CS5200 Database Management Integrity and Security

Ken Baclawski
Spring 2017

Outline

- Solution to Assignment #3
- Integrity
  - Checks and Assertions
  - Triggers
- Security
- Content Management
- Translating to the Relational Algebra
- Index Design
- Assignment #5
- Review for Mid-Term Exam

Solution to Assignment #3

Integrity

Check Constraints

- Clause in a create table statement
- Constrains the values of one or more fields
- Examples:

```
check (weight >= 0)
check (gender in ('Male', 'Female'))
check (exists(
        select * from Bond b
         where b.partOf = id))
```

Constraints for Single Table Strategy

- Single table strategy for inheritance requires constraints
- For the chemistry database
  - If a bond is ionic then
    - charge is not null
    - polarity is null
  - If a bond is covalent then
    - polarity is not null
    - charge is null

- SQL only supports AND, OR and NOT.
- Is it possible to express these constraints using AND, OR and NOT?
- Yes, it is possible!

Logical Implication

- The implies operator is expressible in logic like this:

  A implies B   is equivalent to   (not A) or B

- For example,

  If a bond is ionic then charge is not null

  is equivalent to

```
(bondType != 'Ionic')
or (charge is not null)
```

Assertion

- Declared outside any table
- Example:

```
create assertion MoleculeHasBonds
  check (exists(
         select * from Bond b, Molecule m
         where b.partOf = m.id))
```

- In theory, any assertion could be specified as a check constraint and vice versa

Database Support for Checks and Assertions

- In reality, database systems do not support assertions and have limited support for check constraints
- What can one do?
  - Enforce using a programming language
  - Use triggers
- Triggers are more reliable but have disadvantages
  - No standard syntax or semantics
  - Very complex language
  - Limited support for testing and debugging
  - Triggers can interact with one another

Example Database

```
create table Student(
  id int primary key auto_increment,
  name varchar(255),
  major varchar(255)
);
create table Course(
  id int primary key auto_increment,
  code varchar(255) not null
);
create table Section(
  id int primary key auto_increment,
  number int not null,
  partOf int not null references Course(id)
    on update cascade on delete cascade
);
create table Registration(
  registers int references Student(id)
    on update cascade on delete cascade,
  registeredIn int references Section(id)
    on update cascade on delete cascade,
  primary key(registers, registeredIn)
);
```

Enforcement Strategies

- Singular Table
- Error Table
- Stored Procedure Language
  - Raise exception
  - Rollback

Singular Table

- Table with exactly one row
  - It is essential that there never be another row in the table
- Example:

```
create table Singular (
  id int primary key
);
insert into Singular(id) values (1);
```

Enforcing Not Null Constraint

- Start with a simple case: `Student.name` is not null

```
create trigger NameNotNullInsert before insert on Student
for each row
  insert into Singular(id)
  select *
    from Singular si
   where new.name is null;
```

- This will be executed ("triggered") whenever the insert command is executed on the `Student` table
  - Each row that is inserted will execute the insert command into the `Singular` table
  - If the new record has null name, then the triggered insert command will fail and so the insert into the `Student` table will also fail.

Enforcing Not Null Constraint

- It is also necessary to check updates

```
create trigger NameNotNullInsert before update on Student
for each row
  insert into Singular(id)
  select *
    from Singular si
   where new.name is null;
```

- It is not necessary to check deletes

Triggering Events

- Each table has 6 events that can trigger a command:
  - before or after
  - insert, update or delete
- Each event can have only one trigger
- The old record is indicated with `old` and the new record with `new`
  - The names for these records vary with database
  - The syntax for triggers is not standard

Converting check constraint to trigger

- Define triggers for insert and update that negate the check constraint
- Example in the `Section` table: `check (number > 0)`

```
create trigger NumberPositiveInsert before insert on Section
for each row
  insert into Singular(id)
  select *
    from Singular si
   where new.number <= 0;

create trigger NumberPositiveUpdate before update on Section
for each row
  insert into Singular(id)
  select *
    from Singular si
   where new.number <= 0;
```

Converting assertion to trigger

- This is more complex
  - One must determine which of the tables are being constrained
  - One must define insert, update and delete triggers
- Example assertion: A student may not be enrolled in two different sections of the same course

```
create assertion NoRedundantSectionEnrollment
  check not exists (
          select *
            from Student s, Registration r,
                 Section se, Course c
           where s.id = r.registers
             and se.id = r.registeredIn
             and c.id = se.partOf
           group by s.id, c.id
          having count(*) > 1
        );
```

Converting assertion to trigger

- Simplify as much as possible

```
create assertion NoRedundantSectionEnrollment
  check not exists (
          select *
            from Registration r, Section se
           where se.id = r.registeredIn
           group by r.registers, se.partOf
          having count(*) > 1
        );
```

Converting assertion to trigger

- Determine the table being constrained
  - In this case, the registrations are being constrained

```
create assertion NoRedundantSectionEnrollment
  check not exists (
          select *
            from Registration r, Section se
           where se.id = r.registeredIn
           group by r.registers, se.partOf
          having count(*) > 1
        );
```

Converting assertion to trigger

- Negate and define the triggers
  - This is the insert trigger

```
create trigger NoRedundantSectionEnrollmentInsert
after insert on Registration
for each row
  insert into Singular(id)
  select *
    from Singular si
   where exists (
          select *
            from Registration r, Section se
           where se.id = r.registeredIn
           group by r.registers, se.partOf
          having count(*) > 1
        );
```

Error Table Strategy

- Instead of a singular table, use an error table like this:

```
create table Error(
  id int primary key auto_increment,
  message varchar(1023)
);
```

Error Table Strategy

- Instead of a singular table, use an error table like this:

```
create trigger WarnRedundantSectionEnrollmentInsert
  after insert on Registration
for each row
  insert into Error(message)
  select concat('A student ', s.name,
      ' has registered in two sections of ', c.code)
    from Student s, Course c
   where exists (
           select *
             from Section se
            where s.id = new.registers
              and se.id = new.registeredIn
              and c.id = se.partOf
            having count(*) > 1
         );
```

Error Table Strategy

- Advantages
  - More information about the error
  - Does not fail immediately
- Disadvantages
  - Does not fail immediately
  - Someone must check the `Error` table

Stored Procedure Exception Strategy

- This uses the MySQL stored procedure language

```
create trigger WarnRedundantSectionEnrollmentInsert
  after insert on Registration
for each row
  declare RedundantSectionEnrollment exception;
  begin
    if (exists (
          select *
            from Section se, Registration r
           where s.id = r.registers
             and se.id = r.registeredIn
             and c.id = se.partOf
           group by s.id, c.id
           having count(*) > 1
        )
    then raise RedundantSectionEnrollment;
    end if;
  end;
```

Stored Procedure Exception Strategy

- Advantages
  - Fails immediately
  - Error is explained
- Disadvantages
  - Highly non-standard and non-portable
  - Few databases support user-defined exceptions

Stored Procedure Rollback Strategy

- This uses the MySQL stored procedure language

```
create trigger WarnRedundantSectionEnrollmentInsert
  after insert on Registration
for each row
  begin
    if (exists (
          select *
            from Section se, Registration r
           where se.id = r.registeredIn
           group by r.registers, se.id
           having count(*) > 1
        )
    then rollback;
    end if;
  end;
```

Stored Procedure Rollback Strategy

- Advantages
  - Fails immediately
  - Most databases will support this
- Disadvantages
  - No explanation of error
  - Highly non-standard and non-portable

Conclusions

- Very clumsy way to enforce integrity constraints compared to not null, check and assertion constraints
- Unfortunately, no major DBMS supports all of them
  - Most support not null constraints
  - Most support check constraints that do not have subqueries
- The only reliable technique for constraint enforcement is with triggers

Assertion Problem 1

```
create table Company(
  id int primary key,
  name varchar(500) not null,
  product varchar(500)
);
create table Person(
  id int primary key,
  name varchar(200) not null,
  worksFor int,
  foreign key (worksFor) references Company(id)
    on update cascade on delete no action
);
```

It is required that there is a company named IBM in the database.

Assertion Solution

It is required that there is a company named IBM in the database.

```
create assertion IBMisCompany check
  exists (
    select *
      from Company c
     where c.name = 'IBM'
  );
```

Assertion Problem 2

```
create table Company(
  id int primary key,
  name varchar(500) not null,
  product varchar(500)
);
create table Person(
  id int primary key,
  name varchar(200) not null,
  worksFor int,
  foreign key (worksFor) references Company(id)
    on update cascade on delete no action
);
```

Require that every company has at least one person who works for the company.

Assertion Solution Part 1

Require that every company has at least one person who works for the company.

First rewrite: For every company c there exists a person p such that p works for c

Second rewrite: There does not exist a company c such that there does not exist a person p such that p works for c

Assertion Solution Part 2

There does not exist a company c such that there does not exist a person p such that p works for c

```
create assertion CompanyHasAWorker check
  not exists (
    select *
      from Company c
     where not exists (
              select *
                from Person p
               where p.worksFor = c.id
            )
  );
```

Check Constraint Solution

For every company there exists a person p such that p works for the company

```
alter table Company
  add constraint CompanyHasAWorker check(
    exists(
      select *
        from Person p
       where p.worksFor = id
    )
  );
```

Assertion Problem 3

```
create table Company(
  id int primary key,
  name varchar(500) not null,
  product varchar(500)
);
create table Person(
  id int primary key,
  name varchar(200) not null,
  worksFor int,
  foreign key (worksFor) references Company(id)
    on update cascade on delete no action
);
```

It is required that if a company has an employee, then the company has a product.

Write your answer on paper or on your laptop.

Security

Granting Privileges

- Similar to operating system privileges
  - The creator (owner) of a file specifies who has access and update privileges on the file
  - The creator of a table specifies who has privileges on the table
- Possible privileges on tables
  - select for reading the table
  - update for updating table records
  - delete for deleting records of the table
  - Many other kinds of privilege

Granting Privileges

- Syntax is:

  `grant` privileges `on` table `to` users or roles

- *Roles* are groups of users
- Some databases allow one to specify the columns of the table that are being authorized
- One can specify views as well as tables
  - This allows very fine-grained access rights
- One can add “`with grant option`” to delegate the granted authority

Security Problem 1

```
create table Company(
  id int primary key,
  name varchar(500) not null,
  product varchar(500)
);
create table Person(
  id int primary key,
  name varchar(200) not null,
  worksFor int,
  foreign key (worksFor) references Company(id)
    on update cascade on delete no action
);
```

Give 'alice' permission to access all persons who work for IBM.

Security Problem 1 Solution

Give 'alice' permission to access all persons who work for IBM.

```
create view IBMEmployees as
  select p.id, p.name
    from Person p, Company c
   where p.worksFor = c.id
     and c.name = 'IBM';
grant select on IBMEmployees to 'alice';
```

Security Problem 2

```
create table Company(
  id int primary key,
  name varchar(500) not null,
  product varchar(500)
);
create table Person(
  id int primary key,
  name varchar(200) not null,
  worksFor int,
  foreign key (worksFor) references Company(id)
    on update cascade on delete no action
);
```

Give 'alice' permission to change the employees who work for IBM.

Security Problem 2 Analysis

Give 'alice' permission to change the employees who work for IBM.

The company is specified by its name, but the company id is necessary for changing the employees. So one must have read permission to the id and name fields of the Company record for IBM. No attribute of Person is specified, so the id is used for identifying a person. One must have read and update permission to the id and worksFor fields of Person.

Security Problem 2 Solution

Give 'alice' permission to change the employees who work for IBM.

```
create view IBM as
  select c.id, c.name
    from Company c
   where c.name = 'IBM';
grant select on IBM to 'alice';

create view PersonIdWorksFor as
  select p.id, p.worksFor from Person p;
grant select, update
  on PersonIdWorksFor to 'alice';
```

Landform Security Problem

```
create table Landform (
  id int primary key,
  name varchar(200) not null
);
create table Hill (
  id int primary key,
  foreign key(id) references Landform(id)
    on update cascade on delete cascade,
  height double not null,
  summitArea double,
  partOf int,
  foreign key(partOf) references Hill(id)
    on update cascade on delete cascade
);
create table Valley (
  id int primary key,
  foreign key(id) references Landform(id)
    on update cascade on delete cascade,
  area double not null
);
```

Landform Security Problem

```
create table Border (
  borders int not null,
  foreign key(borders) references Valley(id)
    on update cascade on delete cascade,
  isBorderedBy int not null,
  foreign key(isBorderedBy) references Hill(id)
    on update cascade on delete cascade,
  primary key(borders, isBorderedBy)
);
```

Give the 'geographer' role permission to update the borders of a valley and to delegate this permission.

Write your answer on paper or on your laptop.

Content Management

Definition

- A content management system (CMS) is a system used to organize and facilitate collaborative content creation
- One can regard a CMS as being a software layer above the database system
- There are around 100 open source CMSs, and at least as many proprietary CMSs
- This is a very active area, and one can expect considerable growth in the future

Examples

- One CMS among hundreds is WordPress
  - Estimated to be used on over 25% of all websites
  - Mainly used for blogging sites
- Another CMS is MediaWiki
  - The CMS used by Wikipedia
  - Over 25,000 public sites currently being tracked
  - MediaWiki software is downloaded over 50,000 times every month, mainly to China
  - Even WordPress uses MediaWiki
- Piazza is a kind of content management system

Standards

- The SQL standard played a significant role in the success of the relational database industry
- However, SQL is only an API, not a protocol
  - Databases cannot easily interoperate
- There is a standard for CMS which is built on SQL and includes a protocol
  - The standard is CMIS
  - Latest version is at

http://docs.oasis-open.org/cmis/CMIS/v1.1/CMIS-v1.1.html

CMIS Standard

The Content Management Interoperability Services (CMIS) standard defines a domain model and Web Services, Restful AtomPub and browser (JSON) bindings that can be used by applications to work with one or more Content Management repositories/systems.

The CMIS interface is designed to be layered on top of existing Content Management systems and their existing programmatic interfaces. It is not intended to prescribe how specific features should be implemented within those CM systems, nor to exhaustively expose all of the CM system’s capabilities through the CMIS interfaces. Rather, it is intended to define a generic/universal set of capabilities provided by a CM system and a set of services for working with those capabilities.

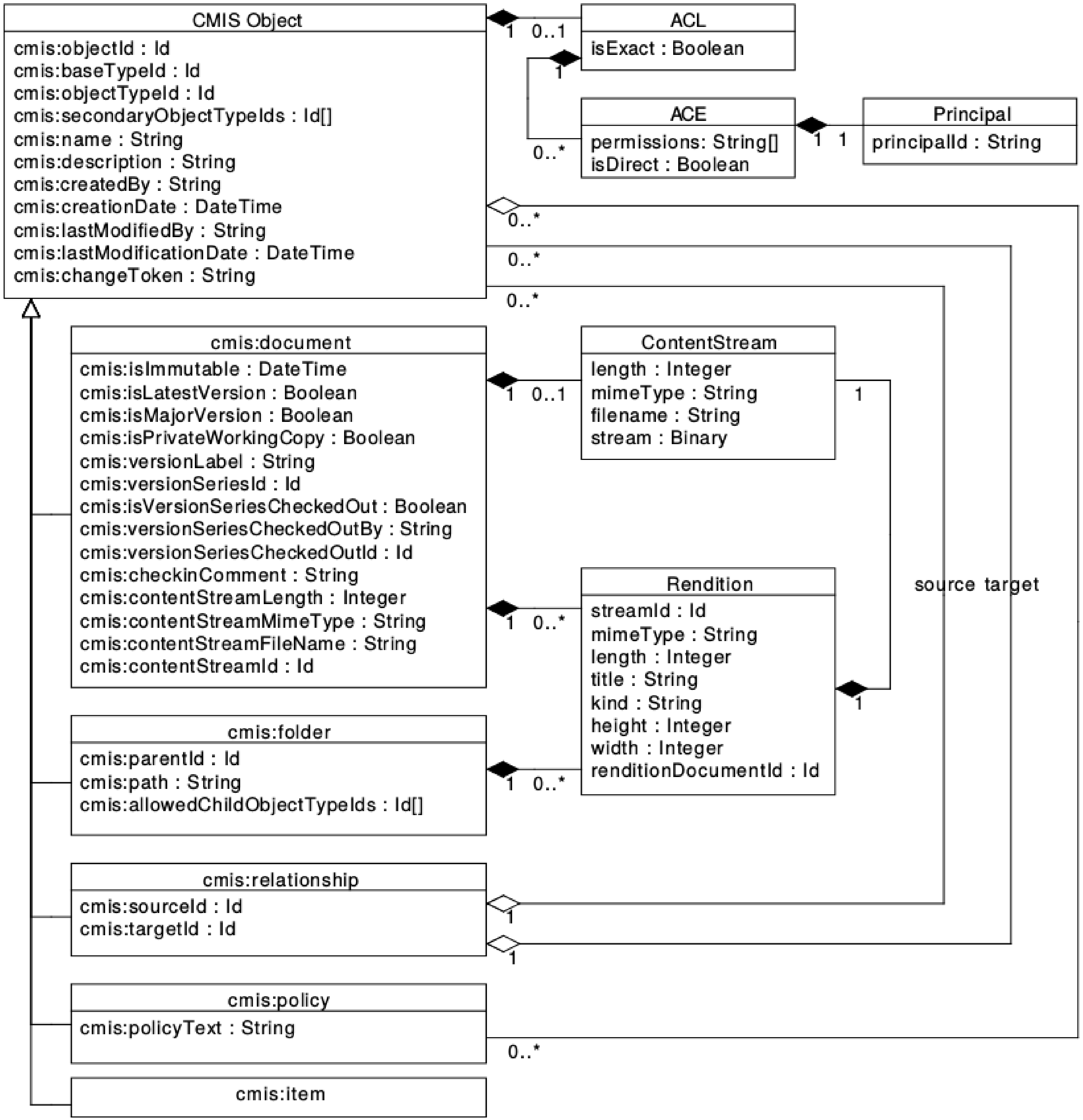
CMIS Services

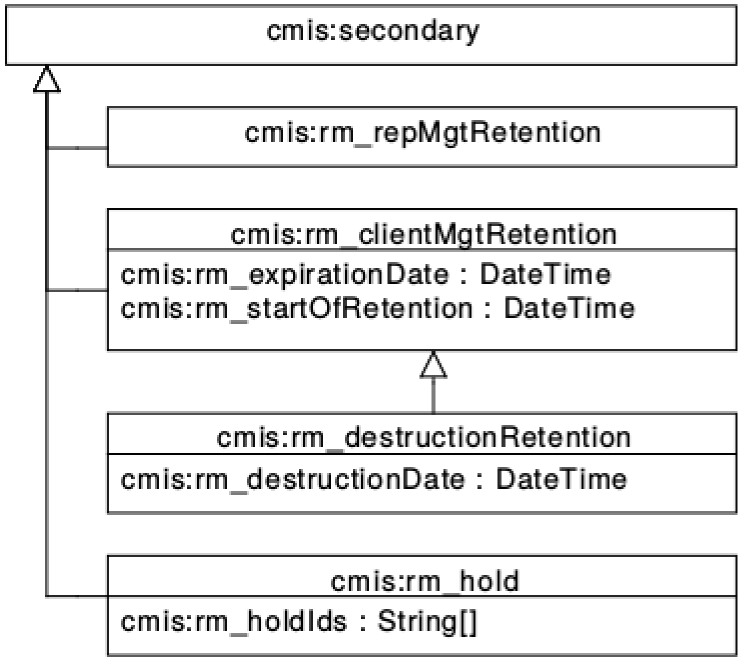
- CMIS is an interface to a repository
  - May access multiple repositories
  - May federate multiple databases
- Core model
  - Persistent information entities
  - Basic services for access and manipulation of entities

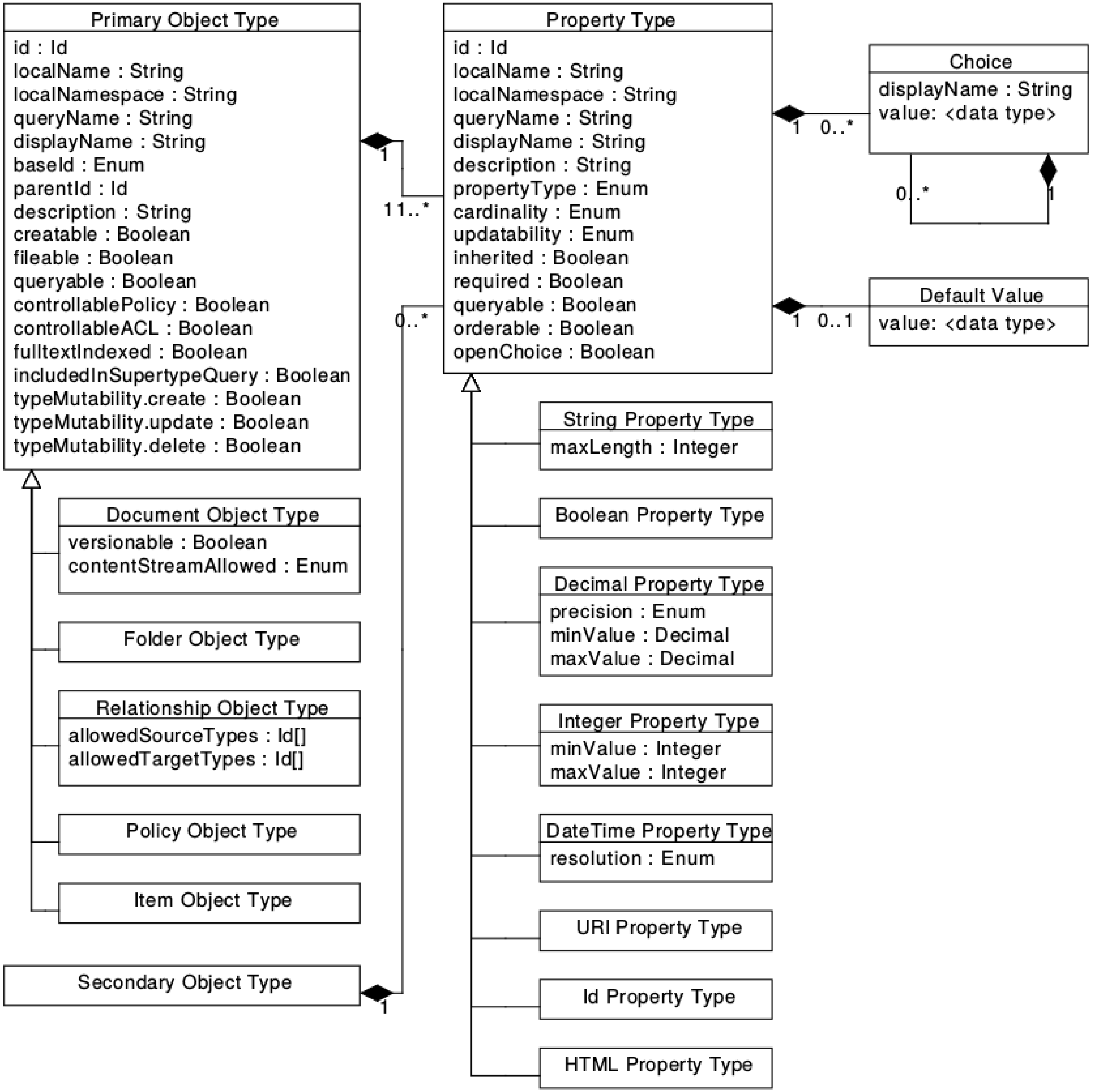
CMIS Limitations

CMIS does not include

- Transient entities
- User profiles
- Compound documents
- Virtual documents
- Work flows and business processes
- Events
- Subscriptions
- Inter-repository operations


CMIS Object
cmis:objectId : Id
cmis:baseTypeId : Id
cmis:objectTypeId : Id
cmis:secondaryObjectTypeIds : Id[]
cmis:name : String
cmis:description : String
cmis:createdBy : String
cmis:creationDate : DateTime
cmis:lastModifiedBy : String
cmis:lastModificationDate : DateTime
cmis:changeToken : String
ACL
isExact : Boolean
ACE
permissions: String[]
isDirect : Boolean
Principal
principalId : String
1
0..1
1
0..*
1
1
0..*
0..*
0..*
cmis:document
cmis:isImmutable : DateTime
cmis:isLatestVersion : Boolean
cmis:isMajorVersion : Boolean
cmis:isPrivateWorkingCopy : Boolean
cmis:versionLabel : String
cmis:versionSeriesId : Id
cmis:isVersionSeriesCheckedOut : Boolean
cmis:versionSeriesCheckedOutBy : String
cmis:versionSeriesCheckedOutId : Id
cmis:checkinComment : String
cmis:contentStreamLength : Integer
cmis:contentStreamMimeType : String
cmis:contentStreamFileName : String
cmis:contentStreamId : Id
ContentStream
length : Integer
mimeType : String
filename : String
stream : Binary
1
0..1
1
Rendition
streamId : Id
mimeType : String
length : Integer
title : String
kind : String
height : Integer
width : Integer
renditionDocumentId : Id
1
0..*
1
source
target
cmis:folder
cmis:parentId : Id
cmis:path : String
cmis:allowedChildObjectTypeIds : Id[]
1
0..*
cmis:relationship
cmis:sourceId : Id
cmis:targetId : Id
1
1
cmis:policy
cmis:policyText : String
0..*
cmis:item


cmis:secondary
cmis:rm_repMgtRetention
cmis:rm_clientMgtRetention
cmis:rm_expirationDate : DateTime
cmis:rm_startOfRetention : DateTime
cmis:rm_destructionRetention
cmis:rm_destructionDate : DateTime
cmis:rm_hold
cmis:rm_holdIds : String[]


Primary Object Type
id : Id
localName : String
localNamespace : String
queryName : String
displayName : String
baseId : Enum
parentId : Id
description : String
creatable : Boolean
fileable : Boolean
queryable : Boolean
controllablePolicy : Boolean
controllableACL : Boolean
fulltextIndexed : Boolean
includedInSupertypeQuery : Boolean
typeMutability.create : Boolean
typeMutability.update : Boolean
typeMutability.delete : Boolean
Document Object Type
versionable : Boolean
contentStreamAllowed : Enum
Folder Object Type
Relationship Object Type
allowedSourceTypes : Id[]
allowedTargetTypes : Id[]
Policy Object Type
Item Object Type
Secondary Object Type
1
1 1..*
0..*
1
Property Type
id : Id
localName : String
localNamespace : String
queryName : String
displayName : String
description : String
propertyType : Enum
cardinality : Enum
updatability : Enum
inherited : Boolean
required : Boolean
queryable : Boolean
orderable : Boolean
openChoice : Boolean
String Property Type
maxLength : Integer
Boolean Property Type
Decimal Property Type
precision : Enum
minValue : Decimal
maxValue : Decimal
Integer Property Type
minValue : Integer
maxValue : Integer
DateTime Property Type
resolution : Enum
URI Property Type
Id Property Type
HTML Property Type
Choice
displayName : String
value: <data type>
1 0..*
0..*
1
Default Value
value: <data type>
1 0..1

Translating to the Relational Algebra

Translate Query 6 to relational algebra

- List all molecules consumed by a reaction named WFO

```
select m.id
  from Molecule m, Reaction r,
       Consumption c
 where m.id = c.consumes
   and c.isConsumedBy = r.id
   and r.name = 'WFO'
```

- The query performs these steps:
  - Join 3 tables
  - Select the rows of the join such that `r.name` is `'WFO'`
  - Return the `m.id` column

Translate Query 6 to relational algebra

- List all molecules consumed by a reaction named WFO
- The relational algebra expression is

  $\pi_{\{m.id\}}\sigma_{P}((\text{Reaction} \bowtie_{K} \text{Consumption}) \bowtie_{J} \text{Molecule})$

  where

  P is the constraint “r.name = 'WPO'”

  J is the join condition c.consumes = m.id

  K is the join condition r.id = c.consumedBy

Rearranging a relational algebra expression

- The RA expression can be rearranged to improve performance as follows:

$$\pi_{\{m.id\}}\sigma_P((\text{Reaction} \bowtie_K \text{Consumption}) \bowtie_J \text{Molecule})$$

$$\pi_{\{m.id\}}(\sigma_P(\text{Reaction} \bowtie_K \text{Consumption}) \bowtie_J \text{Molecule})$$

$$\pi_{\{m.id\}}((\sigma_P\text{Reaction} \bowtie_K \text{Consumption}) \bowtie_J \text{Molecule})$$

Advantages of the rearrangement

- The rearranged expression has better performance
- The original expression computed the entire join of all three tables
  - The resulting join has as many records as the `Consumption` table
- The new expression joins one record of the `Reaction` table with the `Consumption` table and then joins these with the `Molecule` table.

Translating a query to relational algebra

- Show the code and number of bonds for every molecule, but if a molecule has no bonds, then show the code and "unbonded"
- Here is the SQL query

```
select m.code,
       if(count(b.id) = 0,
          'unbonded',count(b.id))
  from Molecule m left join Bond b
    on (m.id = b.partOf)
 group by m.id
```

Translating a query to relational algebra

- Show the code and number of bonds for every molecule, but if a molecule has no bonds, then show the code and "unbonded"
- The relational algebra expression is

$$\pi_{\{m.code,if\}}(\xi_{if}(\xi_{count(b.id)}(\psi_C(\text{Molecule} ⟗_J \text{Bond}))))$$

where

if is the if expression in the select clause

C is the grouping criterion "group by m.id"

J is the join condition m.id = b.partOf

Implementing a query

The relational algebra expression is how the query is implemented

$\pi_{\{m.code,if\}}(\xi_{if}(\xi_{count(b.id)}(\psi_C(Molecule \bowtie_J Bond))))$

1. Read the Molecule table from the database

2. For each row of the Molecule table match with rows in the Bond table

3. Hash the results using the value of `m.id`

4. Count the number of values of `b.id` in each hash set

5. Apply the if expression

6. Discard all columns except the `m.code` and if expression columns

Equivalent query trees

- A relational algebra expression is also called a *query tree*
  - The operations are the nodes
  - The child nodes are the operands (parameters) of the operations
- The relational algebra satisfies axioms
  - Products and joins are commutative and associative
  - Selections with AND are split into several selections
    - Use conjunctive normal form
  - Selection and join sometimes satisfy distributivity
    - Move selections up and down the query tree

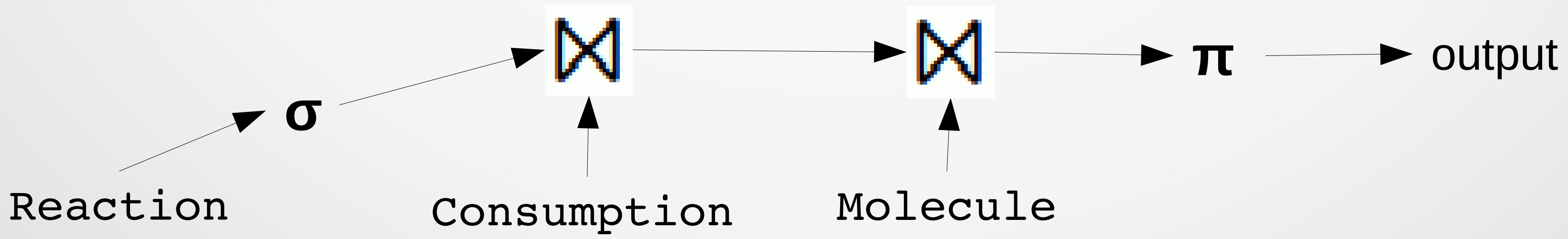
Equivalent query trees

- Other axioms
  - Partition and join sometimes satisfy distributivity
    - Move partitions up and down the query tree
  - Projection and join sometimes satisfy distributivity
    - Move partitions up and down the query tree

Query Processing

- Translate to relational algebra
- Optimize the relational algebra expression
- Select program for each operator
- Pipeline processing

$$\pi_{\{m.id\}}((\sigma_P \text{Reaction} \bowtie_K \text{Consumption}) \bowtie_J \text{Molecule})$$



Optimization

- List candidate query trees
  - Restrict to pipeline (left-deep) query trees
  - Use heuristics to limit the number of possibilities
- For each candidate query tree select an implementation of every operator in the tree
  - This is the Strategy Design Pattern
- Compute the cost (time and memory requirements) of each candidate query tree
  - Depends on storage devices being used
- Use the lowest cost candidate query tree

Selecting Implementations

- Selection
  - Either scan or index
  - Index strategy requires an index
    - Must be on a table
    - Must be selective
- Projection
  - Only keep columns that will later be used

Selecting Implementations

- Join
  - Nested loop join (including product)
    - Always performed by blocks
  - Index loop join
    - Requires index on one of the columns
    - Must be selective
  - Hash join
    - Takes advantage of sequential writes
  - Merge join
    - Same as hash join but also sorts
    - Can also take advantage of sequential writes

Index Design

Index Design

- Index Types
  - Hash table
  - B-tree
- Required indexes
  - Primary key
  - Uniqueness constraints
  - Target of foreign key constraint
- Optional indexes
  - Selection
  - Join
- Index Design
  - Both required and optional indexes
  - Specify type for each index
  - Order of columns for B-tree indexes

Hash versus B-tree

- Hash index for exact match only
- Hash index is faster
- Use hash index if range query is not needed

- B-tree index supports both exact match and range queries
- B-tree index is slower

Index Design

- Given a set of queries and commands find the candidate indexes
- Only one design for the set, not one design per query
- Must include all required indexes
- Must specify type of every index
- If B-tree has multiple columns, must specify order of columns in the index

Index Design

- For this query

P is the constraint “r.name = 'WPO'”

J is the join condition c.consumes = m.id

K is the join condition r.id = c.consumedBy

$$\pi_{\{m.id\}}((\sigma_P \text{Reaction} \bowtie_K \text{Consumption}) \bowtie_J \text{Molecule})$$

$$\begin{tikzcd}
 & & \bowtie \arrow[r] & \bowtie \arrow[r] & \pi \arrow[r] & \text{output} \\
 & \sigma \arrow[ur] & & & & \\
\text{Reaction} \arrow[ur] & & \text{Consumption} \arrow[uu] & \text{Molecule} \arrow[uu] & &
\end{tikzcd}$$

Index Design

- For this query

  P is the constraint "`r.name = 'WPO'`"

  Hash index on `Reaction(name)`

  J is the join condition `c.consumes = m.id`

  Hash index on `Molecule(id)`

  K is the join condition `r.id = c.consumedBy`

  Hash index on `Consumption(consumedBy)`

- However, there is a primary key index on `Consumption`

  Better design is B-tree index on `Consumption(consumedBy, consumes)`

Range Queries

- Kinds of range query
  - Comparison constraint: `r.name > 'Mary'`
  - Prefix wildcard: `r.name like 'M%'`
  - Prefix in multicolumn index: `c.consumedBy = r.id`
- Maintaining an index is costly
  - One index will almost always be better than two

Assignment #5

Checking assertion syntax

- Since databases don't support assertions, how can one check that the syntax is correct?
- Simple trick: rewrite the assertion as a query
- For example,

```
create assertion MoleculeHasBonds
  check (exists(
           select * from Bond b, Molecule m
            where b.molecule = m.id)

select * from Reaction r where
  exists(
    select * from Bond b, Molecule m
     where b.molecule = m.id)
```

Use any table here that is not used by the assertion

Review for MidTerm

Data Modeling

- Object versus Value
- Class structure
- Attributes
- Multiplicity
- Attribute Type
- Enumeration
- Specialization
- Association
- Roles
- Association Class
- Aggregation
- Composition
- Datatype
- Stereotype

Relational Model

- Tables
  - Rows
  - Columns
- Column Types
- Meaning of NULL
- Primary Key
- Create table syntax

- Table versus Class
  - No object identity
  - No multi-attributes
  - Only primitive values
  - No direct object references
  - Limited extendability
  - No inheritance

Translating to Relational Model

- Strategies
- Simulating object identity
- Class as a table
- Uniqueness constraints
- Single-valued attributes
- Multivalued attributes
- Foreign keys
- Update and delete behavior
- Associations
  - Many-to-one
  - Many-to-many
- Compositions
- Aggregations
- Cyclic dependencies

Translating to Relational Model

- Subclass hierarchies
  - Joined strategy
  - Table per class
  - Single table
- Selecting a strategy
- Enumerations
  - Column type
  - Separate table
- Design Issues
  - Attribute vs Association
  - Attribute vs Class
  - Reification

Basic Queries

- Navigation
- Basic clauses
  - From
  - Where
  - Group by
  - Having
  - Order by
  - Select
- NULL and logic
- Wildcards
- Aliases
  - Table
  - Column
- Joins
  - Inner
  - Outer
- Join syntax
- Using select distinct
- Aggregation
- Conditional expression

Advanced Queries

- Nested Queries
- Exists operator
  - Exists vs duplicate elimination
- Rewriting requirements
- Simplifying queries
- Universal quantification
- Alternative syntaxes
- Subquery contexts
- Set operations
- Common errors
  - Multivalued attributes
  - Canceling negations

Modifying the Database

- Insert
  - One row
  - Query
- Update
  - Basic
  - Subquery
- Delete
  - Basic
  - Subquery
- Auto increment
- Views