CS5200
Database Management
Use Cases and Basic Queries

Ken Baclawski

Spring 2017

Outline

- Use Cases Continued
- Project Functional Requirements
- Basic Queries
- Assignment #3

© 2017 Ken Baclawski All Rights Reserved

Use Cases

© 2017 Ken Baclawski All Rights Reserved

Use Case Step

- Defines an elementary interaction event
- Initiated by an actor or by the system
- Performs an action which is one of these:
  - An actor performs an action with the system
  - The system performs an action that either modifies the state of the system or affects an actor
- The action *must* be described by an active verb
- An actor never performs an action with another actor
  - A use case only describes interactions between the actors and the system

Alternative flow versus exception

- An alternative flow is not another use case
- An alternative flow does not have a precondition or postcondition
- Upon completion of an alternative flow, the use case postcondition will be true
- An alternative flow does not return to the use case that invoked the alternative flow

- An exception is another use case
- An exception has its own precondition and postcondition
- A use case that invokes an exception may not satisfy the use case postcondition
- An exception does not return to the use case that invoked the exception

© 2017 Ken Baclawski All Rights Reserved

Examples of common errors

- Using a passive verb in a step
  - “The update is confirmed”
  - Does not specify which actor confirms the update
- Using an active verb in a precondition or postcondition
  - “An editor requests a document”
  - Conditions are logical expressions not actions being performed
- Interactions between actors
  - “The editor asks the administrator for permission to update a document”
  - Does not involve the system, so it is irrelevant
- Implementation details
  - “The user interface subsystem calls the storage subsystem to store the document”
  - Never mention subsystems, modules, classes, etc.
  - Use cases define requirements not design or implementation details

Common Trap

- By far the most common error is to regard use case descriptions as a kind of programming language
- It is tempting for a programmer to use familiar constructs such as constructs similar to if and while statements in a use case description.
- In other words, the use case developer is thinking in terms of implementing the system rather than specifying its requirements.
- Unfortunately, many textbook authors and the Wikipedia article on use cases have fallen into this trap.
- Many other misunderstandings about use cases are the result of falling into this same trap.

Sample Requirements Document

- The Open Ontology Repository
  - If the link does not work, then type in this URL:

    `http://www.ccs.neu.edu/home/kenb/ontologies/oor-usecase.xml`
- Complete requirements document for the OOR
  - Conforms to the UCDO standard
  - Written in the Web Ontology Language (OWL)
  - Uses CSS for viewing the requirements

    `http://www.ccs.neu.edu/home/kenb/ontologies/auxfiles/styleowl.xsl`
- The XML format is available at:

  `http://www.ccs.neu.edu/home/kenb/ontologies/oor-usecase.owl`

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- The following slides discuss a number of claimed limitations that are in the Wikipedia article on use cases.
  - Some are true limitations to some extent
  - Most are false and do not represent limitations of use cases
- There are many misunderstandings about use cases
  - Most of the misunderstandings are due to authors who do not understand use cases, and as a result the misunderstandings have become established

Misunderstandings

- “Not appropriate for non-functional requirements”

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- “Not appropriate for non-functional requirements”
  - This is true but it is not a *limitation* of use cases.
  - Use cases were *always* only intended for functional requirements.
  - Nonfunctional requirements such as performance guarantees are declared separately.

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- “No standard definitions”

Misunderstandings

- ~~“No standard definitions”~~
  - This is incorrect since there is a standard.
  - Unfortunately, the standard is not as well-known as some informal, non-standard definitions of use cases.

Misunderstandings

- “The user interface is not supposed to be reflected in a use case”

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- ~~“The user interface is not supposed to be reflected~~ in a use case”
  - This is incorrect. The user interface should be specified. However, the implementation details should not be specified.
  - An example of the mistake of programming rather than specifying.

Misunderstandings

- “Some use case relationships are ambiguous and difficult to understand”

Misunderstandings

- ~~“Some use case relationships are ambiguous and~~ difficult to understand”
  - This is incorrect. The standard is unambiguous and relatively easy to understand.
  - This misunderstanding is largely due to the popularity of the informal, non-standard definitions of use cases by authors who did not understand use cases.

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- “Use cases can be over-emphasized”

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- “Use cases can be over-emphasized”
  - This is true, but any technique or language can be used for purposes that were never intended.
  - Not a limitation of use cases if they are developed as intended.

Misunderstandings

- “Use cases must be modified to be used for test design”

Misunderstandings

- ~~“Use cases must be modified to be used for test design”~~
  - This shows a confusion between scenarios and use cases. A use case is a class of scenarios.
  - Since each test is based on a scenario, a use case cannot be directly used for test design.
  - One must *specialize*, not *modify*, a use case to a scenario to design a test case.

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- “Use cases do not show their motivations”

Misunderstandings

- “Use cases do not show their motivations”
  - This is true, but it is not a limitation of use cases since use cases are not intended to be used for decision making.
  - There are many tools and methodologies for decision making, including a formal decision making process.

Misunderstandings

- “Use cases are not agile”

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- ~~“Use cases are not agile”~~
  - This is incorrect. Agile development techniques do allow use cases.

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- “Use cases are diagrams”

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- ~~“Use cases are diagrams”~~
  - This is incorrect. In fact, use case diagrams are not very informative, and can even be misleading.
  - Unfortunately, this is one of the most common misunderstandings.

Misunderstandings

- “Use cases have too much user interface content”

Misunderstandings

- ~~“Use cases have too much user interface content”~~
  - This is not a limitation of use cases but rather an example of specifying use cases incorrectly
  - An example of the mistake of programming rather than specifying.

© 2017 Ken Baclawski All Rights Reserved

Misunderstandings

- “Use cases are a waste of time”

Misunderstandings

- ~~“Use cases are a waste of time”~~
  - This is incorrect. Use cases have proven to be valuable.

Project Functional Requirements

© 2017 Ken Baclawski All Rights Reserved

Requirements

- Use Case Descriptions
- The number of points must be at least 10 times the number of team members
- The requirements can be changed later, but must have enough points
- If a student leaves the team, then the requirements should be reduced
- A student cannot join another team

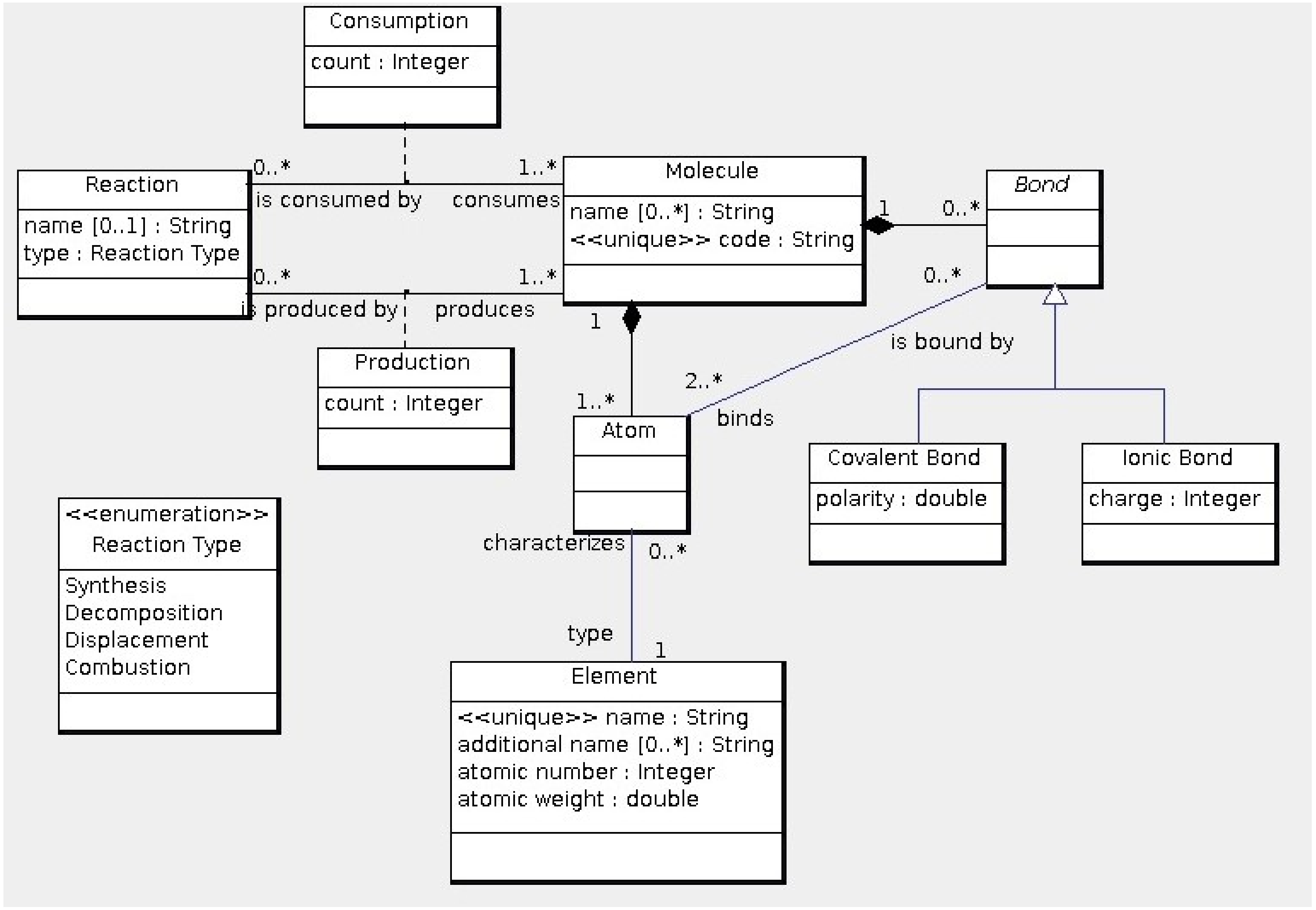
© 2017 Ken Baclawski All Rights Reserved

Basic Queries

Query Language

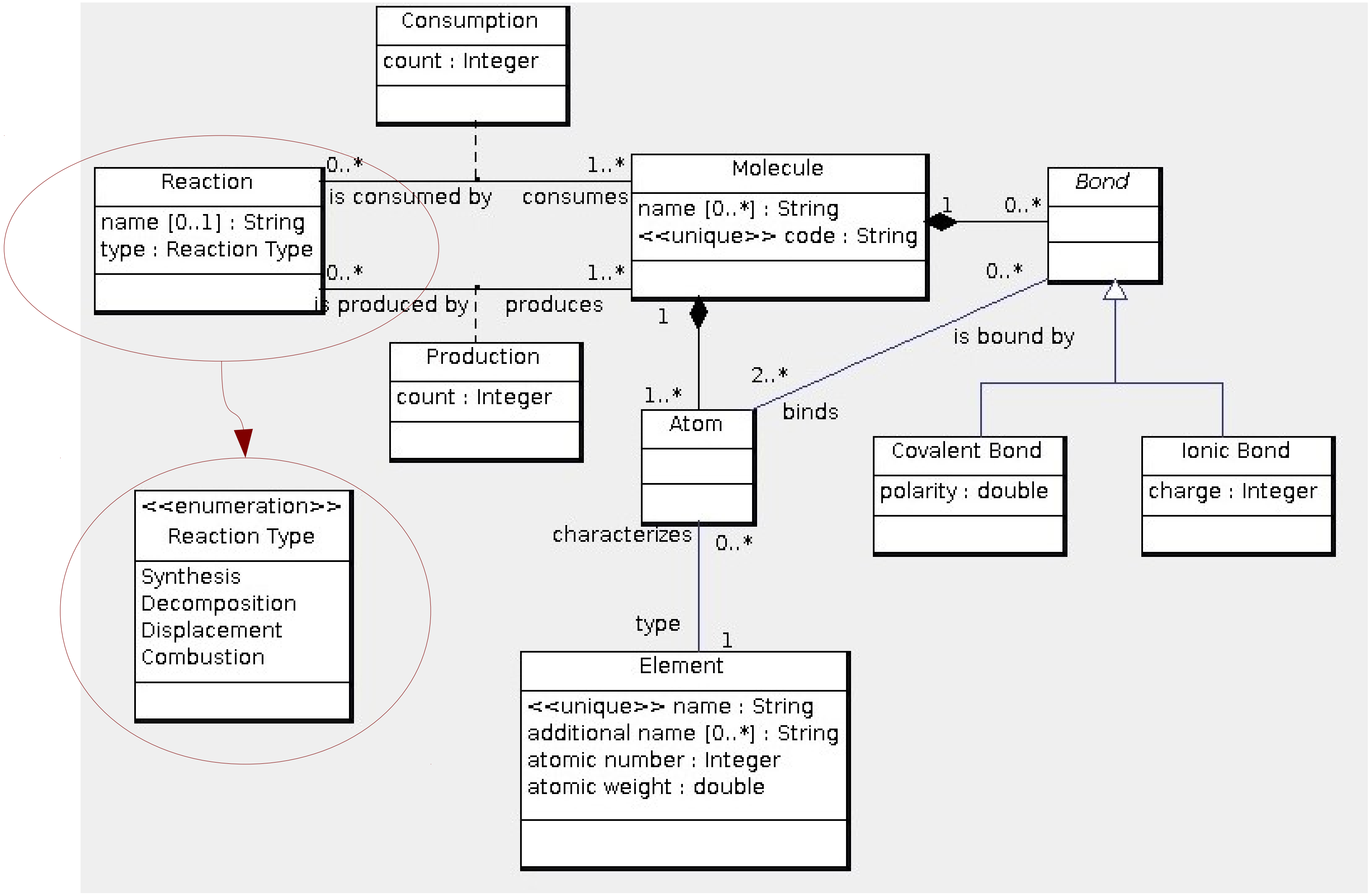
- A query extracts information from a database
  - A query consists of clauses
  - Each clause has a different purpose
- From clause: The source of the data
- Where clause: Which rows should be extracted
- Select clause: The fields to be returned from each row
- Other clauses will be introduced later

Example queries will use the chemistry database



Query 1

- What are the ids, names and types of all reactions?


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
2..*
binds
1
1..*
Production
count : Integer
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 1

What are the ids, names and types of all reactions?

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

© 2017 Ken Baclawski All Rights Reserved

Query 1

What are the ids, names and types of all reactions?

The data is in the Reaction table so the from clause is

```
from Reaction
```

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

© 2017 Ken Baclawski All Rights Reserved

Query 1

What are the ids, names and types of all reactions?

The data is in the Reaction table so the from clause is

```
from Reaction
```

The fields to be returned are the id, name and types fields so the select clause is

```
select id, name, type
```

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

© 2017 Ken Baclawski All Rights Reserved

Query 1

What are the ids, names and types of all reactions?

The data is in the Reaction table so the from clause is

```
from Reaction
```

The fields to be returned are the id, name and types fields so the select clause is

```
select id, name, type
```

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

The whole query is

```
select id, name, type
   from Reaction
```

© 2017 Ken Baclawski All Rights Reserved

SQL Syntax

- The logical order of the clauses is from-where-select
- However, the SQL syntax requires select-from-where
- The where clause can be omitted if all of the rows are to be extracted
- If all of the fields are to be returned, then use an asterisk:

```
select *
  from Reaction
```

Query 2

What are the ids and names of all combustion reactions?

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

© 2017 Ken Baclawski All Rights Reserved

Query 2

What are the ids and names of all combustion reactions?

The data is in the Reaction table so the from clause is

```
from Reaction
```

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

Query 2

What are the ids and names of all combustion reactions?

The data is in the Reaction table so the from clause is

```
   from Reaction
```

The rows to be extracted have type equal to 'Combustion' so the where clause is

```
 where type='Combustion'
```

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

Query 2

What are the ids and names of all combustion reactions?

The data is in the Reaction table so the from clause is

```
   from Reaction
```

The rows to be extracted have type equal to 'Combustion' so the where clause is

```
 where type='Combustion'
```

The fields to be returned are id and name so the select clause is

```
select id, name
```

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

© 2017 Ken Baclawski All Rights Reserved

Query 2

What are the ids and names of all combustion reactions?

The data is in the Reaction table so the from clause is

```
   from Reaction
```

The rows to be extracted have type equal to 'Combustion' so the where clause is

```
 where type='Combustion'
```

The fields to be returned are id and name so the select clause is

```
select id, name
```

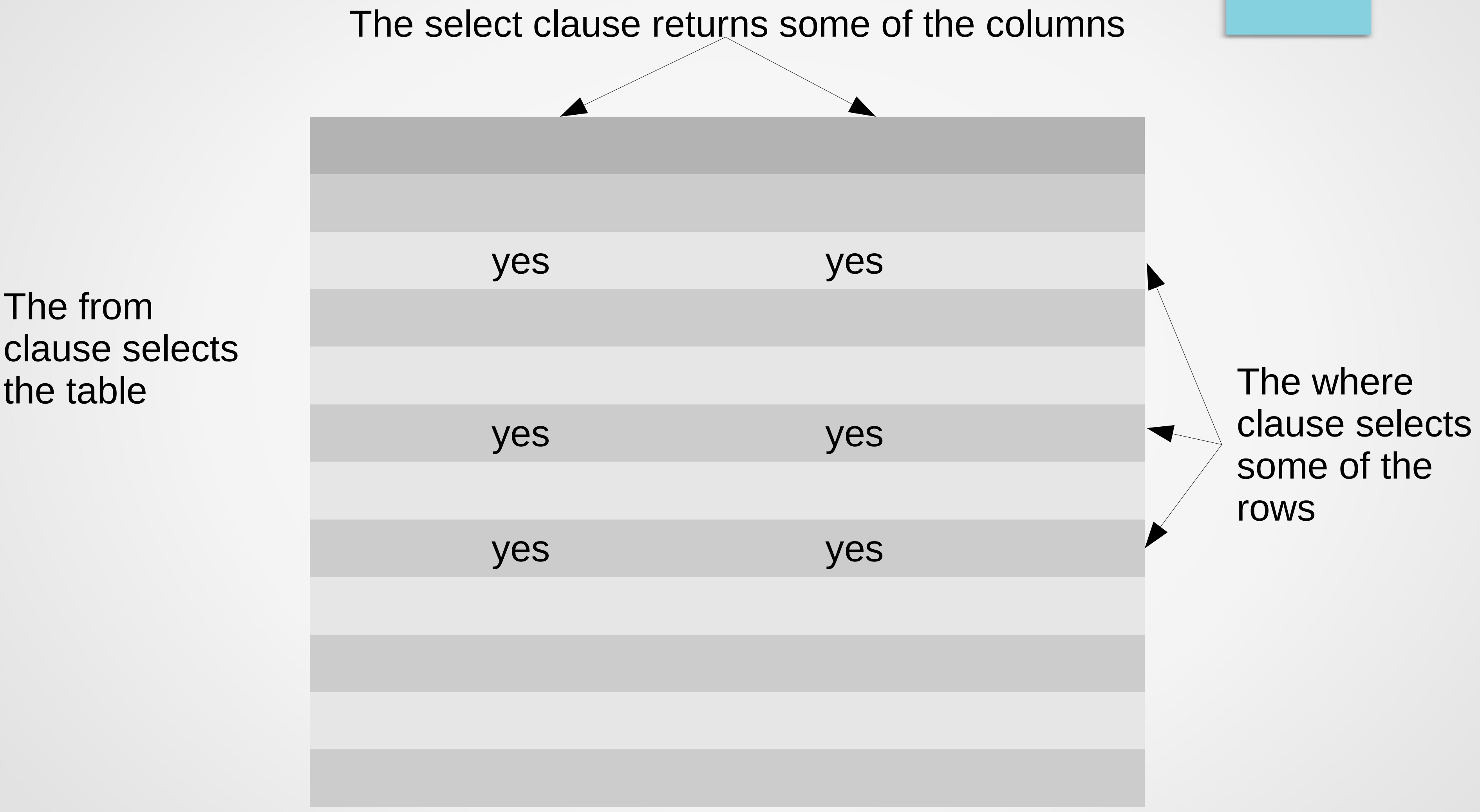
```
create table Reaction (
  id int primary key,
  name varchar(200),
  type enum ('Synthesis',
    'Decomposition',
    'Displacement',
    'Combustion') not null
);
```

The whole query is:

```
select id, name
  from Reaction
 where type='Combustion'
```

© 2017 Ken Baclawski All Rights Reserved

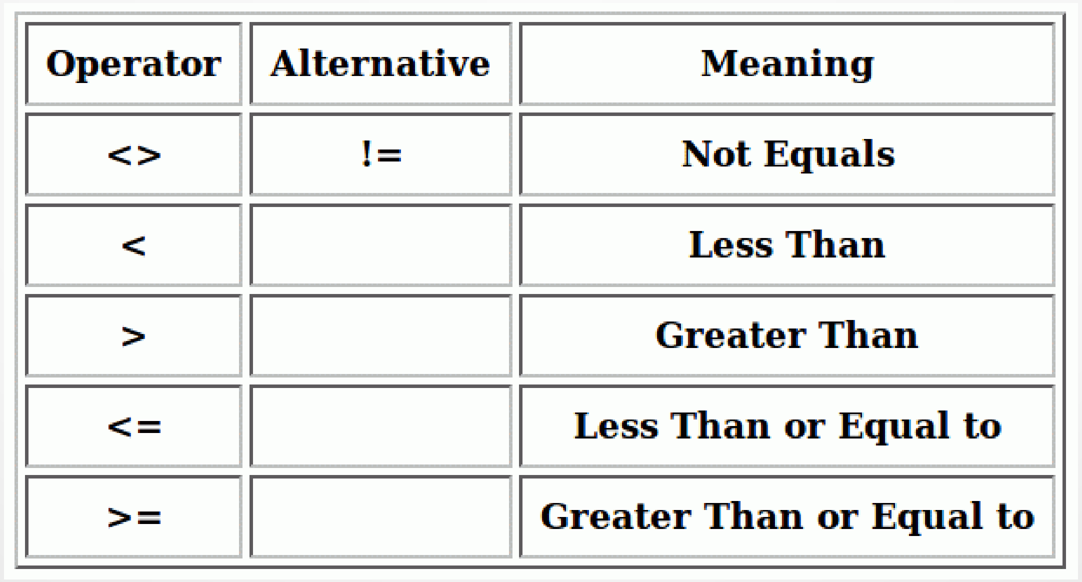
Querying a Table


The select clause returns some of the columns
The from clause selects the table
The where clause selects some of the rows
yes
yes
yes
yes
yes
yes

© 2017 Ken Baclawski All Rights Reserved

SQL Syntax

- The equality comparison is a single equal sign
  - SQL does not have an assignment statement so there is no ambiguity
- Here are some of the other comparison operators:



Operator	Alternative	Meaning
<>	!=	Not Equals
<		Less Than
>		Greater Than
<=		Less Than or Equal to
>=		Greater Than or Equal to

NULL

- The Reaction name field is allowed to be null
  - If a reaction does not have a name its field will be marked as NULL
  - If a reaction has a null name field, then the name field of the reaction has no value.
  - It does **not** mean the name field for this row has value NULL
- The SQL NULL is not a value
- Essentially all textbooks make this mistake!
- This will be discussed again later on.

Query 3

What are the ids and names of all combustion reactions that have a name?

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

© 2017 Ken Baclawski All Rights Reserved

Query 3

What are the ids and names of all combustion reactions that have a name?

We need to add another condition to the where clause.

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

Query 3

What are the ids and names of all combustion reactions that have a name?

How about this:

```
select id, name
  from Reaction
 where type='Combustion'
   and name<>null
```

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

Query 3

What are the ids and names of all combustion reactions that have a name?

How about this:

```
select id, name
  from Reaction
 where type='Combustion'
   and name<>null
```

This does not work because null is not a value so it is not meaningful to compare with null

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

© 2017 Ken Baclawski All Rights Reserved

Query 3

What are the ids and names of all combustion reactions that have a name?

The correct query is:

```
select id, name
  from Reaction
 where type='Combustion'
   and name is not null
```

SQL has a special syntax to deal with null

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

© 2017 Ken Baclawski All Rights Reserved

Three-Valued Logic

- When a comparison fails, SQL marks it as being null
  - Comparison with null
  - Division by 0
  - Boolean field with no value
- Logical expressions use the so-called *three-valued logic*
  - Technically it is still two-valued but it allows for the possibility of the lack of a value

AND	**True**	**False**	**Null**
True	True	False	Null
False	False	False	False
Null	Null	False	Null

OR	**True**	**False**	**Null**
True	True	True	True
False	True	False	Null
Null	True	Null	Null

Query 4

What are the ids and names of all reactions that have a name that begins with 'Aldol'?

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

Wildcard Matches

- SQL has a “wildcard” match feature
- The syntax is:

  column-name `like` pattern

- In the pattern there are two special characters:

  `%` matches zero or more arbitrary characters

  `_` (underscore) matches exactly one arbitrary character

- Examples
  - The pattern for strings that end with “tion” is `'%tion'`
  - The pattern `'exten_ion'` will match “extension”, “extention”, etc.

© 2017 Ken Baclawski All Rights Reserved

Query 4

What are the ids and names of all reactions that have a name that begins with 'Aldol'?

The query is:

```
select id, name
  from Reaction
 where name like 'Aldol%'
```

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

© 2017 Ken Baclawski All Rights Reserved

Other comparison operators

- Test for within a range: `between ... and ...`

  For example, `(id between 101 and 200)`

  is the same as `(id >= 101) and (id <= 200)`
- Test for membership in a set: `in ...`

  For example, `(id in (101, 106, 203, 307))`

Query 5

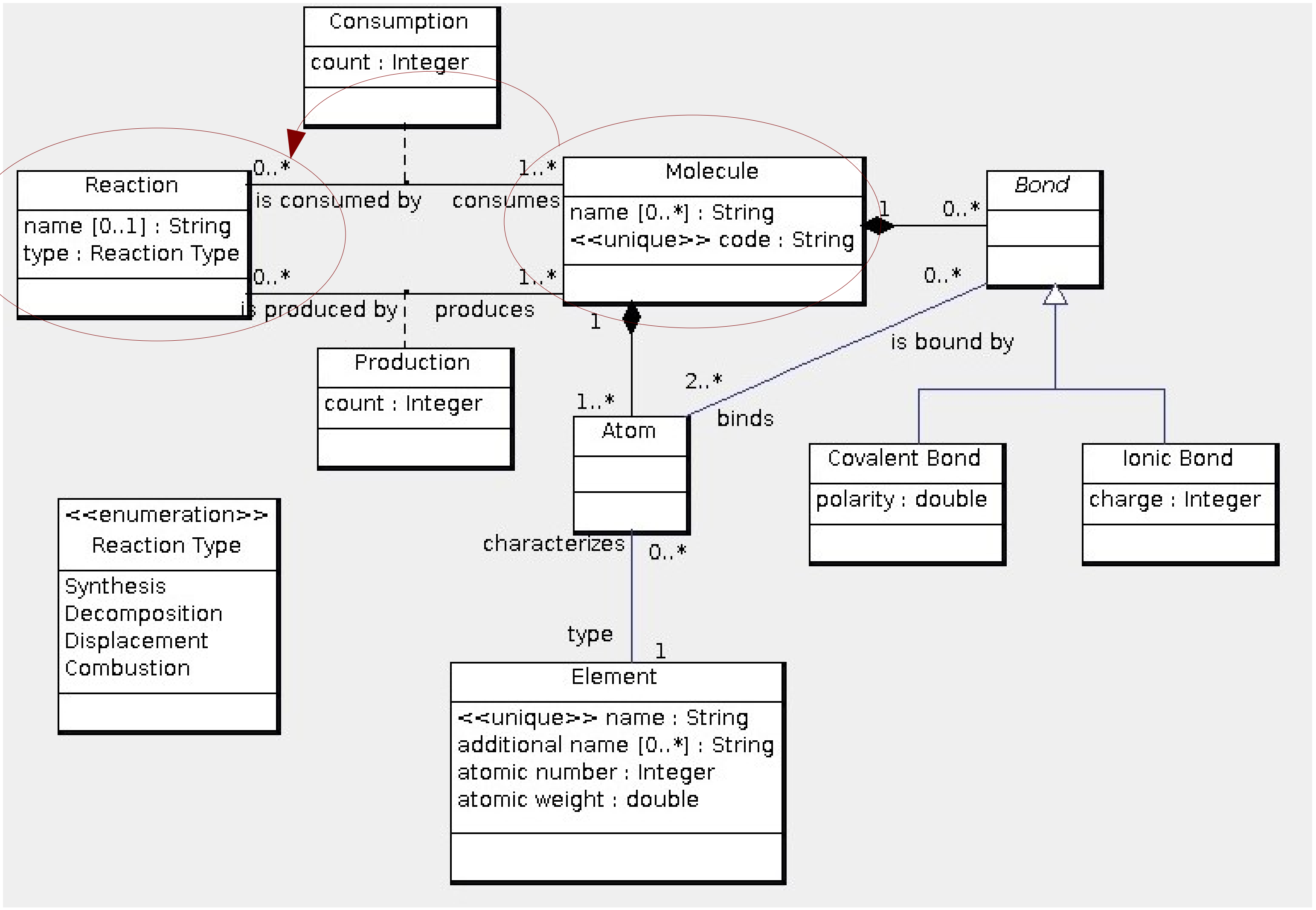
- List all synthesis reactions
- Up to now all queries specified the attributes that were to be returned. This query asks for a list of objects, not the attributes of objects.
- By convention *in this course*,
  - When a query asks for objects, the primary keys must be returned.
  - When a query asks for attributes of objects, only the attributes should be returned.
- The query is:

```
select id
  from Reaction
 where type='Synthesis'
```

Query 6

- List all molecules consumed by a reaction named WFO
- Up to now all queries were for attributes of objects in a single class.
- This query requires objects from several classes as well as the associations between them.
- The process of determining which classes and associations are required is called *navigation*.
- Once the data model has been navigated, the SQL query can be developed.

Navigation for Query 6



Query 6

- The navigation proceeds from Molecule to Reaction via the is consumed by association
- In the relational model this uses 3 tables: `Molecule`, `Reaction` and `Consumption`.
- The from clause is

```
from Molecule, Reaction, Consumption
```

- This presents a problem because now there are ambiguous column names: Both `Molecule` and `Reaction` have `id` columns.
- The solution is to use *table aliases*. The from clause is now

```
from Molecule m, Reaction r, Consumption c
```

© 2017 Ken Baclawski All Rights Reserved

Table Aliases

- Table aliases have many uses
  - Resolve ambiguous column names
  - Make queries more readable
- It is a good habit to use table aliases in all queries
  - Even if there is only one table
- The syntax has an optional form. For example, the from clause for Query 6 could be written like this:

```
from Molecule as m, Reaction as r, Consumption as c
```

Computing a Join

- Rows on the left side of the join are matched with rows on the right side of the join using the join condition
- The result of the join is a table with rows that are the concatenation of the fields of the left side with the fields of the right side.
- If the left side has M rows and the right side has N rows, then the number of rows in the join ranges from:
  - 0 rows if no pairs of rows match
  - MN rows if every pair of rows matches
- When every pair of rows match, the join is called the *cartesian product*

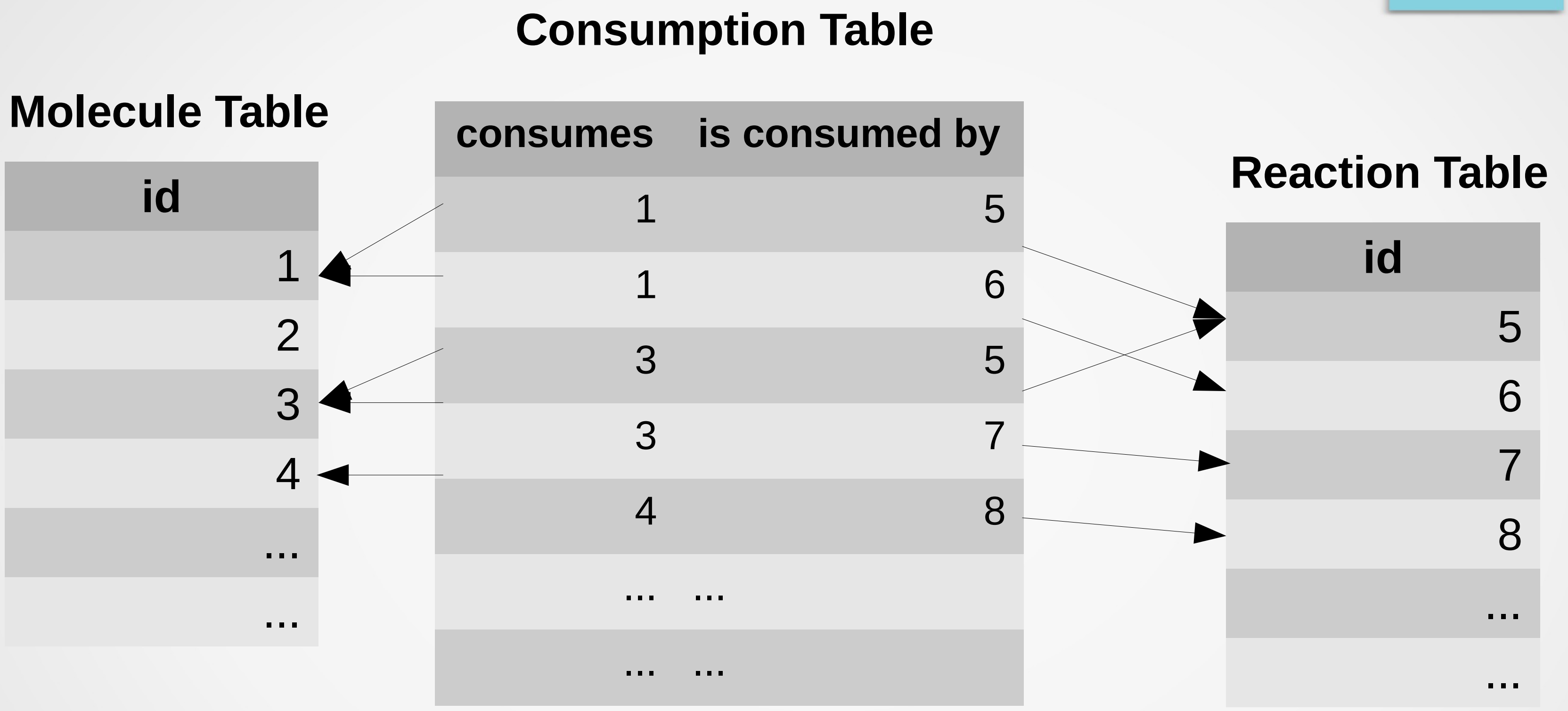
© 2017 Ken Baclawski All Rights Reserved

Query 6

- The 3 tables are connected to each other in the query using *join conditions*.
  - A join condition is a constraint in the where clause that involves attributes in two tables
  - An ordinary condition is a constraint in the where clause that involves attributes in one table
- In most cases, the join condition will be a foreign key constraint.
- For this query, remember how the association was translated to the relational model.

© 2017 Ken Baclawski All Rights Reserved

Translating many-to-many associations



Molecule Table

id
1
2
3
4
...
...

Consumption Table

consumes	is consumed by
1	5
1	6
3	5
3	7
4	8
...	...
...	...

Reaction Table

id
5
6
7
8
...
...

Molecule 1 is consumed by reactions 5 and 6
Molecule 3 is consumed by reactions 5 and 7
Molecule 4 is consumed by reaction 8

Reaction 5 consumes molecules 1 and 3
Reaction 6 consumes molecule 1
Reaction 7 consumes molecule 3
Reaction 8 consumes molecule 4

© 2017 Ken Baclawski All Rights Reserved

Solution to Query 6

- List all molecules consumed by a reaction named WFO
- The join conditions are:

  `m.id = c.consumes`

  `c.isConsumedBy = r.id`
- The ordinary condition is

  `r.name = 'WFO'`
- The select clause is

  `m.id`

Putting all these together with the from clause gives the whole query:

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

© 2017 Ken Baclawski All Rights Reserved

Join Syntax

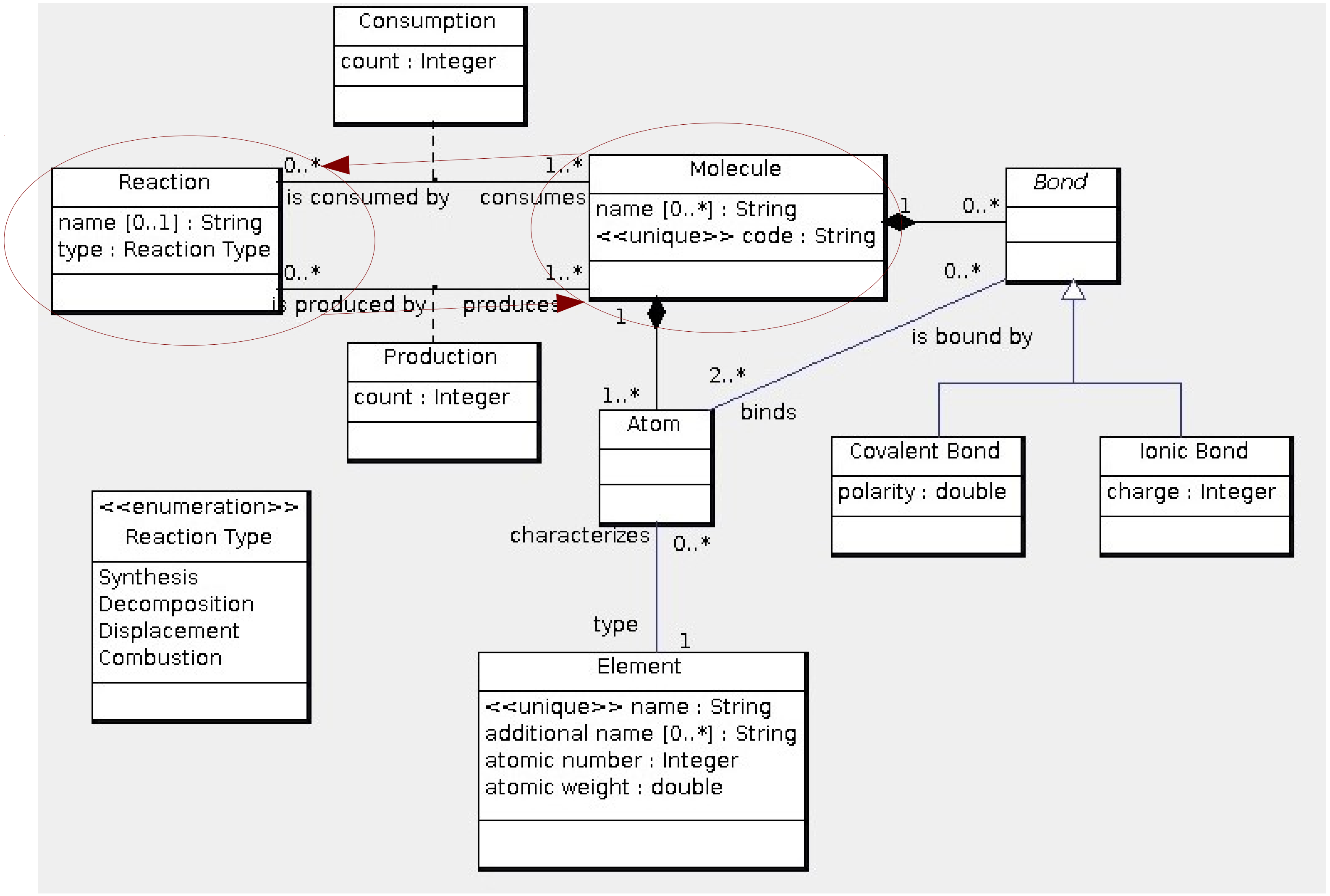
- Alternative syntax for specifying joins
- The joins and the join conditions are specified in the from clause rather than the where clause:

```
select m.id
  from Molecule m join Consumption c on (m.id = c.consumes)
       join Reaction r on (c.isConsumedBy = r.id)
 where r.name = 'WFO'
```

Query 7

- List the molecules by code that are consumed and produced by a WFO reaction
- “List ... by ...” means that every object should be listed, and for each object the specified attributes should be returned.
- Begin by navigating...

Navigation for Query 7



© 2017 Ken Baclawski All Rights Reserved

Query 7

- The navigation proceeds from Molecule to Reaction and back to Molecule via the is consumed by and produces associations
- The Molecule class occurs twice, playing two different roles in the query.
  - The first is the molecule that is consumed
  - The second is the molecule that is produced
- In the relational model this uses 4 tables: `Molecule`, `Reaction`, `Consumption` and `Production`.
- However, the from clause has 5 tables because the Molecule class plays two roles.
- The from clause is

```
from Molecule m1, Reaction r, Consumption c, Production p,
Molecule m2
```

- Table aliases are essential for this query because the same table occurs twice.

Query 7

- Using the foreign key constraints, the query is:

```
select m1.code, m2.code
  from Molecule m1, Reaction r,
       Consumption c,
       Production p, Molecule m2
 where m1.id = c.consumes
   and c.isConsumedBy = r.id
   and r.id = p.isProducedBy
   and p.produces = m2.id
   and r.name = 'WFO'
```

© 2017 Ken Baclawski All Rights Reserved

Query 8

- 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 similar to Query 7 except that all reactions are being queried.
- For each reaction, a molecule that is both produced and consumed (with the same counts) is a catalyst

© 2017 Ken Baclawski All Rights Reserved

Try to Solve Query 8

- This is the first try at the query
  - The constraint on `r.name` was dropped
  - The new constraints are in boldface

```
select m1.code
  from Molecule m1, Reaction r,
       Consumption c,
       Production p, Molecule m2
 where m1.id = c.consumes
   and c.isConsumedBy = r.id
   and r.id = p.isProducedBy
   and p.produces = m2.id
   and m1.id = m2.id
   and c.count = p.count
```

© 2017 Ken Baclawski All Rights Reserved

Incorrect Solution to Query 8

- This is the first try at the query
  - The constraint on `r.name` was dropped
  - The new constraints are in boldface

```
select m1.code
  from Molecule m1, Reaction r,
       Consumption c,
       Production p, Molecule m2
 where m1.id = c.consumes
   and c.isConsumedBy = r.id
   and r.id = p.isProducedBy
   and p.produces = m2.id
   and m1.id = m2.id
   and c.count = p.count
```

© 2017 Ken Baclawski All Rights Reserved

Query 8

- However, this query is not correct
- A molecule could be a catalyst for several reactions, and each catalyst should be listed once
- The best way to deal with this situation is to use nested queries.
- However, another way is to use `select distinct`
- This removes any duplicates in the returned values.
- Warning: The `select distinct` statement solves this query because the `code` column is not null and unique. If it allowed null or was not unique, then only a nested query will solve this query.

© 2017 Ken Baclawski All Rights Reserved

Solution to Query 8

- A correct query is:

```
select distinct m1.code
  from Molecule m1, Reaction r,
       Consumption c,
       Production p, Molecule m2
 where m1.id = c.consumes
   and c.isConsumedBy = r.id
   and r.id = p.isProducedBy
   and p.produces = m2.id
   and m1.id = m2.id
   and c.count = p.count
```

© 2017 Ken Baclawski All Rights Reserved

The select distinct operation

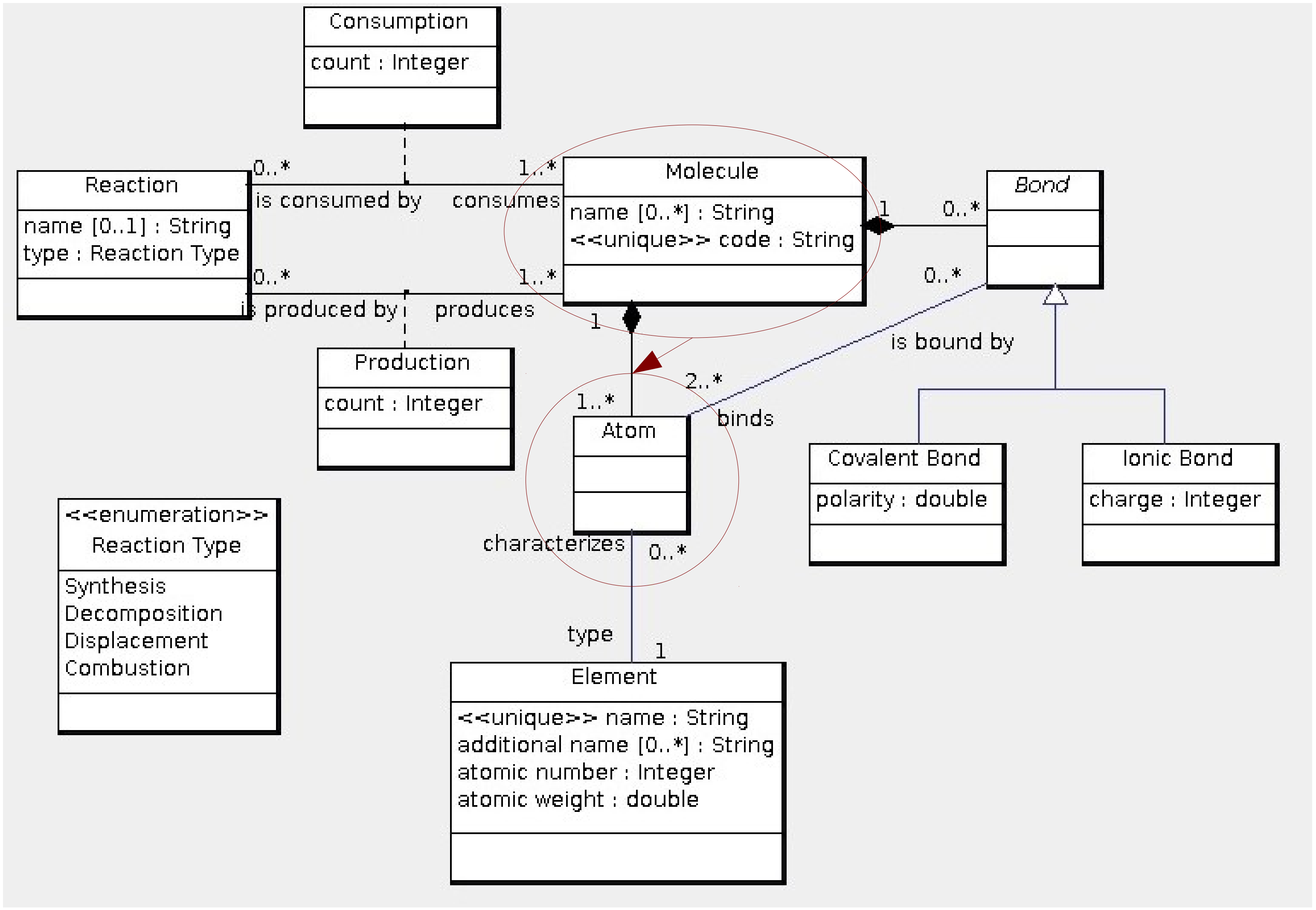
- Elimination of duplicates
- The operation is relatively costly
- For a large set of results elimination of duplicates can require more time than the rest of the query combined
- Eliminate duplicates only if it is essential
- The nested query technique to be introduced later is a much more efficient way to deal with this problem
  - It produces the correct results without ever creating any duplicates

© 2017 Ken Baclawski All Rights Reserved

Query 9

- Show the code and number of atoms for every Molecule
- This requires more than just extracting rows from tables. One must also compute the number of atoms
- This is an *aggregate function*
  - Unrelated to the UML notion of aggregation
  - Aggregate functions include: sum, count, average, max, min
- Aggregate functions are computed on groups of rows
  - The process is called *grouping*

Navigation for Query 9



Solution to Query 9

The first step is to find the molecules and their atoms.

```
select m.code, a.id
  from Molecule m, Atom a
 where a.partOf = m.id
```

The next step is to partition the rows into groups such that each group has all of the atoms of one atom

```
select m.code
  from Molecule m, Atom a
 where a.partOf = m.id
 group by m.id
```

The last step is to apply the aggregate function to each group in the partition

```
select m.code, count(*)
  from Molecule m, Atom a
 where a.partOf = m.id
 group by m.id
```

The `count(*)` aggregate function counts the number of *records* in each group

Clause Order

- The clauses are logically run in this order:

from	retrieve tables
where	select rows from the tables
group by	group the selected rows
select	compute functions and return

- Although the select clause appears first in a query, it is logically the last one to be computed

© 2017 Ken Baclawski All Rights Reserved

Computing the join

Atom Table

id	type	partOf
1	1	1
2	1	1
3	2	1
4	1	2
5	1	2
6	2	2
7	2	2
...	...	...

Molecule Table

id	code
1	H2O
2	H2O2
...	...

The result of the join

a.id	a.type	a.partOf	m.id	m.code
1	1	1	1	H2O
2	1	1	1	H2O
3	2	1	1	H2O
4	1	2	2	H2O2
5	1	2	2	H2O2
6	2	2	2	H2O2
7	2	2	2	H2O2
...	...	...	...	...
...	...	...	...	...

Grouping

a.id	a.type	a.partOf	m.id	m.code
1	1	1	1	H2O
2	1	1	1	H2O
3	2	1	1	H2O
4	1	2	2	H2O2
5	1	2	2	H2O2
6	2	2	2	H2O2
7	2	2	2	H2O2
...	...	...	...	...
...	...	...	...	...

```
m.id=1

m.id=2

...
```

© 2017 Ken Baclawski All Rights Reserved

Apply Aggregate Function

count(*)	m.code
3	$H2O$
4	$H2O2$
...	...

© 2017 Ken Baclawski All Rights Reserved

Query 10

- Show the code and number of atoms for every Molecule that has at least 4 atoms
- This is the same as Query 9 except for the added requirement
- The added requirement is not meaningful for the where clause because `count(*)` is computed *after* grouping
- The having clause is similar to the where clause except that it constrains which aggregated rows are selected.

```
select m.code, count(*)
  from Molecule m, Atom a
 where a.partOf = m.id
 group by m.id
having count(*) >= 4
```

© 2017 Ken Baclawski All Rights Reserved

Clause Order

- The clauses are logically run in this order:

from	retrieve tables
where	select rows from the tables
group by	group the selected rows
having	select groups from the grouping
select	compute functions and return

- Although the select clause appears first in a query, it is logically the last one to be computed

© 2017 Ken Baclawski All Rights Reserved

Query 11

- Show the code and number of atoms in order by the number of atoms for every Molecule that has at least 4 atoms
- This is the same as Query 10 except for the ordering of results
- The order by clause sorts the results by one or more output columns
  - If two results have the same first column, then the next column is compared, and so on.
  - This is called lexicographic (dictionary) order
  - The default is ascending (asc) order
  - One can also specify descending (desc) order

```
select m.code, count(*)
  from Molecule m, Atom a
 where a.partOf = m.id
 group by m.id
having count(*) >= 4
 order by count(*)
```

Clause Order

- The clauses are logically run in this order:

from	retrieve tables
where	select rows from the tables
group by	group the selected rows
having	select groups from the grouping
order by	sort by specified columns
select	compute functions and return

- Although the select clause appears first in the query, it is logically the last one to be computed

© 2017 Ken Baclawski All Rights Reserved

Output Column Aliases

- Output columns can also be given aliases
- The syntax is the same as table aliases
- An output column alias is a *synonym* for the output column expression, it is not a variable that has been assigned to the value of the output column
- The output column aliases can be used in those clauses where the output column is meaningful

```
select m.code mcode, count(*) atomct
  from Molecule m, Atom a
 where a.partOf = m.id
 group by m.id
having atomct >= 4
 order by atomct
```

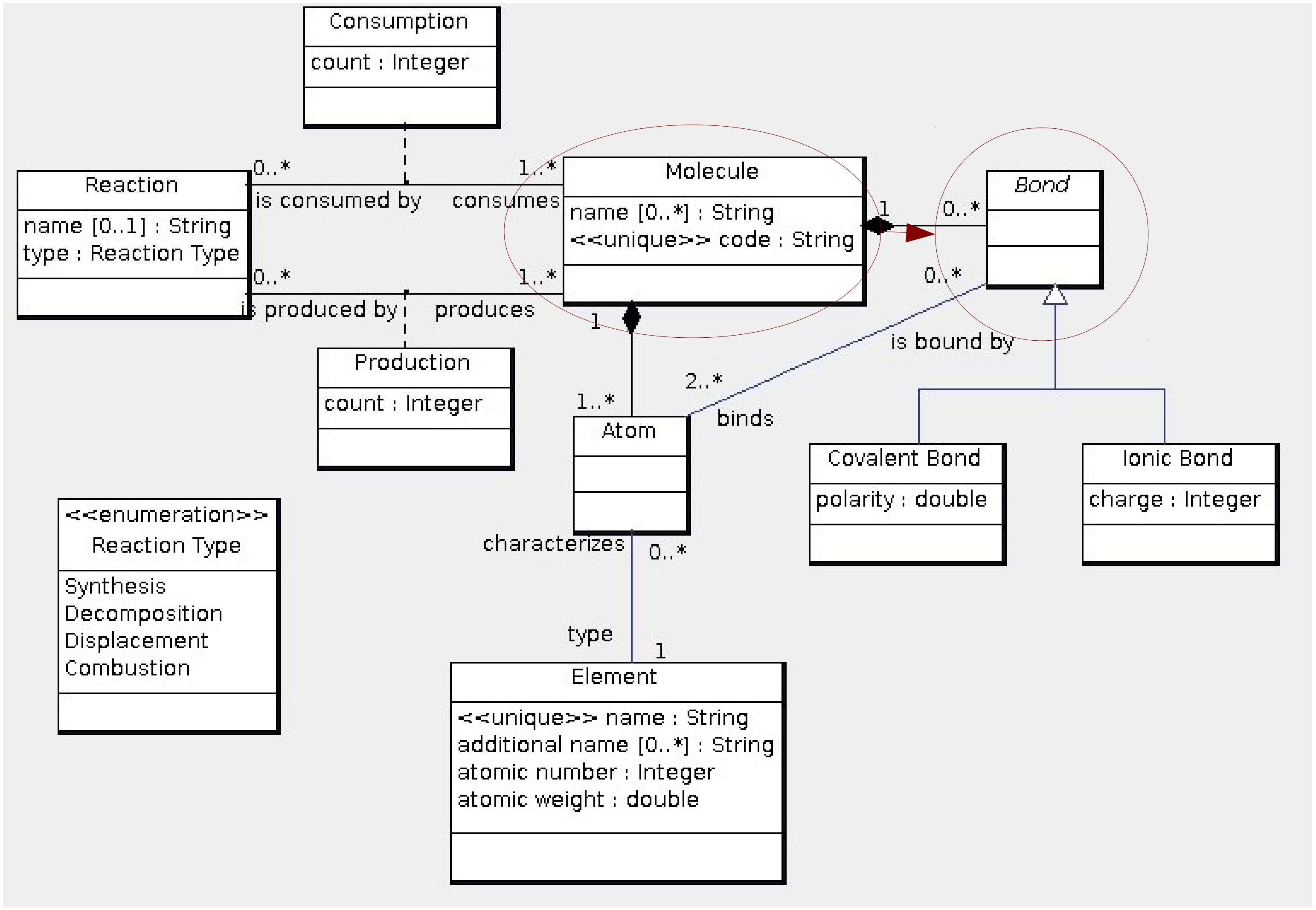
© 2017 Ken Baclawski All Rights Reserved

Query 12

- Show the code and number of bonds for every molecule
- Begin with the navigation...

© 2017 Ken Baclawski All Rights Reserved

Navigation for Query 12



Query 12

- Show the code and number of bonds for every molecule
- This seems to be very similar to Query 9, but it is possible for a molecule to have no bonds at all.
- If a molecule has no bonds, then the join will have no row for that molecule so there will not be a group.
- An *outer join* can be used to solve this problem.

© 2017 Ken Baclawski All Rights Reserved

Inner and Outer Joins

- A normal join is an *inner join*
  - Rows on the left side of the join are matched with rows on the right side of the join using the join condition
  - The result of the join consists of rows that are the concatenation of the fields of the left side with the fields of the right side.
- *Left outer join*
  - In addition to the inner join rows, the rows on the left side that do not match any row on the right side are also included
  - The unmatched left side rows are concatenated with right side rows that have NULL in every field.
- *Right outer join*
  - Same as left outer join but the unmatched right rows are the ones that are concatenated with left side rows that have NULL in every field
- *Full outer join*
  - Both left and right outer join

© 2017 Ken Baclawski All Rights Reserved

Query 12

Show the code and number of bonds of every molecule

This is the outer join:

```
select m.code
  from Molecule m left join Bond b
    on (m.id = b.partOf)
```

This groups by molecule:

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

© 2017 Ken Baclawski All Rights Reserved

Solution to Query 12

Show the code and number of bonds of every molecule

Finally, compute the number of bonds:

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

The query does not use `count(*)` because `count(*)` counts the number of records, not the number of bonds.

- `count(*)` is 1 for a molecule with no bonds
- `count(b.id)` is 0 for a molecule with no bonds because `b.id` is null for such a molecule and null is not a value.

Query 13

- Show the code and number of bonds for every molecule, but if a molecule has no bonds, then show the code and “unbonded”
- This is the same as Query 12 except that for a molecule with no bonds, the number 0 is to be replaced by the string “unbonded”.
- SQL has many functions that can be used in a select statement to compute the return values. For Query 13, one can use an `if` expression in MySQL (`iif` in SQL Server):

```
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
```

© 2017 Ken Baclawski All Rights Reserved

SQL Functions

- One can usually expect a DBMS to support many functions such as numeric functions and operators, string functions, date/time functions, and conversions between different datatypes
  - Unfortunately, the details vary so there is little portability
- If you need portability
  - Only return column values or aggregate function values
  - Perform any computations in a programming language

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.

Query Problems

List the valleys by name whose area exceeds 600.

List the hills by name that are part of a hill whose summit area exceeds 600.

© 2017 Ken Baclawski All Rights Reserved

Assignment #3