CS5200 Database Management Advanced SQL

Ken Baclawski
Spring 2017

Outline

- Solution to Assignment #2
- Advanced Queries
- Relational Algebra
- Modifying Data
- Views
- Assignment #4

Solution to Assignment #2

Advanced Queries

Query 14

- What are the molecules by code that have a name?
- This looks like Query 3 but for molecules rather than reactions.
- Try this:

```
select m.code
  from Molecule m
 where m.name is not null
```

Query 14

- What are the molecules by code that have a name?
- This looks like Query 3 but for molecules rather than reactions.
- Try this:

```
~~select m.code~~
~~  from Molecule m~~
~~ where m.name is not null~~
```

- This query will not be accepted because Molecule does not have a name field

Query 14

- What are the molecules by code that have a name?
- This looks like Query 3 but for molecules rather than reactions.
- Try this:

```
~~select m.code
  from Molecule m
 where m.name is not null~~
```

- This query will not be accepted because Molecule does not have a name field
- Here is another try:

```
select m.code
  from Molecule m,
       MoleculeName n
 where n.molecule = m.id
   and m.name is not null
```

Query 13

- What are the molecules by code that have a name?
- This looks like Query 3 but for molecules rather than reactions.
- Try this:

```
~~select m.code~~
~~  from Molecule m~~
~~ where m.name is not null~~
```

- This query will not be accepted because Molecule does not have a name field
- Here is another try:

```
~~select m.code~~
~~  from Molecule m,~~
~~       MoleculeName n~~
~~ where n.molecule = m.id~~
~~   and m.name is not null~~
```

  - This also wrong

Query 14: Multivalued Attributes

- What are the molecules by code that have a name?
- The second attempt is also wrong. In fact, `m.name` is never null because it is a column in the primary key
- Multivalued attributes and many-to-many relationships should use a new concept: `exists`

- A molecule m has a name when *there exists* a record in the MoleculeName table whose molecule field is equal to m.id
- This is a correct query:

```
select m.code
  from Molecule m
 where exists(
     select *
       from MoleculeName n
      where n.molecule = m.id
   )
```

Exists Operator and Subqueries

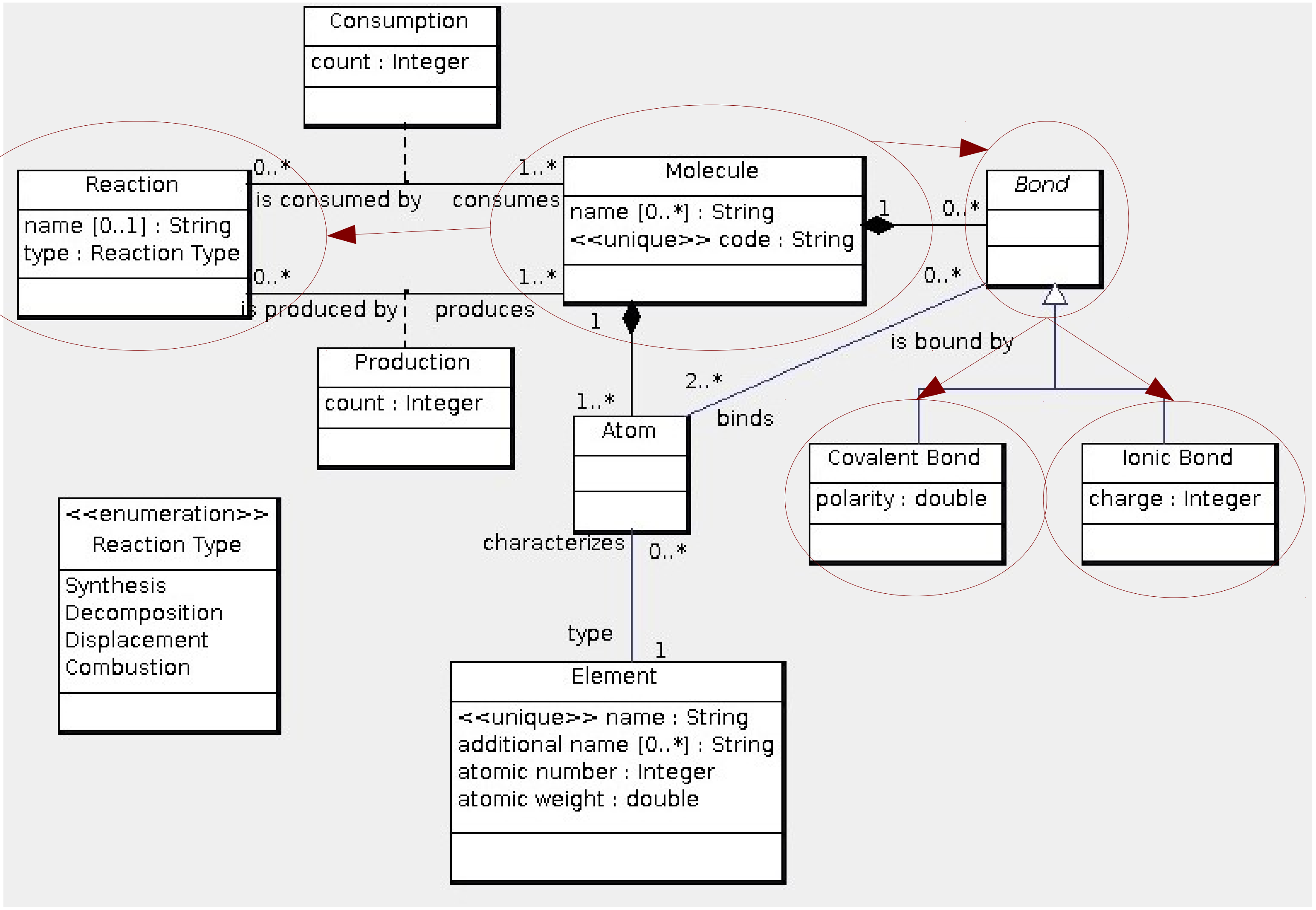
- The `exists` operator in a where clause applies to another query
  - Called a *subquery* or *nested query*
  - The subquery uses `select *` because it is only testing whether there is a row in the subquery, not what field values are in the row
- There are other operators that apply to subqueries but all are expressible in terms of the `exists` operator
- The scope of a table in a subquery is restricted to the subquery
  - The tables and their columns are not available outside the subquery
  - This is a common error when using subqueries

Subquery versus Duplicate Elimination

- A mentioned in Query 8, sometimes one can use `select distinct` or a `group by` rather than a nested query
- It is better to use nested queries for several reasons
  - The `exists` operator is more efficient than a join
    - One can stop when a row is found
  - Duplicate elimination and partitioning (group by) is very costly
    - Requires a separate sort or hash operation
  - It is not always correct to use duplicate elimination or partitioning (group by)
    - Requires not null and unique

Query 15

- Show the code and bonds for every molecule produced by a combustion reaction, showing the charge of an ionic bond and polarity of a covalent bond. If a molecule has no bonds, then do not show it at all.
- Begin with the navigation...


Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
consumes
1..*
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is produced by
produces
1..*
Production
count : Integer
1
is bound by
2..*
binds
1..*
Atom
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Query 15

- Show the code and bonds for every molecule produced by a combustion reaction, showing the charge of an ionic bond and polarity of a covalent bond. If a molecule has no bonds, then do not show it at all.
- This will require a nested query as well as outer joins
  - To constrain the molecules to those produced by combustion reactions, use a nested query
  - To show the ionic and covalent bonds, use outer joins

Solution to Query 15

```
select m.id, i.charge, c.polarity
  from Molecule m join Bond b on (m.id = b.partOf)
       left join IonicBond i on (i.id = b.id)
       left join CovalentBond c on (c.id = b.id)
 where exists(
         select *
           from Production p, Reaction r
          where p.produces = m.id
            and p.isProducedBy = r.id
            and r.type = 'Combustion'
       )
```

Query 16

- List all molecules by code that have a bond
- Similar to Query 14 but with a many-to-many relationship rather than a multivalued attribute
- Rewrite this query like this:

  Find the molecules m such that there exists a bond b such that b is a part of m, showing m.code for each m
- The important word here is “exists”
  - This differs from a join because it does not matter what bonds are a part of m or how many there are
  - All that matters is whether there is one

Solution to Query 16

- Here is a solution to Query 16:

```
select m.code
  from Molecule m
 where exists(
         select *
           from Bond b
          where b.partOf = m.id
       )
```

Semijoin

- The Molecule and Bond tables are joined, but it is a new kind of join, called a *semijoin*
  - The name *semijoin* comes from the fact that after joining the tables only the attributes of one table remain
  - Another difference with the join is that each row of the first table occurs at most once in the result of the query no matter how many matching rows are in the second table
- The nested query in Query 16 can use the table alias m, but the main query cannot refer to the table alias b
  - The scope of a nested query includes the query that contains it
  - The containing query cannot use anything in the nested query

Query 17

- List all molecules by code that are catalysts. A catalyst is a molecule that is both consumed and produced by a reaction with the same counts.
- This is Query 8, but now use a semijoin rather than a join
  - This will eliminate duplicates with much less effort
- Rewrite Query 17 like this:

  Find the molecules m such that there exists a reaction `r` such that `m` is produced by `r` and consumed `r` with the same counts, showing `m.code` for each `m`

Query 17

- Find the molecules `m` such that there exists a reaction `r` such that `m` is produced by `r` and consumed by `r` with the same counts, showing `m.code` for each `m`
- The query is

```
select m.code
  from Molecule m
 where exists(
         select *
           from Reaction r, Consumption c, Production p
          where c.consumes = m.id
            and p.produces = m.id
            and c.isConsumedBy = r.id
            and p.isProducedBy = r.id
            and c.count = p.count
       )
```

Query 17

- The query shows each molecule at most one time
  - The exists condition is either true or false
  - If the exists condition is true the `m.code` value is included in the results
  - If the exists condition is false the molecule is not included in the results
  - A molecule can never be included twice
- No duplicate molecules need to be eliminated
- The exists condition is true precisely when there is a reaction r with the required properties: r both produces and consumes m with the same counts
- In fact, the subquery need not be fully evaluated
  - When a row is found, the processing can stop

Simplifying Queries

- Queries can sometimes omit a table T in the from clause
- This is possible when all of the following are true:
  - No column other than the primary key of T is needed
  - The primary key of T is the target of a foreign key fk of a table S
  - The where clause of the query includes the join condition which equates the primary key of T with the foreign key fk of S
- When all of these conditions are true, then the table T can be omitted and all occurrences of the primary key of T can be replaced by the the foreign key column fk of table S.
- The foreign key constraint is crucial: If it is not enforced, then the simplification will not always give the correct results.

Simplifying Query 17

- This rule can be applied to Query 17 as follows:
  - T is the table `Reaction r`
  - S is the table `Production p`
  - The foreign key is `p.isProducedBy`
- The rule applies because
  - No column of `r` is needed in the query other than `r.id`
  - There is a foreign key constraint on `p.isProducedBy` with target `r.id`
  - The constraint `p.isProducedBy = r.id` is in the where clause
- Therefore, `Reaction r` can be omitted and `r.id` can be replaced by `p.isProducedBy`

Simplifying Query 17

The following is the result of the simplification rule:

```
select m.code
  from Molecule m
 where exists(
          select *
            from Consumption c, Production p
           where c.consumes = m.id
             and p.produces = m.id
             and c.isConsumedBy = p.isProducedBy
             and p.isProducedBy = p.isProducedBy
             and c.count = p.count
       )
```

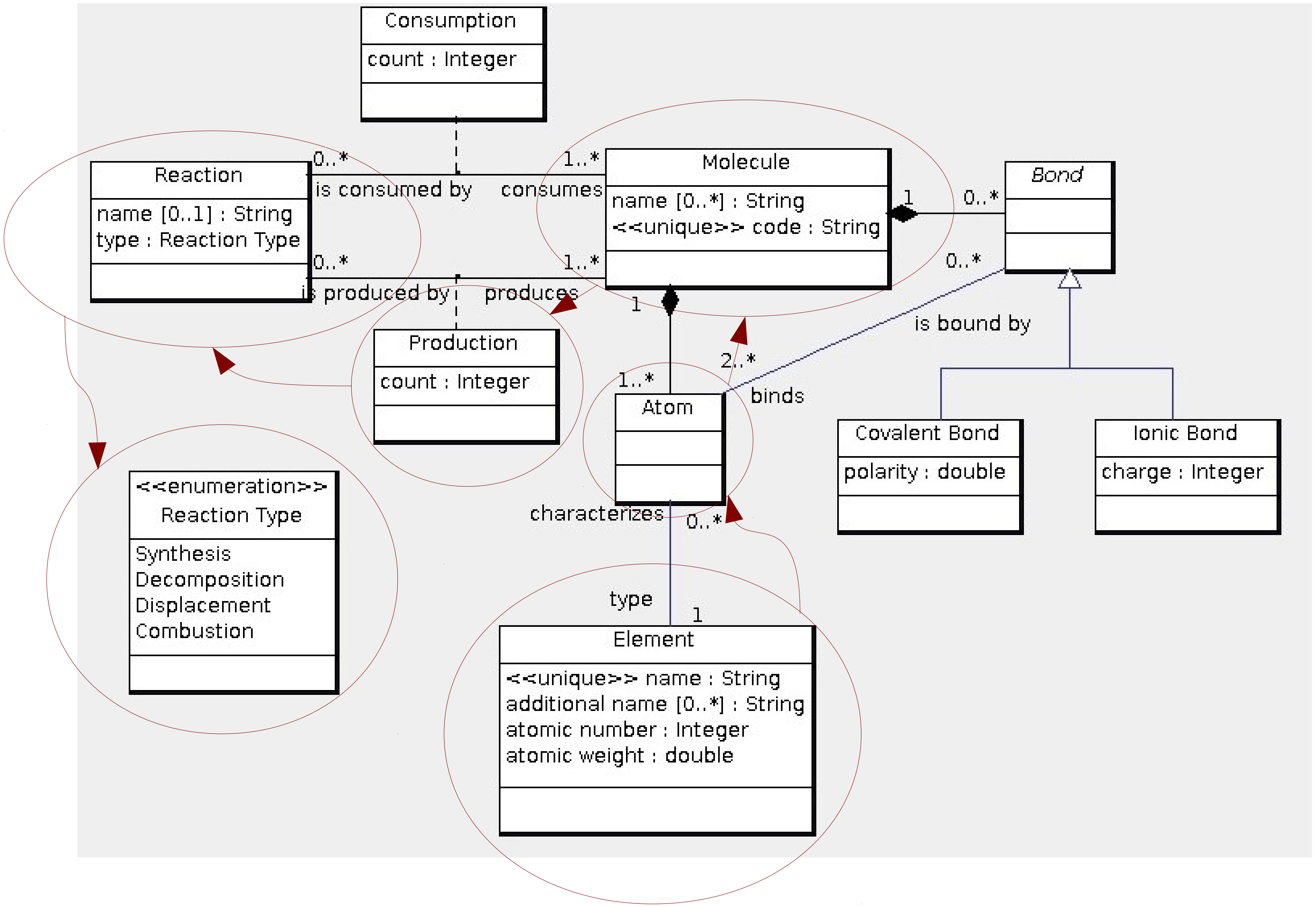
Simplifying Query 17

One of the conditions in the where clause is now redundant so it can be eliminated, giving the final query:

```
select m.code
  from Molecule m
 where exists(
         select *
           from Consumption c, Production p
          where c.consumes = m.id
            and p.produces = m.id
            and c.isConsumedBy = p.isProducedBy
            and c.count = p.count
       )
```

Query 18

- List all elements, showing name, atomic number and atomic weight, that are produced in a synthesis reaction
- Start by navigating...



Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
consumes
1..*
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is produced by
produces
1..*
1
is bound by
Production
count : Integer
2..*
1..*
binds
Atom
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Query 18

- List all elements, showing name, atomic number and atomic weight, that are produced in a synthesis reaction
- The query will have the form to ensure that each element that satisfies the requirement will occur exactly once:

```
select e.name, e.atomicNumber, e.atomicWeight
  from Element e
 where exists(
          ...
       )
```

Query 18

- List all elements, showing name, atomic number and atomic weight, that are produced in a synthesis reaction
- Joining the Element table with the `Atom`, `Molecule`, `Production` and `Reaction` tables gives the query:

```
select e.name, e.atomicNumber, e.atomicWeight
  from Element e
 where exists(
         select *
           from Atom a, Molecule m, Production p, Reaction r
          where a.type = e.id
            and a.partOf = m.id
            and p.produces = m.id
            and p.isProducedBy = r.id
            and r.type = 'Synthesis'
       )
```

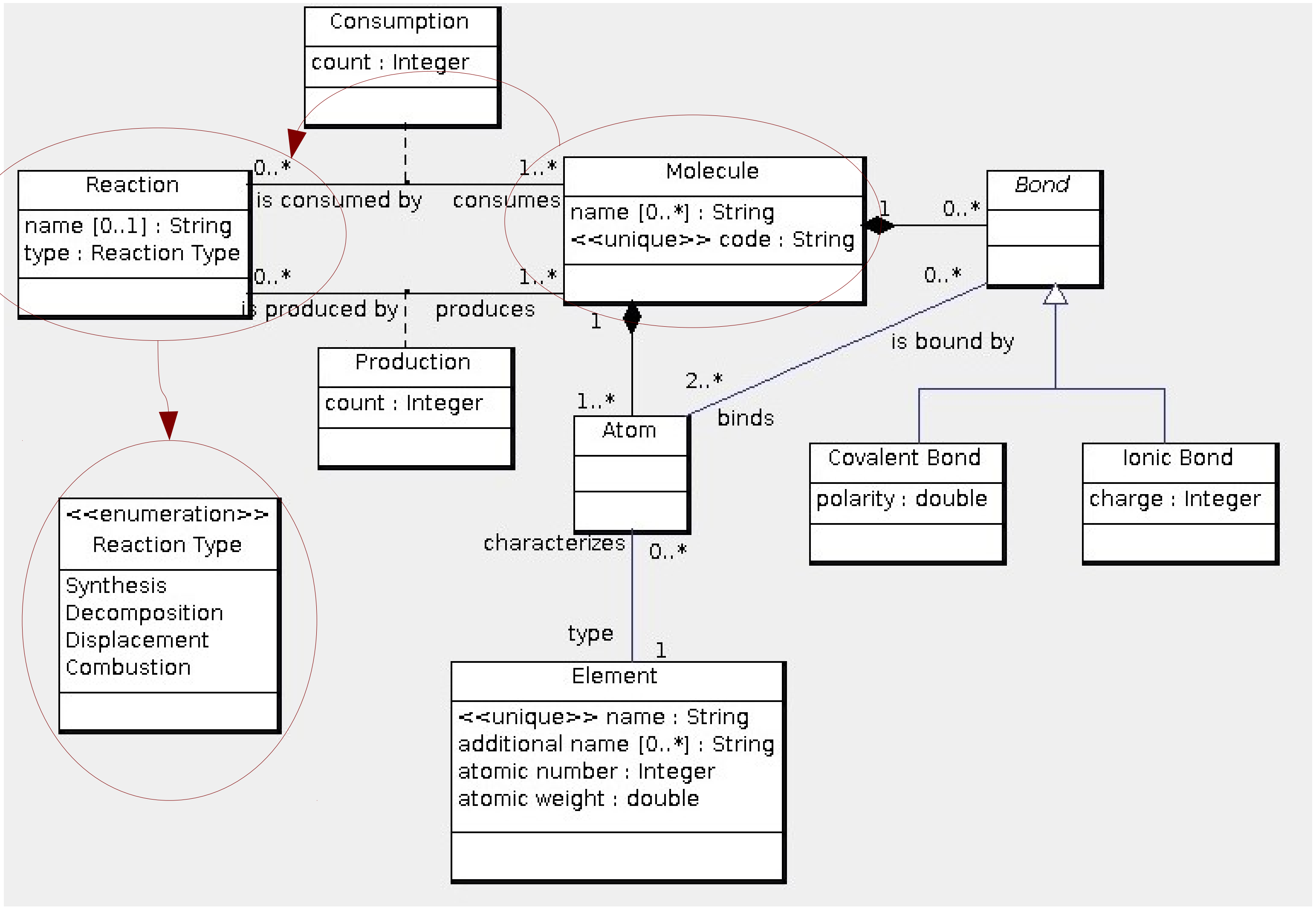
Simplifying Query 18

- List all elements, showing name, atomic number and atomic weight, that are produced in a synthesis reaction
- The query can be simplified to the following:

```
select e.name, e.atomicNumber, e.atomicWeight
  from Element e
 where exists(
         select *
           from Atom a, Production p, Reaction r
          where a.type = e.id
            and a.partOf = p.produces
            and p.isProducedBy = r.id
            and r.type = 'Synthesis'
       )
```

Query 19

- List all molecules by code that are consumed by a combustion reaction
- Start by navigating...


Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
consumes
1..*
0..*
is produced by
produces
1..*
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is bound by
Production
count : Integer
1
1..*
2..*
binds
Atom
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Query 19

- List all molecules by code that are consumed by a combustion reaction
- This should now be relatively routine:

```
select m.code
  from Molecule m
 where exists(
         select *
           from Consumption c, Reaction r
          where c.consumes = m.id
            and c.isConsumedBy = r.id
            and r.type = 'Combustion'
       )
```

Query 20

- List all molecules by code that are consumed by every combustion reaction
- This looks like the same query as Query 19, but it is very different.
  - “consumed by **a** combustion reaction” (Query 19)
  - “consumed by **every** combustion reaction” (Query 20)
- In Query 19, a single combustion reaction is sufficient for all of the molecules consumed by the reaction to be included in the results of the query
- In Query 20, all it takes is for one combustion reaction not to consume a molecule for that molecule not to be included in the results of the query

Quantifiers

- There exists is written ∃
- For every is written ∀
- SQL only supports ∃ because ∀ can be computed using ∃

  ∀P is equivalent to not (∃ (not (P)))

- To solve such a query:
  1. Rewrite it using the phrase “such that for every ...”
  2. Replace “such that for every ...” with “such that there does not exists ... such that ...”

Rewriting Query 20

- List all molecules by code that are consumed by every combustion reaction
- Rewrite this query as follows:

1. List all molecules m by code such that for every combustion reaction r the molecule m is consumed by r
2. List all molecules m by code such that there does not exist a combustion reaction r such that the molecule m is not consumed by r

Solution to Query 20

- List all molecules by code that are consumed by every combustion reaction

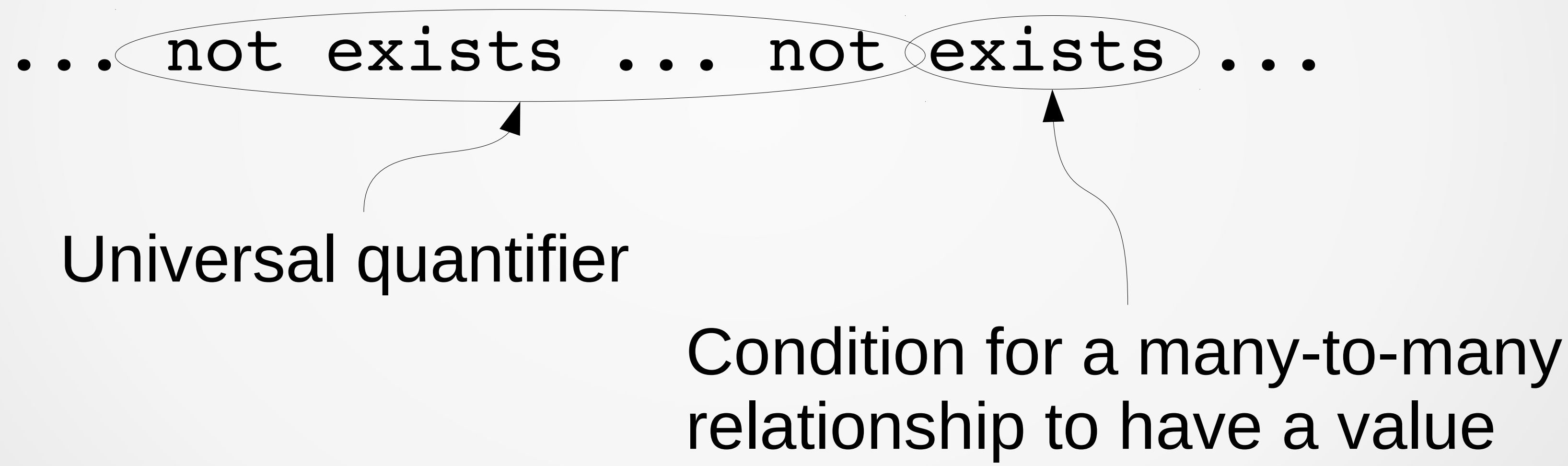
```
select m.code
  from Molecule m
 where not exists(
         select *
           from Reaction r
          where r.type = 'Combustion'
            and not exists(
                  select *
                    from Consumption c
                   where c.consumes = m.id
                     and c.isConsumedBy = r.id
                )
       )
```

Misunderstanding Quantifiers

- It is a common error to think that negations cancel
- This is true only for simple propositions that do not involve exists
- When `exists` is in between the negations, the existential quantifier changes to the universal quantifier

Understanding different uses of exists

- Query 20 has a where clause with two uses of the `exists` operator
- The two uses have different purposes:



```
... not exists ... not exists ...
```

Universal quantifier

Condition for a many-to-many relationship to have a value

The antijoin

- A `not exists` condition in a where clause is called an *antijoin*
- An antijoin is the negation of a semijoin.

Query 21

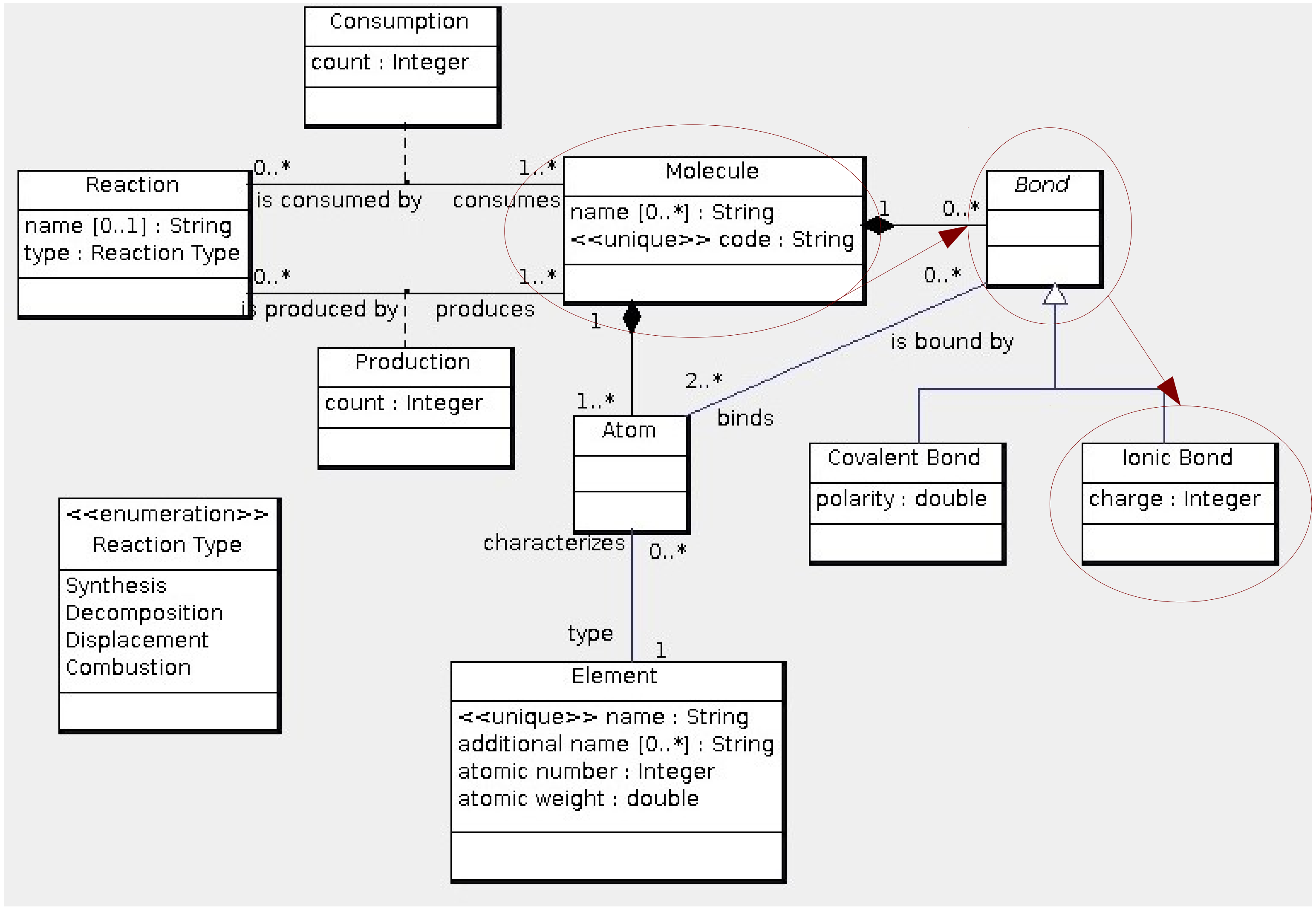
- List all elements, showing name, atomic number and atomic weight, that are produced by **every** synthesis reaction
- This looks like the same query as Query 18, but now **every** synthesis reaction rather than **some** synthesis reaction
- Rewrite the query like this:
  1. List all elements `e`, showing name, atomic number and atomic weight, such that for every synthesis reaction `r`, the element `e` is produced by `r`
  2. List all elements `e`, showing name, atomic number and atomic weight, such that there does not exist a synthesis reaction `r` such that the element `e` is not produced by `r`

Solution to Query 21

```
select e.name, e.atomicNumber, e.atomicWeight
  from Element e
 where not exists(
         select *
           from Reaction r
          where r.type = 'Synthesis'
            and not exists(
                  select *
                    from Atom a, Production p
                   where a.type = e.id
                     and a.partOf = p.produces
                     and p.isProducedBy = r.id
                )
       )
```

Query 22

- List all molecules by code that have an ionic bond with charge at least 5
- Begin by navigating...


Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
1..*
consumes
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is produced by
1..*
produces
1
Production
count : Integer
is bound by
2..*
1..*
Atom
binds
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Solution to Query 22

- List all molecules by code that have an ionic bond with charge at least 5
- The subclass relationship is joined like any other relationship.

```
select m.code
  from Molecule m
 where exists(
         select *
           from Bond b, IonicBond i
          where b.partOf = m.id
            and m.id = i.id
            and i.charge >= 5
       )
```

Alternative Subquery Syntaxes

- There are several other syntaxes for subqueries
- Since all of them can be converted to the exists form, it is a matter of personal style which one is preferable
- The `in` operator is popular
  - It tests whether a value or values occurs as a row of a relation
  - The next slides show Query 22 using the `in` operator in several ways

Alternative Solution to Query 22

- List all molecules by code that have an ionic bond with charge at least 5

```
select m.code
  from Molecule m
 where m.id in (
         select b.partOf
           from Bond b, IonicBond i
          where m.id = i.id
            and i.charge >= 5
       )
```

Converting between the `exists` and `in` operators

```
select m.code
  from Molecule m
 where exists(
         select *
           from Bond b, IonicBond i
          where b.partOf = m.id
            and m.id = i.id
            and i.charge >= 5
       )
```

```
select m.code
  from Molecule m
 where m.id in (
         select b.partOf
           from Bond b, IonicBond i
          where m.id = i.id
            and i.charge >= 5
       )
```

Another Solution to Query 22

- List all molecules by code that have an ionic bond with charge at least 5

```
select m.code
  from Molecule m
 where m.id in (
         select i.id
           from Bond b, IonicBond i
          where b.partOf = m.id
            and i.charge >= 5
       )
```

Other subquery contexts

- There are many other contexts for subqueries
- Most of these are best avoided because they are difficult to understand and can always be expressed in terms of the `exists` or `not exists` operators (i.e., using a semijoin or antijoin).
- Any equality or comparison operator can be used with `ANY` or `ALL` and a subquery
  - operator `ANY` is true if the operator is true for any result in the subquery
    - The `in` operator is the same as `=ANY`
  - operator `ALL` is true if the operator is true for every result in the subquery
    - The `not in` operator is the same as `<>ALL`

Set operations

- The set operations are union, intersection and difference
- The union is seldom needed, and the other two are so rare that database systems (such as MySQL) don't bother to support them.
  - The intersection can be implemented using a join
  - The difference can be implemented using an antijoin
  - The next slide has an example of a union query

Query 23

List all of the names of Reactions and Molecules without eliminating any duplicates

```
select r.name
  from Reaction r
union all
select n.name
  from MoleculeName n
```

Note the use of `union all` rather than just `union`. If the `all` is not specified, then duplicates are eliminated. Duplicate elimination is time-consuming and should be avoided unless it is essential.

Union Compatibility

- Two relations are *union compatible* if they have the same number of columns and corresponding columns have the same type
- One can perform the union of two relations only if they are union compatible

Landform Schema

```
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 Schema

```
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)
);
```

Valley Query Problem

List the valleys by name that are bordered by every hill that is part of a hill whose summit area exceeds 600.

Write your answer on a piece of paper.

Relational Algebra

Relational Algebra

- SQL is the standard API for accessing relational data
- The DBMS parses the SQL and translates it to another language: relational algebra
- An algebra is a mathematical structure
  - A collection of objects
  - Operations on the objects
  - Axioms on the operations
- Everyone should know about the algebra of real numbers
  - Collection is the set of real numbers
  - Operations are addition and multiplication
  - Axioms: commutativity, associativity, distributivity

Relational Algebra

- The relational algebra consists of operations and axioms for relations (tables)
- Why use relational algebra rather than SQL?
- Relational operators are the fundamental functions of a DBMS
  - Most relational operators have many implementations
- Axioms allow the DBMS to optimize queries
  - Select best implementation of each operator
  - Rearrange the query to improve performance
- None of this is possible using SQL alone

Relational Algebra Operators

- Projection π
- Extension ξ
- Selection σ
- Rename ρ
- Product X
- Join ⋈
- Left Join ⟕
- Right Join ⟖
- Full Outer Join ⟗
- Semijoin ⋉
- Antijoin ▷
- Union ∪
- Intersection ∩
- Difference −
- Partition ψ

Projection

- Use only some of the fields of each row.
- The rest of the fields are discarded.
- Written $\pi_s(R)$ where S is the set of fields of a relation R that should be retained

Extension

- Add a new field to every row by computing it from the values of the other fields
- For a relation R write $\xi_f(R)$ for the relation obtained by adding a new column to R using the function f
- The select clause of an SQL query may require the use of the projection, extension, rename and partition operators.

Selection

- Select some of the rows of a relation
- For a relation R and a logical expression P, the selection of R using P is written $\sigma_P(R)$
  - P is true for the rows in $\sigma_P(R)$, and P is false for the rest of the rows in R
- This use of the word “select” is unrelated to the select clause of an SQL query
- The logical expression P is also called the *predicate*

Rename

- Change the name of a column
- Written $\rho_{A,B}(R)$ where A is the old name of a column of R and B is the new name for the column
- Used for specifying a column alias

Cartesian Product

- For relations R and S, the cartesian product is the relation obtained by combining rows of R and S in every possible way.
- If R has M rows, and S has N rows, then R x S has MN rows.
- This operation is rarely used by itself because the size of the result is so large.

Join

- The inner join of two relations
- The join of relations R and S is written $R \bowtie_J S$ where J is the join condition
- The join is the same as a selection applied to a cartesian product, but it is so common (and the cartesian product is so rare) that it was given its own operator.
- There are 4 variations on the join operator.

Joins

The join of two relations R and S using J as the join condition

- Inner join is R $\bowtie_J$ S
- Left outer join is R $⟕_J$ S
- Right outer join is R $⟖_J$ S
- Full outer join is R $⟗_J$ S

Semijoin and Antijoin

- The *semijoin* of relations R and S and a join condition J is the set of rows of R that satisfy the join condition J with S
  - It is the same as joining R and S using J and then projecting only the columns of R
  - In terms of SQL it corresponds to an `exists` subquery.
  - It is written $R \ltimes_J S$
- The *antijoin* of relations R and S and a join condition J is the set of rows of R that do not satisfy the join condition J with S
  - It corresponds to an SQL `not exists` subquery
  - It is written $R \rhd_J S$

Set Operations

- The set operations of union, intersection and difference are the same as in mathematics.
- They can be applied to relations R and S only if the two relations have the same number of columns and corresponding columns have the same type.
- The operations are written the same way as in mathematics:
  - Union $R \cup S$
  - Intersection $R \cap S$
  - Difference $R - S$

Partition

- This includes a number of operations that seem to be very different from one another, yet are implemented in the same ways.
  - Grouping
  - Sorting
  - Eliminating duplicates
- It differs from the other operations because its output is not a relation, so strictly speaking, it is not a relational algebra operation
- For a relation R, partitioning using criterion C is written $\psi_C(R)$
  - This is not a standard notation

Landform Schema

```
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
);
```

The JOINED strategy is used for subclasses.

Relational Algebra Problems

Express the query “List the valleys by name whose area exceeds 600” using the relational algebra.

Express the query “List the hills by name that are part of a hill whose summit area exceeds 600” using the relational algebra.

Write your answers on a piece of paper.

Modifying Data

Modifying the Database

- Data is modified with three commands
  - Insert to create new records
  - Delete to delete existing records
  - Update to modify existing records
- Together with queries, the commands are known as the CRUD commands for Create, Read, Update, Delete
- The CRUD commands are necessary for any kind of data storage, not just relational databases

Insert Command 1

Create a new decomposition reaction with id 534, named “ammonium nitrate decomposition”

The following command is correct but depends on knowing the order of the columns in the create table statement:

```
create table Reaction (
  id int primary key,
  name varchar(200),
  type enum ('Synthesis',
    'Decomposition',
    'Displacement',
    'Combustion') not null
);
```

```
insert into Reaction
  values(534,
         'ammonium nitrate decomposition',
         'Decomposition')
```

Insert Command 1

Create a new decomposition reaction with id 534, named “ammonium nitrate decomposition”

A better solution is to explicitly specify the columns in the insert command:

```
create table Reaction (
  id int primary key,
  name varchar(200),
  type enum ('Synthesis',
    'Decomposition',
    'Displacement',
    'Combustion') not null
);
```

```
insert into Reaction(id, name, type)
  values(534,
         'ammonium nitrate decomposition',
         'Decomposition')
```

Autoincrement

- Creating new identifiers can be done with the autoincrement feature
  - It is inconvenient have to come up with a new primary key value for each new reaction record
  - The value might already be in use
- The syntax for autoincrement differs from one DBMS to another
  - The syntax shown here is for MySQL
- If the autoincrement field is omitted in the insert command, the field value will be generated by the database

```
create table Reaction (
  id int primary key
    auto_increment,
  name varchar(200),
  type enum ('Synthesis',
    'Decomposition',
    'Displacement',
    'Combustion') not null
);
```

```
insert into Reaction(name, type)
  values('ammonium nitrate decomposition',
         'Decomposition')
```

Strategies for generating field values

There are many strategies for generating field values

- Increment by 1 the highest value currently in the field
- Increment by some number other than 1
- Maintain a table with a single row containing a number
    - Whenever a new number is needed increment the number in the special table
    - This can be used for generating field values in many tables
    - Avoids conflicts throughout the database

Insert Command 2

Make the molecule with code “H2O” into one of the products of ammonium nitrate decomposition with count 2. Assume that the molecule and decomposition reaction are already in the database.

Although this only requires adding one record to the database, it has an issue that must be handled:

- To insert into the Product table, one must have the ids of the molecule and reaction
- One could look them up manually using a query, but this is tedious and error-prone

The solution is to use another kind of insert command

Solution to Insert Command 2

Make the molecule with code “H2O” into one of the products of ammonium nitrate decomposition with count 2. Assume that the molecule and decomposition reaction are already in the database.

```
insert into Production(produces, isProducedBy, count)
  select m.id, r.id, 2
    from Molecule m, Reaction r
   where m.code = 'H2O'
     and r.name = 'ammonium nitrate decomposition'
     and r.type = 'Decomposition'
```

Delete Command 1

Delete the decomposition reaction named “ammonium nitrate decomposition”

Here is the command:

```
delete from Reaction
  where name = 'ammonium nitrate decomposition'
    and type = 'Decomposition'
```

Effects of Delete Command 1

Delete the decomposition reaction named “ammonium nitrate decomposition”

The command actually deletes all of the records that satisfy the where clause

- The name field is not unique so there could be more than one matching record

The command also deletes any related Consumption and Production records because of the `on delete cascade` in the create table statements

Delete Command 2

Revise the database so that the molecule with code “H2O” is not one of the products of the decomposition reaction named “ammonium nitrate decomposition”

This requires information that is not in the Production table so a subquery is necessary

```
delete from Production
  where exists(
    select *
      from Reaction r, Molecule m
     where r.id = isProducedBy
      and m.id = produces
      and r.name = 'ammonium nitrate decomposition'
      and r.type = 'Decomposition'
  )
```

Update Command 1

Revise the database so that the name of the decomposition reaction named “ammonium nitrate decomposition” has the name “AND”

This requires information that is not in the Production table so a subquery is necessary

```
update Reaction set name = 'AND'
  where name = 'ammonium nitrate decomposition'
    and type = 'Decomposition'
```

Update Command 2

Revise the database so that the molecule with code “H2O” is produced with count one larger than the current count as a product of the decomposition reaction named “ammonium nitrate decomposition”

This requires information that is not in the Production table so a subquery is necessary

```
update Production set count = count + 1
  where exists(
    select *
      from Reaction r, Molecule m
     where r.id = isProducedBy
       and m.id = produces
       and r.name = 'ammonium nitrate decomposition'
       and r.type = 'Decomposition'
  )
```

Views

Virtual Tables

- A view or virtual table is a named query
- A view can be used in a query as if it was a table.
  - Can be used to simplify complex queries
  - Important for restricting permissions to parts of tables
- Modification commands can use a view only if the view is *updatable*
  - Only relatively simple views can be used for updates
  - Complex views cannot be used for updates because the modifications to the database might not be unique or even possible

View 1

Create a view that includes the Combustion reactions

The following view uses the same names for the columns in the view as in the Reaction table

```
create table Reaction (
  id int primary key,
  name varchar(200),
  type enum ('Synthesis',
    'Decomposition',
    'Displacement',
    'Combustion') not null
);
```

```
create view Combustion1 as
  select *
    from Reaction r
   where r.type = 'Combustion'
```

View 2

Create a view that includes the Combustion reactions but rename the columns and do not include the type column

The following view uses different names for the columns in the view

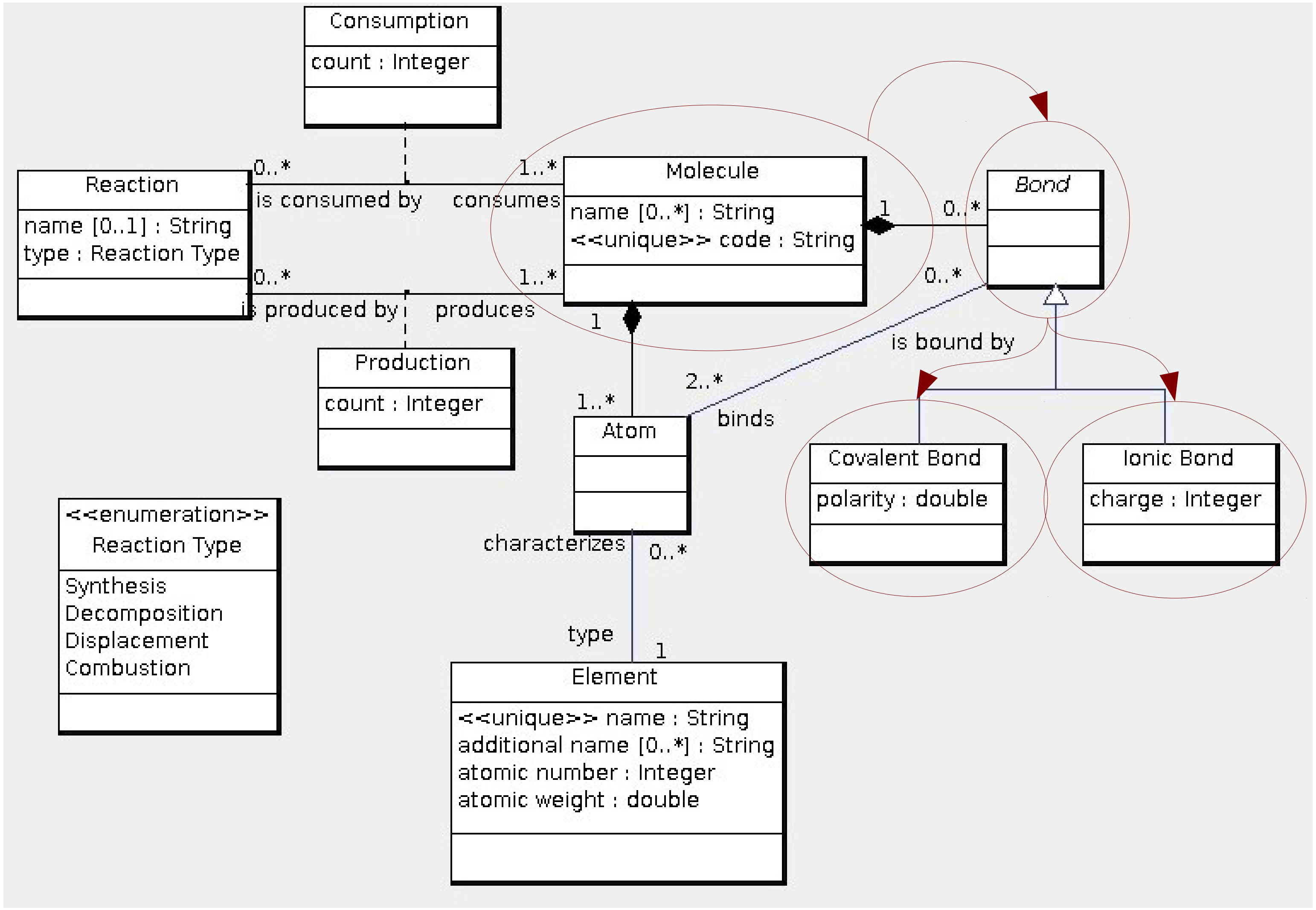
```
create table Reaction (
  id int primary key,
  name varchar(200),
  type enum ('Synthesis',
    'Decomposition',
    'Displacement',
    'Combustion') not null
);
```

```
create view Combustion2(identifier, combustionName) as
  select r.id, r.name
    from Reaction r
   where r.type = 'Combustion'
```

View 3

Create a view for molecules and their bonds that shows the charge and polarity

Begin with navigation...


Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
consumes
1..*
0..*
is produced by
produces
1..*
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is bound by
1
Production
count : Integer
1..*
Atom
2..*
binds
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

View 3

Create a view for molecules and their bonds that shows the charge and polarity

This view is a join of four tables: Molecule, Bond, CovalentBond and IonicBond. Outer joins are necessary.

```
create view MoleculeBond(mid, code, bid, charge, polarity) as
  select m.id, m.code, b.id, i.charge, c.polarity
    from Molecule m join Bond b on (m.id = b.partOf)
         left join CovalentBond c on (c.id = b.id)
         left join IonicBond i on (i.id = b.id)
```

Using Views

- Find molecule and bond data about all bonds that have a charge that is at least 5

```
select *
  from MoleculeBond mb
 where mb.charge >= 5
```

Assignment #4