allows user to specify workflow
* Also DryadLINQ language ← seat on top of Dryad

Key-value stores ← OLTP on massive database

Extremely simple interface
* Data model(key, value) pairs
* Operations: Insert(key,value), fetch (key), update(key), delete(key)

*Some allow (non-uniform) columns within value
* Some allow FEtch on range of keys ← 2< key < 10

Implementation: efficiency, scalability, fault-tolerance (if a node fails)
* Records distributed to nodes based on key
* Replication
* Single-record transactions, “eventual consistency”

Example systems:
* Google BigTable
* Amazon Dynamo
* …

Document stores
Like Key-value stores except value is document
* data model (key, document) pairs
Document: JSON, XML, other semistructured formats
Basic operations: Insert(key,document), fetch(key),Update(key),delete(key)
…..

Graph database systems
* Data model: nodes and edges
* Nodes may have properties (including ID)
* Edges may have labels or roles

ID: 1
Name: Amy
Grade: 9

ID:2
Name:Ben

GRade: 9

ID:3
Name:Carol
Grade:10

Edges:
Amy Likes Ben, Ben likes Carol
Amy friends with Carol

…
Implementation
* Neo4j
* FlockDB
* Pregel
...