Notes for Database Course

Jing LI
pkuyplijing@gmail.com

June 24, 2016

Sincere gratitude to Professor Jennifer Widom
for offering such a wonderful class.

Contents

Chapter 1

Introduction and Relational Databases

1.1 Introduction to DBMS

A **Database Management System(DBMS)** provides an efficient, reliable, convenient and safe multi-user storage of and access to massive amounts of persistent data.

Massive Databases handle data at a massive scale. Databases are handling TBs of data, sometimes every day. The size of data managed by a DBMS is much larger than what can fit in the memory of a typical computing system, thus they are designed to handle data that reside out of memory.

Persistent Data handled by a DBMS outlives programs that use it. It is not uncommon that multiple programs operate on the same data.

Safe DBMS handles data used by critical applications. It has to be guaranteed that the data stays in consistent state in case of hardware failures and software failures.

Multi-user Concurrency control mechanism ensures that the data stays consistent when the database is accessed by multi-users. The control occurs at the level of data items rather than the database as a whole, which avoids too much sacrifice of performance.

Convenient DBMS makes it easy to make powerful and interesting operations on data. The concept of **physical data independence** ensures that the way the data is stored and laid out on the disk is independent of the way that programs think of it. Databases are queried by high level query languages that are **declarative**: user describes what data is needed without specifying the algorithm to get the data, and the system itself will get the data efficiently with appropriate algorithms.

Efficiency Databases are supposed to carry out thousands of query/update operations per second. Performance is one of the major concerns of users.

Reliability DBMS ought to guarantee 99.99999% up time for its critical applications.

The following key concepts will be covered in this course:

Data model A description of how the data is structured. For example, relational database, in which data is thought of as a set of records. Other popular data models include XML, JSON, graph model, etc.

Schema v.s. data Their relationship is like the relationship of types and variables in a programing language. Schema is usually set up at the beginning and does not change very often, whereas data can change frequently.

Data definition language(DDL) Used to set up the schema.

Data manipulation language(DML), or query language Used to query and modify data.

A few roles will be involved:

- DBMS implementer
- Database designer
- Database application developer
- Database administrator

Most of the time we will think of databases from the perspective of designers and developers.

1.2 The Relational Model

The relation model is the data model used by all major commercial database systems. It is a simple model that allows query with simple, yet expressive high level languages. There exist efficient implementations of the relational model and the query languages it use.

In relational model, a database consists of a set of **relations**, or **tables**, each of which has a name. Each relation features a set of named **attributes**, or **columns**. Typically each attribute has a **type**, or **domain**. The domain can be atomic, e.g. integers, doubles, enums, strings, etc, but databases can also support structured types. The actual data is stored in **tuples**, or **rows** in the tables. Each tuple has a value for each attribute. **NULL** is a special value for unknown or undefined state of any type.

The **schema** is a structural description of relations in database, including the name, the attributes and their types, while the **instance** is the actual contents

of a table at a given time. Typically the schema is set in advance, whereas the instance changes over time.

A **key** is an attribute(or a set of attributes) whose value is (or whose combination of values are) unique in each tuple of a relation. Key is used to identify a specific tuple so that a user can find it by asking for its key. Database systems tend to build special index structures or store the database in a particular way so that finding a tuple according to its key is efficient. Moreover, when one relation wants to refer to a tuple in another relation, it has to use its unique key since their is no notion of pointers in relational databases.

Creating tables in SQL is simple:

```
1 Create Table Student(ID, name, GPA, photo);
2
3 --Provide types of attributes
4 Create Table College (name string, state char(2), enrollment
    integer);
```

1.3 Querying Relational Databases

General steps in creating and using a relational database:

1. Design the schema and create it using DDL.
2. "Bulk load" initial data into the DB.
3. Repeat: execute queries and modifications.

Relational DB supports ad-hoc queries in high-level language. A few examples of possible queries:

- All students with GPA>3.7 applying to Stanford and MIT only.
- All engineering departments in CA with <500 applicants.
- College with highest average accept rate over last 5 years.

Some queries are easy to pose while others are difficult. Some are easy for DBMS to execute efficiently while others are harder. Note that the difficulty in posing a query is not correlated with the difficulty in executing it.

All queries return relations, i.e. the language is **closed**. This nice feature allows pipelining query results: a new query can be run on the result of a previous query.

We are going to cover two kinds of query languages. **Relational algebra** is a formal language, while **SQL** is an actual/implemented language. Relational algebra is the foundation of SQL. It helps to define the semantics of SQL. As a taste for both languages, we will write the following query with them:

IDs of students with GPA>3.7 applying for Stanford.

In relational algebra, the query is written as

$$\Pi_{\texttt{ID}}\sigma_{\texttt{GPA}>3.7\wedge\texttt{cName}=\text{“}\texttt{Stanford}\text{”}}\,(\texttt{Student}\bowtie\texttt{Apply})$$

The same query is written as

```
1 Select Student.ID
2 From Student, Apply
3 Where Student.ID=Apply.ID And GPA>3.7 And college='Stanford'
```

in SQL.

Chapter 2

Querying Relational Database

A **query** on a set of relations produces a relation as result. We will use a simple college admissions database as example in this chapter. It contains three relations:

```
College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS[1])
Apply(sID, cName, major, decision)
```

The key in each relation is underlined.

2.1 Relational Algebra

2.1.1 Relation Name Query

The simplest query is a relation name. For example, the query

`Student`

will return a copy of the `Student` relation. In more complex queries, we will use **operators** to filter, slice and combine relations.

2.1.2 Select Operator

The **Select** operator is used to pick out certain rows. It is written as a σ with a subscript being the condition of the selection followed by the relation being queried. For example:

- Students with GPA>3.7:

$$\sigma_{\texttt{GPA}>3.7}\texttt{Student}$$

[1] sizeHS means size of the high school the student attended.

- Students with GPA>3.7 and sizeHS<1000:

$$\sigma_{\texttt{GPA}>3.7 \,\wedge\, \texttt{sizeHS}<1000}\texttt{Student}$$

- Applications to Stanford CS major:

$$\sigma_{\texttt{cName}=\texttt{'Stanford'} \,\wedge\, \texttt{major}=\texttt{'CS'}}\texttt{Apply}$$

2.1.3 Project Operator

The **Project** operator picks certain columns. It is written as a Π with a subscript being the columns to be picked followed by the relation being queried. For example:

- ID and decision of all applications:

$$\Pi_{\texttt{sID,decision}}\texttt{Apply}$$

If we want to pick both rows and columns, we can simply combine select and project. For example:

- ID and name of students with GPA>3.7:

$$\Pi_{\texttt{sID,sName}}\left(\sigma_{\texttt{GPA}>3.7}\texttt{Student}\right)$$

In relation algebra, duplicates in a query result are always eliminated. It is guaranteed that the same tuple does not appear more than once. This is different from SQL, in which duplicates are not eliminated. The low-level reason is that relational algebra is based on sets while SQL is based on multi-sets.

2.1.4 Cross-product Operator

The **Cross-product, a.k.a the Cartesian product** operator combines two relations. The schema of the new relation is the combination of the schemata of the two old relations. If they share the same attribute name, the attribute will be prefixed with the relation name. The number of tuples in the new relation will be the product of the numbers of tuples in the two relations. For example, the cross-product of `Student` and `Apply` will have 8 attributes, including `Student.sID` and `Apply.sID`. If there are `s` tuples in `Student` and `a` tuples in `Apply`, there will be `sa` tuples in `Student`×`Apply`.

Cross-product seems not that interesting, but it is useful when combined with other operators. For example:

- Names and GPAs of students with HS>1000 who applied to CS and were rejected:

$$\Pi_{\texttt{sName,GPA}}\left(\sigma_{\substack{\texttt{Student.sID=Apply.sID} \,\wedge\, \texttt{sizeHS}>1000 \\ \wedge\, \texttt{major}=\texttt{'CS'} \,\wedge\, \texttt{decision}=\texttt{'rejected'}}}\left(\texttt{Student} \times \texttt{Apply}\right)\right)$$

2.1.5 Natural Join Operator

The **Natural Join** operator performs a cross-product, enforces select for equality on all attributes with the same name, and eliminates one copy of duplicate attributes. By convention it is written as $\bowtie$ (pronounced as "bowtie"). We use the same example to illustrate its use:

- Names and GPAs of students with HS>1000 who applied to CS and were rejected:

$$\Pi_{\texttt{sName,GPA}} \left(\sigma_{\texttt{sizeHS>1000} \wedge \texttt{major='CS'} \wedge \texttt{decision='rejected'}} \left(\texttt{Student} \bowtie \texttt{Apply}\right)\right)$$

Obviously the expression is more elegant and compact. Consider a more complicated example:

- Names and GPAs of students with HS>1000 who applied to CS at college with enrollment>20000 and were rejected:

$$\Pi_{\texttt{sName,GPA}} \left(\begin{array}{l} \sigma_{\texttt{enrollment>20000} \wedge \texttt{sizeHS>1000}} \\ \wedge \texttt{major='CS'} \wedge \texttt{decision='rejected'} \end{array} \left(\texttt{Student} \bowtie \texttt{Apply} \bowtie \texttt{College}\right) \right)$$

Actually all we have to do is to add the `College` relation in the natural join, and add a select condition for the enrollment.

Note that natural join does not add any expressive power to relational algebra. It can always be substituted with a project composed with a select imposed on the cross-product.

2.1.6 Theta Join Operator

The **Theta Join** operator is equivalent to applying a select on the result of a cross-product. It's written as $\bowtie_\theta$, in which θ is the condition for the select. From the definition we have

$$\texttt{Exp}_1 \bowtie_\theta \texttt{Exp}_2 = \sigma_\theta(\texttt{Exp}_1 \times \texttt{Exp}_2)$$

Like natural join, theta join does not add any extra expressive power to relational algebra, but rather facilitates the notation. It is implemented as a basic operation in most DBMSs. In the terminology of DB management, the term "join" usually means theta join.

2.1.7 Union Operator

Union does the standard set union in set theory. It is also written as $\cup$, the standard union operator in set theory. For example:

- List of college and student names:

$$\Pi_{\texttt{cName}}\texttt{College} \cup \Pi_{\texttt{sName}}\texttt{Student}$$

Note that this result cannot be obtained from combination of previous operators: college names and student names are listed in one column rather than two different columns. The two expressions on the two sides of the union operator ought to have the same schema, which is not the case here. This will be clarified later.

2.1.8 Difference Operator

The **Difference** operator is written as the minus sign -. As an example:

- IDs of students who didn't apply anywhere:

$$\Pi_{\texttt{sID}}\texttt{Student} - \Pi_{\texttt{sID}}\texttt{Apply}$$

- IDs and names of students who didn't apply anywhere:

$$\Pi_{\texttt{sID,sName}}\left((\Pi_{\texttt{sID}}\texttt{Student} - \Pi_{\texttt{sID}}\texttt{Apply}) \bowtie \texttt{Student}\right)$$

Note the use of natural join in the second example. This trick is called **join back**.

2.1.9 Intersection Operator

The **Intersection** operator is written as $\cap$. For example:

- Names that are both a college name and a student name:

$$\Pi_{\texttt{cName}}\texttt{College} \cap \Pi_{\texttt{sName}}\texttt{Student}$$

We face the same problem of schema contradiction as with union operator. Also note that intersection operator does not add expressive power. Actually we have $\texttt{E}_1 \cap \texttt{E}_2 \equiv \texttt{E}_1 - (\texttt{E}_1 - \texttt{E}_2)$. From another perspective, since two expressions between which the intersection operator is used must have the same schema, we actually have $\texttt{E}_1 \cap \texttt{E}_2 \equiv \texttt{E}_1 \bowtie \texttt{E}_2$ in this case.

2.1.10 Rename Operator

The **Rename** operator is written as $\rho_{\texttt{R}(\texttt{A}_1,\ldots,\texttt{A}_n)}(\texttt{E})$, which means assign the schema $\texttt{R}(\texttt{A}_1, \ldots, \texttt{A}_n)$ to the result of $\texttt{E}$. If only the relation name is to be changed, it can be abbreviated as $\rho_{\texttt{R}}(\texttt{E})$. Similarly, if only the attribute names will be changed, it can be written as $\rho_{\texttt{A}_1,\ldots,\texttt{A}_n}(\texttt{E})$. For the second abbreviation there has to be a list of attributes, otherwise it cannot be distinguished from the first abbreviation.

Rename operator can be used to unify schemata for the set operators (union, difference and intersection). For example:

- List of college and student names:

$$\rho_{\texttt{C(name)}}\left(\Pi_{\texttt{cName}}\texttt{College}\right) \cup \rho_{\texttt{C(name)}}\left(\Pi_{\texttt{sName}}\texttt{Student}\right)$$

It can also be used for disambiguation in "self-joins". For example:

- Pairs of colleges in the same state:

$$\Pi_{\texttt{n1,n2}}\left(\sigma_{\texttt{s1=s2}\,\wedge\,\texttt{n1<n2}}\left(\rho_{\texttt{C1(n1,s1,e1)}}(\texttt{College})\times\rho_{\texttt{C2(n2,s2,e2)}}(\texttt{College})\right)\right)$$

A simpler solution to this problem is:

$$\Pi_{\texttt{n1,n2}}\left(\sigma_{\texttt{n1<n2}}\left(\rho_{\texttt{C1(n1,s,e1)}}(\texttt{College})\bowtie\rho_{\texttt{C2(n2,s,e2)}}(\texttt{College})\right)\right) \tag{2.1}$$

Note that $n1 < n2$ is to avoid repetitions of the same pair. $n1 \neq n2$ is enough to avoid self-pair, but the same pair will appear twice in reverse order.

2.1.11 Alternative Notations

Assignment Statements

Assignment Statements can help modularize expressions. The expression (2.1) can be rewritten as

$$\begin{aligned}
\texttt{C1} &\coloneqq \rho_{\texttt{n1,s,e1}}(\texttt{College})\\
\texttt{C2} &\coloneqq \rho_{\texttt{n2,s,e2}}(\texttt{College})\\
\texttt{CP} &\coloneqq \texttt{C1}\bowtie\texttt{C2}\\
\texttt{Ans} &\coloneqq \Pi_{\texttt{n1,n2}}(\sigma_{\texttt{n1<n2}}\texttt{CP})
\end{aligned}$$

Expression Trees

Expression trees help visualize the structure of an expression. Suppose we want to get the GPA of students applying to CS in CA, we can write the expression:

$$\Pi_{\texttt{GPA}}(\sigma_{\texttt{major='CS'}\,\wedge\,\texttt{state='CA'}}(\texttt{Student}\bowtie\texttt{College}\bowtie\texttt{College}))$$

This can be written as a tree with the three relations as leaves, the σ as their root and the Π as the root of the σ.

As a summary, relational algebra is a formal language that serves as the foundation of all implemented relational DBMSs. Core operators in relational algebra include relation, select, project, cross product, union, difference and rename. Other operators are useful abbreviations that do not add expressive power: natural join, theta join and intersection.

2.2 SQL

SQL query language, based on relational algebra, is supported by all major commercial database systems. It's a standardized language that has been evolving over time with a lot of new features incorporated, while the essence of the language is still relatively simple. It can be used either interactively via GUI or prompt, and it can also be embedded in programs. SQL is a declarative

language, which means that user has to describe only what data is needed, not how to get it. The declarative nature of SQL makes the component named **query optimizer** extremely important. SQL includes two parts:

Data Definition Language(DML) `create table, drop table, etc.`

Data Manipulation Language(DDL) `select, insert, delete, update, etc.`

2.2.1 `select` Statement

The basic `select` statement has the form

```
select A1,A2,...,An
from R1,R2,...,Rm
where condition
```

The `from` clause specifies the relations to query from, the `where` clause is used to combine and query the relations, and the `select` clause sets what to return. It is actually equivalent to the relational algebra expression

$$\Pi_{\texttt{A1},\ldots,\texttt{An}}\sigma_{\texttt{condition}}(\texttt{R1} \times \texttt{R2} \times \texttt{Rm})$$

The result of a select statement is a relation whose schema is defined by the `select` clause.

The attributes used in the condition does not necessarily appear in the result.

```
select sID, sName from Student where GPA>3.6;
```

What is done by the natural join operator in relational algebra has to be specified explicitly in SQL.

```
select sName, major
from Student, Apply
where Student.sID = Apply.sID;
```

If a few relations share the same attribute, the attribute to which it belong has to be specified to avoid ambiguous error.

```
select College.cName --error without College
from College, Apply
where College.cName = Apply.cName and enrollment>20000 and major =
    'CS';
```

`distinct` keyword

SQL selection result contains duplicate values since it is based on a multi-set model, different from relational algebra which is based on a set model. Nonetheless, duplicate values can be avoided by adding the `distinct` keyword.

```
select distinct sName, major
from Student, Apply
where Student.sID = Apply.sID;
```

`order by` clause

SQL is an unordered model. The same select statement can result in values in different orders when run multiple times. If the result values are supposed to be ordered according to a certain attribute, an `order by` clause needs to be added. The default behavior is to order in ascending order. If descending order is expected, it needs to be specified explicitly at the end of the clause. If further ordering according to other attributes is expected for rows that have the same value for the attribute specified, they should be listed afterwards in the clause.

```
select Student.sID, sName, GPA, Apply.cName, enrollment
From Student, College, Apply
where Student.sID = Apply.sID and College.cName = Apply.cName
order by GPA desc, enrollment;
```

`like` keyword

The `like` predicate can be used to do string pattern matching for attributes in the `where` clause.

```
select sID, major
from Apply
where major like '%bio%';
```

`select *`

If all attributes in the relations are needed, instead of listing them explicitly, we can use `select *` that represents selecting all attributes.

```
select *
from Apply
where major like '%bio%';
```

`as` keyword

Arithmetics can be used in SQL clauses. Attribute names can be changed by the `as` keyword.

```
select sID, sName, GPA, GPA*(sizeHS/1000.0) as scaledGPA
from Student;
```

Table Variables

Relations used in a select statement can be given another name using table variables.

```
select S.sID, sName, GPA, A.cName, enrollment
from Student S, Apply A, College C
where A.sID = S.sID and C.cName = S.cName;
```

Table variables are especially useful when the same relation is used multiple times in the statement. The following statement selects all pairs of students with the same GPA.

```
1 select S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA
2 from Student S1, Student S2
3 where S1.GPA = S2.GPA and S1.sID < S2.sID;
```

Set Operators: `union`, `intersect` and `except`

We can use the union operator to get the union of student names and college names.

```
1 select sName as name from Student
2 union all
3 select cName as name from College
4 order by name;
```

SQL allows the `union` even without the `as name`. It will just use `sName` or `cName` as the attribute name in the result. The `all` keyword tells SQL to preserve all duplicate values in the result. By default duplicates are eliminated in the result of `union`. The `order by` clause sorts the result as before.

We can use `intersect` to select students who applied both CS and EE.

```
1 select sID from Apply where major = 'CS'
2 intersect
3 select sID from Apply where major = 'EE';
```

Some database systems do not support `intersect`. They don't lose any expressive power because the expression below is equivalent to using `intersect`:

```
1 select distinct A1.sID
2 from Apply A1, Apply A2
3 where A1.sID = A2.sID and A1.major = 'CS' and A2.major = 'EE';
```

However this does not go for `except`. The following expression

```
1 select sID from Apply where major = 'CS'
2 except
3 select sID from Apply where major = 'EE';
```

cannot be expressed by the constructs we have introduced up to now. Additional constructs are needed not to lose this expressive power.

2.2.2 Subqueries in Clauses

Subqueries are `select` statements nested inside clauses of another `select` statement. The following statement select IDs and names of all students who apply for CS.

```
select sID, sName
from Student
where sID in (select sID from Apply where major = 'CS');
```

This is actually equivalent to

```
select distinct Student.sID, sName
from Student, Apply
where Student.sID = Apply.sID and major = 'CS';
```

Extra Expressive Power

However, `distinct` does not always bring such an equivalent. Consider

```
select GPA
from Student
where sID in (select sID from Apply where major = 'CS');
```

The version without sub-queries

```
select GPA
from Student, Apply
where Student.sID = Apply.sID and major = 'CS';
```

cannot provide the same result, with or without `distinct`. The reason is that two kinds of duplicates are involved in this example:

- GPA of the same student applying for CS multiple times at different colleges.
- Different students applying for CS who have the same GPA.

The statement using a sub-query preserves the second kind and eliminates the first one, which is plausible for calculating the average GPA of students who apply for CS. However, the version without sub-query either preserves both kinds (without `distinct`), or eliminates both(with `distinct`). This example proves that sub-query does add expressive power to SQL.

Besides facilitating the processing of duplicates, sub-queries also makes it possible to express `except` without using it. The example of `except` in the previous section is equivalent to

```
select sID
from Student
where sID in (select sID from Apply where major = 'CS')
  and sID not in (select sID from Apply where major = 'EE');
```

Note that `sID not in ...` is equivalent to `not sID in ...`.

`exists` keyword

```
--Colleges such that some other college is in the same state
select cName, state
from College C1
where exists (select * from College C2
              where C2.state = C1.state and C2.cName <> C1.cName);
--College with the largest enrollment
select cName
from College C1
where not exists (select * from College C2
                  where C2.enrollment > C1.enrollment);
```

`all` keyword

```
--students with highest GPA
select sName, GPA
from Student
where GPA >= all (select GPA from Student);
```

`any` keyword

```
--students not coming from the smallest high school
select sID, sName, sizeHS
from Student
where sizeHS > any (select sizeHS from Student);
```

Some database systems do not support `any` and `all`. They do not lose expressive power.

Sub-queries in `select` and `from` Clauses

```
--students whose GPAs are modified by more that 1.0 after scaling
select sID, sName, GPA, GPA*sizeHS/1000 as scaledGPA
from Student
where GPA*sizeHS/1000 - GPA > 1.0 or GPA - GPA*sizeHS/1000 > 1.0;
```

The statement above can be simplified as

```
select *
from (select sID, sName, GPA, GPA*sizeHS/1000 as scaledGPA
from Student) G
where abs(scaledGPA - GPA) > 1.0;
```

`abs()` is the built-in absolute value function. By putting a sub-query in the `from` clause, we are able to avoid calculating the scaled GPA multiple times.

```
--Pair colleges with their applicants having the highest GPA.
select distinct College.cName, state, GPA,
from College, Apply, Student
where College.cName = Apply.cName
  and Apply.sID = Student.sID
  and GPA >= all
    (select GPA from Student, Apply
     where Student.sID = Apply.sID
       and Apply.cName = College.cName);
```

The statement above can be rewritten as

```
select cName, state,
(select distinct GPA
 from Student, Apply
 where Student.sID = Apply.sID
   and College.cName = Apply.cName
   and GPA > all
   (select GPA from Student, Apply
    where Student.sID = Apply.sID
      and College.cName = Apply.cName)) as GPA
from College;
```

with a sub-query in the `select` clause. In such case it has to be guaranteed that the sub-query returns exactly one value (in this example the highest GPA among applicants for a particular college), because it will be used as an attribute of the returned relation.

2.2.3 `join` Family of Operators

We've been using commas to separate relations in the `from` clause, which results in cross product of the relations. Actually we can use `join` operators to gain explicit join of relations, saving the trouble of adding conditions in the `where` clause. They do not add expressive power to SQL, but can be quite useful for formulating queries, especially the outer join, which is hard to express without the outer join operator itself.

Inner Join on Condition

```
select distinct sID, sName, major
from Student, Apply
where Student.sID = Apply.sID; --natural join!
```

The **inner join on condition** operator is equivalent to the theta join operator in relational algebra. In the statement above, the condition actually results in a natural join of the two tables. This query can be formulated using the inner join on condition.

```
select distinct sID, sName, major
from Student inner join Apply
on Student.sID = Apply.sID;
```

`inner join` is the default join operator in SQL. Thus `inner` can actually be omitted.

Inner join on condition is actually equivalent to a simple select statement in terms of its meaning. It often serves as a hint for the DBMS to choose the most efficient execution. Conditions concerning the combination of the tables should be put in the `on` clause. Take the following statement as an example.

```
1 select sName, GPA
2 from Student join Apply
3 on Student.sID = Apply.sID
4 where sizeHS > 1000 and major = 'CS' and cName = 'Stanford';
```

Natural Join

The first example above can be simplified with a natural join:

```
1 select distinct sID, sName, major
2 from Student natural join Apply;
```

Note that we do not have to prefix `sID` with a table name in this example, because natural join eliminates duplicate attribute names in the tables automatically.

Inner Join using Attributes

There might exist attributes with the same name actually meaning different things, which is especially possible when the tables have a lot of attributes. So it is generally considered a better practice than natural join to use **inner join using attributes**, which explicitly lists the attributes that should be joined, as shown below.

```
1 select distinct sID, sName, major
2 from Student join Apply using (sID);
```

Most SQL implementations do not allow `using` and `on` to appear together. Thus conditions that cannot be put in `using` should be left for the `where` clause.

Outer Join

The following statement extract some information about the students and their applications.

```
1 select sID, sName, cName, major
2 from Student join Apply using(sID);
```

There are some students who haven't filed any applications, which is a typical use case for **outer** join:

```
1 select sID, sName, cName, major
2 from Student left outer join Apply using(sID);
```

Students who haven't filed any applications and thus do not have any related tuples in the `Apply` table will appear in the result, with `cName` and `major` filled with `null` value for them. When `left, right` or `full` appears before a `join`, the word `outer` can be omitted. Outer join can also be combined with natural join, returning the same result:

```
select sID, sName, cName, major
from Student natural left join Apply;
```

Right outer join is obviously the sibling operator of left outer join: tuples that only appear in the table on the right will be present in the returned table. Full outer join provides the union of their results: tuples present only in either table are present.

An interesting feature of outer joins is that **none** of the outer joins is **associative**. We will use full outer join, which in some sense "seems" associative at first sight, to illustrate this.

Consider the three simple tables: A:

a	b
1	2

, B:

b	c
2	3

, and C:

a	c
4	5

.

The results of

```
select a,b,c
from (A natural full join B) natural full join C;
```

and

```
select a,b,c
from A natural full join (B natural full join C);
```

are respectively

a	b	c
1	2	3
4		5

and

a	b	c
1	2	
4		5
	2	3

.

2.2.4 Aggregation

Aggregation functions, including `min, max, sum, avg, count, etc`, appear in the `select` clause and perform computations over sets of values in multiple rows. They can also cooperate with `Group by` and `having` clauses to carry out more complex rules.

A few examples of using aggregation functions:

```
--avg GPA of all students who apply for CS
select avg(GPA)
from Student
where sID in (select sID from Apply where major = 'CS');
--min GPA of all students
select min(GPA)
from Student;
--count of colleges with enrollment larger than 2000
select count(*)
from College
```

```
where enrollment > 2000;
--count of students who apply for Cornell. Will return number of
    applications for Cornell without distinct
select count(distinct sID)
from Apply
where cName = 'Cornell'
--obscure but self-explaining.
select *
from Student S1
where (select count(*) from Student S2
    where S2.sID <> S1.sID and S2.GPA = S1.GPA) =
where (select count(*) from Student S2
    where S2.sID <> S1.sID and S2.sizeHS = s1.sizeHS);
--avgGPA of students applying for CS - avgGPA of other students
select CS.avgGPA - nonCS.avgGPA
from (select avg(GPA) as avgGPA from Student
      where sID in (
         select sID from Apply where major = 'CS')) as CS,
     (select avg(GPA) as avgGPA from Student
      where sID not in (
         select sID from Apply where major = 'CS')) as nonCS;
```

`group by` Clause

The `group by` clause calculates the value of aggregation functions for each group of tuples sharing the same value for the specified attributes, rather than for all the tuples.

```
--count of applications for each college.
select cName, count(*)
from Apply
group by cName;
--min and max for each college & major combination
select cName, major, min(GPA), max(GPA)
from Apply, Student
where Apply.sID = Student.sID
group by cName, major
--number of colleges applied by each student
select Student.sID, sName, count(distinct cName)
from Student, Apply
where Student.sID = Apply.sID
group by Student.sID;
```

Consider the last example above. Students who didn't apply for any college are excluded from the result. If we want them included, we can use the following query:

```
select Student.sID, sName, count(distinct cName)
from Student, Apply
where Student.sID = Apply.sID
group by Student.sID
union
select sID, sName, 0
from Student
where sID not in (select sID from Apply);
```

`having` Clause

The `having` clause is also only used in conjunction with aggregations. It is used after the `group by` clause to check some condition for each group.

```
--colleges with fewer than 5 applicantions.
select cName
from Apply
group by cName
having count(*) < 5;
```

2.2.5 `NULL` Values

Unless specified explicitly, any attribute can take a special value `NULL` when it is undefined or unknown. The existence of `NULL` makes it probable that a presumably tautological condition actually rules out tuples. For example:

```
Select sID from Student
where GPA > 3.5 or GPA <= 3.5;
```

Since it is possible that `GPA = null` for some students, the condition is not exhaustive for all students. Formally speaking, the condition in the `where` clause is evaluated using a 3-value logic: each expression can be `true, false` or `null`.

2.2.6 Data Modification Statements

So far we've been discussing query statements. In this section we will introduction SQL statements to modify data, including instructions to insert, delete and update data in the database.

Insert

Tuples can be inserted into a table via two kinds of statements:

- Insert a single tuple by specifying its values.

```
insert into College values('CMU', 'PA', 11500);
```

- Insert a series of tuples in the result of a select statement.

```
--Let all students who haven't applied for any college apply
    CS in CMU.
insert into Apply
select sID, 'CMU', 'CS', null
from Student
where sID not in (select sID in Apply)) S;
```

Delete

Delete statement takes the form `delete from table where condition`.

```
--remove all students who have applied for more than two majors
    from Student and Apply
delete from Student
where sID in (
  select sID from Apply
  group by sID
  having count(distinct major) > 2);
delete from Apply
where sID in (
  select sID from Apply
  group by sID
  having count(distinct major) > 2);--forbidden sometimes!
```

The second deletion is sometimes forbidden by the DBMS because the table from which tuples will be deleted appears in the condition. The solution is to create a temp table:

```
T = select sID from Apply
    group by sID
    having count(distinct major) > 2;
delete from Apply
where sID in T;
```

Update

Update statements take the form `update table set attr1 = expr1, ..., attrn = exprn where condition`.

```
--accept applicants for CMU with GPA < 3.6 but transfer them into
    economics major.
update Apply
set major = 'economics', decision = 'Y'
where cName = 'CMU' and
  sID in (select sID from Student where GPA < 3.6);
--transfer EE applicant with highest GPA to CS
update Apply
set major = 'CS'
where major = 'EE' and
  sID in
  select sID from Student
  where GPA >= all
    select GPA
    from Student
    where sID in (select sID from Apply where major = 'EE');
```

Chapter 3

Relational Design Theory

In this chapter we will focus on how to design good schemata for relational databases. Usually there exist many different designs for the schema of a database, some of them being better than others. A very nice theory has been developed for the design of relational databases.

3.1 Overview

Consider the example we used in the previous chapter, i.e. the database for students applying for colleges. The following information needs to be stored in the database:

- SSN and name of students
- Colleges applied for.
- High schools of students, with city
- Hobbies of students

Intuitively, a large schema can hold all the information:

```
1 Apply(SSN, sName, cName, HS, HCity, hobby)
```

However it is obviously not a good choice because a lot of duplicates will exist in the database, i.e. the same attribute value is likely to appear multiple times in different tuples. Furthermore, duplicates of an attribute value can be modified separately, which is a pitfall for inconsistence. These shortcomings are called **anomalies** of the design, respectively **redundancy** and **update anomaly**. Another peril is **deletion anomaly**. Since all attributes belong to the same table, it is possible that the deletion of some attributes result in inadvertent deletion of other attributes.

Instead one single large schema, we can use the following schema that includes several tables and induces no anomaly:

```
Student(SSN, sName)
Apply(SSN, cName)
HighSchool(SSN, HS, city)
Hobby(SSN, hobby)
```

The approach we will take is to design by decomposition. We will start with "mega" relations that contain all information, and then decompose them into smaller relations. The decomposition can be done automatically according to the properties of the data we specify. The resulting set of relations satisfy some sort of **normal form** that guarantees that no anomalies exist and that no information is lost. The properties and the related normal forms that we will cover are:

- Functional dependencies ⇒ Boyce-Codd Normal Form
- Multivalued dependencies ⇒ Fourth Normal Form

We will use the following college application information relations to illustrate the concepts:

```
Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)
Apply(SSN, cName, state, date, major)
```

3.2 FDs and BCNF

3.2.1 Functional Dependencies

Functional dependencies enables the DBMS to store the data more efficiently. It also facilitates the optimization of queries.

Suppose priority is an attribute determined only by GPA, which means that any two tuples with the same GPA have the same priority. Formally speaking, we have

$$\forall \mathtt{t}, \mathtt{u} \in \mathtt{Student} : \mathtt{t.GPA} = \mathtt{u.GPA} \Rightarrow \mathtt{t.priority} = \mathtt{u.priority}$$

By substituting GPA and priority with any two attributes A and B, we can get the generalized definition of a **functional dependency** $A \to B$ in a relation R:

$$\forall t, u \in R,\ t.A = u.A \Rightarrow t.B = u.B$$

Actually the definition can be further generalized. We can have a functional dependency between two groups of attributes $A_1, \ldots, A_n \to B_1, \ldots, B_m$:

$$\forall t, u \in R,\ t[A_1, \ldots, A_n] = u[A_1, \ldots, A_n] \Rightarrow t[B_1, \ldots, B_m] = u[B_1, \ldots, B_m]$$

In our example, we have the following functional dependencies in the `Student` table:

- SSN → sName

- SSN → address
- HScode → HSname, HScity
- HSname, HScity → HScode
- SSN → GPA
- GPA → priority
- SSN → priority

The functional dependencies in `Apply` table are trickier: they depend on constraints we set on what happens in the real world.

- Every college accepts applications only on a single date: cName → date
- Students can only apply for one major in each college: SSN, cName → major
- Students are only allowed to apply for colleges in one state: SSN → state

Functional dependencies generalize the notion of keys. Suppose we have a relation R with a series of attributes $\overline{A}, \overline{B}$ ($\overline{A}$ is the abbreviation for $A_1, \ldots, A_n$.) and R contains no duplicate tuples. In such case, having the functional dependency $\overline{A} \rightarrow$ *all attributes* means that $\overline{A}$ is the key of the table.

FDs can be classified into 3 types:

Trivial FD If $\overline{B} \subseteq \overline{A}$, then FD $\overline{A} \rightarrow \overline{B}$ is trivial.

Nontrivial FD If $\overline{B} \not\subseteq \overline{A}$, then FD $\overline{A} \rightarrow \overline{B}$ is nontrivial.

Completely Nontrivial FD If $\overline{A} \cap \overline{B} = \emptyset$, then FD $\overline{A} \rightarrow \overline{B}$ is completely nontrivial.

Completely nontrivial FD is the type in which we are the most interested. FDs obey the following basic rules.

Splitting Rule An FD $\overline{A} \rightarrow \overline{B}$ guarantees the FD $\overline{A} \rightarrow \overline{B'}$ for any $\overline{B'} \subseteq \overline{B}$, i.e. we can split the rhs of an FD. However this does not apply to the lhs.

Combining Rule The inverse of splitting rule: rhs of FDs can be combined if they share the same lhs.

Trivial-Dependency Rule We have $\overline{A} \rightarrow \overline{B}$ for any $\overline{B} \subseteq \overline{A}$.

Transitive Rule If we have FDs $\overline{A} \rightarrow \overline{B}$ and $\overline{B} \rightarrow \overline{C}$, then we must have the FD $\overline{A} \rightarrow \overline{C}$.

Algorithm 3.1 Calculation of $\{A_1, \ldots, A_n\}^+$

```
1: Start with set {A_1, ..., A_n}
2: repeat
3:     if A → B and A in set then
4:         add B to set
5: until set is not changed
```

The notion of **closure** of attributes is quite useful. The closure of attributes $\overline{A}$ is the set of all attributes B such that $\overline{A} \to B$. It is noted as $\overline{A}^+$ and can be calculated using Algorithm (3.1).

If $\overline{A}^+$ is equal to S, the set of all attributes of the table, then $\overline{A}$ is a key of the table. In order to find all keys of the table, we need to calculate the closures of all subsets of S, and pick out those whose closures are equal to S. In order for the algorithm to be more efficient, we can start with subsets of size 1 and increase the size gradually. Once a key has been found, it is guaranteed that all super sets of the key are keys.

Given two sets of FDs S_1 and S_2, S_2 is said to **follow from** S_1 if every relation instance satisfying S_1 also satisfies S_2. This can be verified by testing whether every FD in S_2 follows from S_1. To test whether an FD $\overline{A} \to \overline{B}$ follows from S, we can calculate $\overline{A}^+$ based on S and verify if $\overline{B} \subseteq \overline{A}^+$. It can also be tested with Armstrong Axioms, which will not be covered in detail here. When we specify FDs for a relation, what we look for is **a minimal set of completely nontrivial FDs such that all FDs that hold on this relation follow from the FDs in this set**.

3.2.2 Boyce-Codd Normal Form

In this section we will cover how to design relations in **Boyce-Codd Normal Form(BCNF)** using functional dependencies. Consider a relation $R(A_1, \ldots, A_n)$. If relations $R_1(B_1, \ldots, B_m)$ and $R_2(C_1, \ldots, C_k)$ satisfy

- $\overline{B} \cup \overline{C} = \overline{A}$,
- $R_1 \bowtie R_2 = R$,

then R_1 and R_2 are called a **decomposition** of R. The two properties are called **lossless join**. In this case we actually have $R_1 = \Pi_{\overline{B}} R$ and $R_2 = \Pi_{\overline{C}} R$.

A relation R with FDs is said to be in BCNF if $\overline{A}$ is a key for each $\overline{A} \to \overline{B}$. A relation not in BCNF allows two or more tuples to have the same value of $\overline{A}$ and B for some FD $\overline{A} \to \overline{B}$, which is called a **BCNF violation**.

The decomposition of relation R into relations in BCNF can be carried out with Algorithm (3.2).

In some cases it can be better to have larger relations in the result. For this purpose, instead of using $R_1(\overline{A}, \overline{B})$, we can use $R_1(\overline{A}, \overline{B}A^+)$.

Algorithm 3.2 BCNF decomposition

input:
Relation R + FDs for R
output:
Decomposition of R into BCNF relations with lossless join
1: Compute keys for R
2: **repeat**
3: Pick a relation R' with $\overline{A} \to \overline{B}$ that violates BCNF
4: Decompose R' into $R_1(\overline{A}, \overline{B})$ and $R_2(\overline{A}, rest)$
5: Compute FDs for R_1 and R_2
6: Compute keys for R_1 and R_2
7: **until** All relations are in BCNF

3.3 MVD and 4NF

3.3.1 Multi-valued Dependencies

Consider the relation

```
Apply(SSN,cName,hobby)
```

without any FD (i.e. colleges and hobbies are independent). In this case the key is the combination of all three attributes, and the relation is in BCNF. But this is not a good design because a student that applies for `m` colleges and has `n` hobbies will yield `mn` tuples, which obviously involves a lot of redundancy. Multi-value dependency is introduced to remove such redundancy.

A relation R is said to have multi-valued dependency $\overline{A} \twoheadrightarrow \overline{B}$ if $\forall t, u \in R$ that satisfy $t[\overline{A}] = u[\overline{A}]$, $\exists v \in R$ such that $v[\overline{A}] = t[\overline{A}]$, $v[\overline{B}] = t[\overline{B}]$ and $v[rest] = u[rest]$. In the example above, we have the MVD `SSN` $\twoheadrightarrow$ `cName`.

Consider a more complicated example:

```
Apply(SSN,cName,date,major,hobby)
```

Here we suppose that hobbies are revealed to colleges selectively (i.e. `SSN` $\twoheadrightarrow$ `cName` no longer holds). We also require that each student can apply for each college only once on each date. Students can apply for multiple majors of each college. These specifications leave us with the following dependencies:

- `SSN`, `cName` $\to$ `date`,
- `SSN`, `cName`, `date` $\twoheadrightarrow$ `major`.

As with FDs, we have the following notions:

Trivial MVD An MVD $\overline{A} \twoheadrightarrow \overline{B}$ is trivial if either $\overline{B} \subseteq \overline{A}$ or $\overline{A} \cup \overline{B} = \{all\ attributes\}$.

Nontrivial MVD MVDs that are not trivial.

The triviality when $\overline{A} \twoheadrightarrow \overline{B}$ comes from the fact that u itself can serve as the v whose existence is required by the definition. The case in which $\overline{A} \cup \overline{B} = \{all\, attributes\}$ is trivial because there is no "rest", thus t itself can serve as the v.

The following rules[1] hold for MVD:

FD-is-an-MVD Rule If $\overline{A} \rightarrow \overline{B}$, then $\overline{A} \twoheadrightarrow \overline{B}$. u itself serves as the v in the definition of MVD.

Intersection Rule In the presence of $\overline{A} \twoheadrightarrow \overline{B}$ and $\overline{A} \twoheadrightarrow \overline{C}$, we must have $\overline{A} \twoheadrightarrow \overline{B} \cap \overline{C}$.

Transitive Rule If $\overline{A} \twoheadrightarrow \overline{B}$ and $\overline{B} \twoheadrightarrow \overline{C}$, then $\overline{A} \twoheadrightarrow \overline{C} - \overline{B}$.

3.3.2 4th Normal Form

A relation R with MVDs is said to be in 4th Normal Form(4NF) if for each nontrivial $\overline{A} \twoheadrightarrow \overline{B}$, $\overline{A}$ is a key. Since all FDs are MVDs, a relation in 4NF is guaranteed to be in BCNF, i.e. 4NF is stronger than BCNF.

The decomposition of relation R into relations in 4NF is provided in Algorithm (3.3).

Algorithm 3.3 4NF decomposition

input:
Relation R + FDs for R + MVDs for R
output:
Decomposition of R into 4NF relations with lossless join
1: Compute keys for R (using FDs)
2: **repeat**
3: Pick a relation R' with $\overline{A} \twoheadrightarrow \overline{B}$ that violates 4NF
4: Decompose R' into $R_1(\overline{A}, \overline{B})$ and $R_2(\overline{A}, rest)$
5: Compute FDs and MVDs for R_1 and R_2
6: Compute keys for R_1 and R_2
7: **until** All relations are in 4NF

Consider the example above concerning the relation

```
1 Apply(SSN,cName,date,major,hobby).
```

According to MVD `SSN, cName, date` $\twoheadrightarrow$ `major`, it can be decomposed into

```
1 A1(SSN,cName,date,major),
2 A2(SSN,cName,date,hobby).
```

Then according to the FD `SSN, cName` $\rightarrow$ `date`, `A1,A2` can be decomposed into

[1] Proof TBD

```
A3(SSN,cName,date),
A4(SSN,cName,major),
A5(SSN,sName,hobby).
```

3.4 Shortcomings of BCNF and 4NF

BCNF and 4NF are supposed to help us design "good" relations up to our needs. However, in some cases relations in BCNF or 4NF are not satisfactory. We will provide a few examples to illustrate their shortcomings.

Example 1

Consider the relation

```
Apply(SSN,cName,date,major).
```

Suppose that each student can only apply for each college once for one major, and that colleges have non-overlapping application dates (but one college can have several application dates). In this case we have FDs

- SSN, cName→date, major;
- date→cName.

The key of the relation is `(SSN,cName)`. It is not in BCNF because `date` is not a key. Decomposition of `Apply` into relations in BCNF yields

```
A1(date,cName),
A2(SSN,date,major).
```

They are in BCNF, which however does not make the schema a good design. We would rather keep the original schema, which is actually in **3rd normal form**, a normal form weaker than BCNF.

Example 2

Different orders to apply FDs during the decomposition may result in different relations, sometimes not satisfactory. Consider the relation

```
Student(SSN,HSname,GPA,priority)
```

with FDs

1. SSN→GPA,
2. GPA→priority,
3. SSN→priority,
4. SSN→GPA, priority.

Its key is `(SSN,HSname)` and it is not in BCNF. During the decomposition, if we apply the FDs in the order FD3, FD1, we will end up with

```
S1(SSN,priority),
S2(SSN,GPA),
S3(SSN,HSname).
```

If we start with FD4, we will end up with

```
S1(SSN,GPA,priority),
S2(SSN,HSname),
```

which is clearly a better design.

Example 3

Sometimes it might be preferable not to carry out the decomposition. Consider the relation

```
Score(SSN,sName,SAT,CAT)
```

which keeps record of scores students get in SAT and CAT tests. Each student can take the tests multiple times. The only FD is

- SSN→sName.

The relation can be decomposed into three relations in BCNF, but we might prefer not to do that if all queries that we will make return the combination of student name and test scores.

Example 4

Sometimes rather than decomposing a large relation, we might want to compose small relations that have been "over-decomposed" into larger ones that are still in BCNF/4NF.

Chapter 4

Unified Modeling Language

4.1 UML Data Modeling

Data modeling means the way in which data is represented in applications. There are concrete models like relational model and XML model. But during the design phase, we tend to represent the data in a high-level model that can later be translated, usually automatically, into models specific to various implementations. Famous high-level design models include Entity-Relationship Model(E/R) and **Unified Modeling Language(UML)**, the latter of which is nowadays the mainstream model. It is a graphical model applied to both DB design and program design. We will focus on the data modeling subset of UML.

UML data modeling contains 5 concepts: classes, associations, association classes, subclasses, composition & aggregation.

4.1.1 Class

Class is a UML concept shared by data modeling and program design. It contains a **name**, a series of **attributes** and a series of **methods**. For the purpose of data modeling, we add the concept of **primary key(pk)** for the methods, and we don't need any method. Classes look a lot like relations and can be translated directly into relations. For our student application example, we will have a class `Student` with attributes `sId(pk),sName,GPA`, a class `College` with attributes `cName(pk),state`, as shown in Figure 4.1.

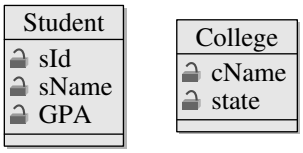

Figure 4.1: Example of UML classes

4.1.2 Association

Association captures relationships between objects of two classes. For example, there can exist `Apply` relationship between `Student` and `College`, as shown in Figure 4.2.

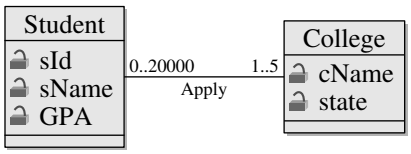

Figure 4.2: Example of UML associations

Multiplicity of an association specifies how many objects of class A can be related to an object of class B. In Figure 4.2, two multiplicities are marked: each student can apply for up to 5 colleges and must apply for at least one college, while each college takes no more than 20000 applications. If no upper limit exists, its position should be held by a `*`. `0..*` can be abbreviated as `*`, while `1..1` can be abbreviated as `1`. Concepts often used are one-to-one, many-to-one, one-to-many and complete associations. "Complete" means that all objects must be involved in the association, i.e. the lower limit is 1. If no multiplicity is specified, the default is `1..1`, i.e. complete one-to-one association.

Association can exist between a class and itself. For example, a college's main campus and its branches, students and their siblings, etc.

4.1.3 Association Class

Association classes captures relationships with attributes between objects of two classes. For example, the `apply` relationship has attributes `date,decision`, as shown in Figure 4.3.

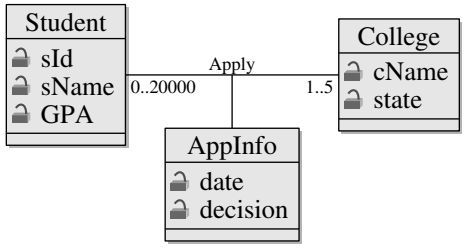

Figure 4.3: Example of UML association classes

A shortcoming of UML is that it cannot describe multiple associations between two objects that exist together, i.e. only one association can be captured between a pair of objects. In our example, if students can apply for the same

college multiple times (e.g. for different majors), we can no longer use the association class `AppInfo`. A class `AppInfo` with separate relationships to students and colleges needs to be added.

Note that if the multiplicity of an association is `1..1` or `0..1`, it is actually unnecessary to have an association class to capture its attributes: they can be moved into the object itself.

4.1.4 Subclass

Subclass provides description of inheritance relationship in UML. An example is given in Figure 4.4. Superclass `Student` has 3 subclasses: `ForeignS` for foreign students, `DomesticS` for domestic students, and `APStudent` for students who took AP courses. Each AP student can take 1-10 AP courses, and there is at least 1 student in each AP course.

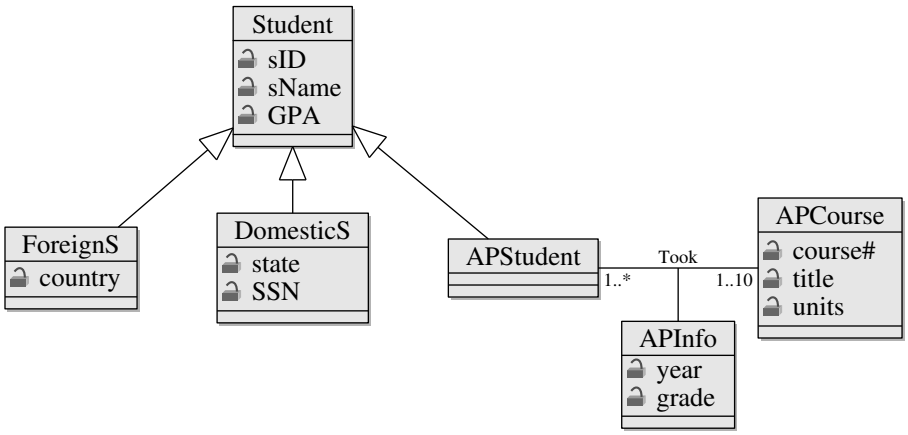

Figure 4.4: Example of UML subclasses

In UML, superclass is called **generalization**, while subclass is called **specialization**. A subclass relationship is said to be **complete** if each instance of the superclass belongs to at least one subclass, otherwise it is said to be **incomplete/partial**. If each instance of the superclass belongs to at most one subclass, the subclass relationship is said to be **disjoint/exclusive**, otherwise it is said to be **overlapping**.

4.1.5 Composition & Aggregation

Composition describes the relationship that objects of one class necessarily belong to objects of another class. If such belonging relationship is not necessary, it is called **aggregation**[1]. An example is provided in Figure 4.5. A college consists of a few departments, and each college can have a few apartments as its property, while an apartment does not have to belong to a college.

[1] Aggregation in UML has nothing to do with the aggregation functions in SQL.

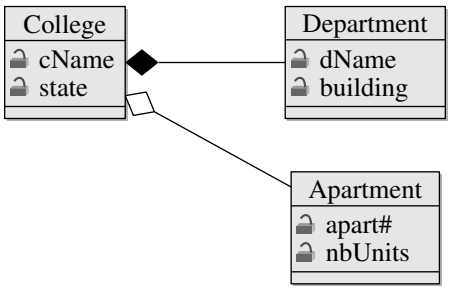

Figure 4.5: Example of UML composition & aggregation

4.2 UML to Relations

Designed specified with UML can be translated into relations automatically provided that each "regular" class(i.e. not an association class, a subclass or a class contained in another class in a composite relationship) has a key.

4.2.1 Class

Classes in UML can be translated directly into relations. Each attribute of the class becomes an attribute of the relation, and the primary key becomes the key attribute of the relation. The classes in Figure 4.1 can be translated into

```
1 Student(sID,sName,GPA),
2 College(cName,state).
```

4.2.2 Association

Each association will be translated into its own relation that grabs the key from both sides of the association. The `Apply` association in Figure 4.2 can be translated into

```
1 Apply(sID, cName).
```

The key of the generated relation depends on the multiplicity of the association. If we have `0..1` or `1..1` on one side of the association, then the primary key of the other side is the key of the relation generated from the association. Otherwise the key is the combination of the two primary keys.

Actually this relation may be not needed at all, depending on the multiplicity of the association. If we have `0..1` or `1..1` on one side of the association, the information can be folded into the relation generated from the class on the other side of the association. If we let the association in Figure 4.2 have `0..1` on the right side, we can get rid of the `Apply` relation and change `Student` into

```
1 Student(sID,sName,GPA,cName).
```

Actually even if we have `0..2` on the right side of the association, we can still get rid of `Apply` by changing `Student` into

```
Student(sID,sName,GPA,cName1,cName2).
```

In the case of self association, the keys from both sides need to be renamed or indexed. For example, for the sibling association between two students, its schema is

```
Sibling(sID1,sID2).
```

4.2.3 Association Class

The relation generated from an association class is the relation generated from the association plus the attributes of the association class. For Figure 4.3, the `Apply` relation becomes

```
Apply(sID,cName,date,decision).
```

4.2.4 Subclass

There exist 3 commonly used translations for subclass relationship:

1. Subclass relations contain superclass key and specialized attributes.
2. Subclass relations contain all superclass attributes and specialized attributes.
3. One relation contains all superclass attributes and subclass attributes.

The choice among the 3 options depends on properties of the relationship. If the subclasses are heavily overlapping, the design 3 is preferable. If the subclasses are disjoint and complete, then design 2 becomes the best choice. Let $sz1, sz2, sz3$ represent number of tuples in each design for s specific set of data, in general we have $sz3 \leq sz2 \leq sz1$, because design 3 has one tuple for each object regardless of subclassing, design 2 has one tuple for each object, plus one more each time an object is in more than one subclass, and design 1 has one tuple for each object plus one more each time an object is in a subclass.

Note that subclasses are not considered as "regular" classes that must have a key for automatic translation. In fact a subclass inherits the key from its superclass.

4.2.5 Composition & Aggregation

In a composition/aggregation relationship, we add the key of the container class into the relation of the contained class. In the case of aggregation, the key can be NULL. The composition & aggregation relationship in Figure 4.5 results in the following relations:

```
College(cName,state),
Department(dName,building,cName),
Apartment(apart#,nbUnits,cName).
```

Note that in a composite relationship, the relation generated from the contained class (e.g. `Department` above) is not considered as a "regular" class.

Chapter 5

Indexes and Transactions

5.1 Indexes

Indexes, or indices are the primary mechanism to gain improved performance in relational database. It's a persistent data structure stored in the database. We will focus on its application (instead of implementation).

When we want to speed up queries that involve conditions over certain columns, we can build **indexes** on those columns. Indexes help the query execution engine to run the queries much faster. Rather than scanning the entire table, it can now locate eligible tuples directly. Indexes can be built on one single attribute or combination of a few attributes. Typically, the performance can be improved by a few orders of magnitude.

Indexes can be implemented with either balance trees (B tree/B+ tree) or hash tables. Balance tree implementation can be used to accelerate queries on comparison or equality conditions, while hash table implementation can only speed up equality tests. Balance tree has logarithmic running time while hash table has constant running time, thus is preferable when all conditions are equality tests.

Many DBMSs build indexes automatically on the primary key attributes, sometimes also on unique attributes.

Indexes do have some drawbacks.

1. Occupies extra space. Not serious in modern systems.
2. Overhead for index creation. Can be quite time-consuming, but usually worthwhile.
3. Overhead for index maintenance. Index has to be rebuilt when the database is modified. If the db is modified too frequently, it may be inappropriate to build indexes on it.

Considering these drawbacks, the benefit of an index depends on:

- size of the table;

- data distributions;
- query v.s. update load.

Over history, **physical design advisors** have been developed to help us decide whether and where to build indexes. They take the database (usually its statistics) and the workload as input, and outputs recommended indexes to build.

SQL statements involving indexes are listed as follows.

```
--Create index on attribute(s)
Create Index IndexName on T(A);
Create Index IndexName on T(A1,...,An);
--Create index with the requirement that no duplicates exist for A.
    Report error otherwise.
Create Unique Index IndexName on T(A);
--Remove an index
Drop Index IndexName;
```

5.2 Transactions

5.2.1 Motivation

The concept of transaction is motivated by two independent requirements:

- Concurrent DB access;
- Resilience to system failures.

Concurrent access to the DB can happen at different levels:

1. **Attribute Level**. If the following two statements are executed concurrently,

```
update College set enrollment = enrollment + 1500 where cName
    = 'Stanford';
update College set enrollment = enrollment + 1000 where cName
    = 'Stanford';
```

   The enrollment of Stanford can end up increased by 1000, 1500 or 2500.

2. **Tuple Level**. Two concurrent statements as follows

```
update Apply set major = 'CS' where sId = 123;
update Apply set decision = 'Y' where sId = 123;
```

   are not guaranteed to update the two attributes as expected. It is possible that only one of them is reflected in the result.

3. **Table Level**. The two concurrent statements

```
update Apply set decision = 'Y'
where sID in (select sID from Student where GPA>3.9);
update Student set GPA=1.1*GPA where sizeHS>2500;
```

are expected to be executed in a certain sequence. A mechanism should exist to guarantee the consistency.

4. **Multi-statement Inconsistency**.

```
--Client 1
insert into Archive
select * from Apply where decision='N';
delect from Apply where decision='N';
--Client 2
select count(*) from Apply;
select count(*) from Archive;
```

The execution of the two groups of statements by two clients are not expected to overlap, otherwise the result received by client 2 is meaningless.

The goal of handling concurrent access to DB is to **make the client feel that the sequence of SQL statements he or she initiated is executed in isolation**. A naive but safe approach would be to always execute them in isolation. But we do want to enable concurrency whenever it is safe to do so.

The ability to survive system failures is essential for databases. The goal of handling system failures is to **guarantee all-or-nothing execution regardless of failures**. In case of a system failures, a sequence of statements in the process of being executed should be rolled back.

5.2.2 Properties

Transaction is the mechanism that handles both concurrency and failures. A transaction is a sequence of one or more SQL operations treated as a unit. A transaction appears to run in isolation, and guarantees to be executed entirely or not at all.

In standard SQL, a transaction begins automatically on the fist SQL statement. On `commit`, a transaction ends and a new one begins. The current transaction also ends when the session terminates. In `autocommit` mode, each single statement is viewed as a transaction.

A transaction implementation should have the ACID properties: Atomicity, Consistency, Isolation, Durability.

Atomicity

Atomicity guarantees that each transaction is all-or-nothing. If the systems crashes during the execution of a transaction, all modifications to the database will be rolled back when the system is restarted.

Although the rollback mechanism is designed to handle crashes, it can also be initiated by client. It can be helpful but must be used with caution: the system rolls back, and only rolls back modifications to data in the database. Other side effects cannot be undone.

Consistency

An integrity constraint is a specification of legal states of the database. Following the consistency requirement, each client can assume that all constraints hold at the beginning of a transaction, and must guarantee that all constraints hold after the transaction ends.

Isolation

A few clients may each initiate a series of transactions, each containing a series of statements. The execution of all the statements must have **serializability**: execution of statements in different transactions may be interleaved, but the execution must be equivalent to some sequential / serial order of all transactions.

Serializability guarantees that the execution is equivalent to some order of all transactions, but the results of different orders are not guaranteed to be the same. In examples 3 and 4 above, the results are different for different orders.

Durability

Durability guarantees that in case the system crashes after a transaction commits, all effects of the transaction remain in the database. The implementation is based on the idea on logging.

5.2.3 Isolation Levels

The transaction mechanism guarantees safe behavior of the database in case of concurrency and failures. But it does cause some overheads reduction of concurrency. DBMSs offer weaker **isolation levels** to counteract against the deficiencies. Three levels are listed in the SQL standard: **read uncommitted, read committed, repeatable read**, from the weakest to the strongest. They induce less overhead, allow more currency, but guarantee less consistency.

Isolation levels are set at transaction level. Each client can set different isolation levels for different transactions. They are set "in the eye of the beholder": the isolation level of a transaction does not have any effect on behavior of any other transactions running concurrently.

A data item is said to be **dirty** if it is written by an uncommitted transaction. Consider the two transactions:

```
1 --T1
2 update College set enrollment=enrollment+1000
3 where cName='Stanford';
4 --T2
5 select avg(enrollment) from College;
```

T1 modifies the value of enrollment for Stanford, which is read by T2. If T1 gets rolled back for some reason, what T2 has read would be a value that never existed in the DB. Such a read is called a **dirty read**. The notion of dirty read only exists between different transactions. A read inside the same transaction is never a dirty read.

Read Uncommitted

A transaction having the isolation level read uncommitted can perform dirty reads. Serializability is obviously broken.

Read Committed

A transaction having the isolation level read committed cannot perform dirty reads. Nonetheless, it does not guarantee serializability. If a value modified in transaction T1 is read twice in transaction T2, it is possible that the two reads occurred respectively before and after T1, which breaks serializability.

Repeatable Read

A transaction with the isolation level repeatable read requires that no dirty reads be performed, and that an item read multiple times cannot change value.

However, even with the additional condition, serializability is not guaranteed. If two values modified in transaction T1 are both read in transaction T2, it is possible that T2 will read the two values respectively before and after T1, which apparently breaks serializability.

Moreover, repeatable read does not prevent new tuples from being added into the relation, which causes the problem of "phantom tuples". Consider the example

```
1 --T1
2 insert into Student /*100 new tuples*/
3 --T2
4 set transaction isolation level repeatable read;
5 select avg(GPA) from Student;
6 select max(GPA) from Student;
```

It is possible that `avg(GPA)` and `max(GPA)` are calculated respectively before and after the insertion, which breaks serializability. The tuples inserted in T1 are the phantom tuples. Note that the problem does not exist for deletion.

Repeatable read is chosen by some mainstream DBMSs as the default isolation level.

5.2.4 Read Only Transaction

Transactions can also be set `Read Only` for optimized performance. It is independent from isolation levels.

Chapter 6

Constraints and Triggers

Constraints, or integrity constraints constrain the allowable states of the database, while triggers monitor database changes, check conditions and possibly initiates actions on being activated.

6.1 Overview

6.1.1 Constraints

Integrity constraints impose restrictions on allowable data beyond those imposed by the structure of the DB and types of the attributes. For example, in our `Student` table, the `GPA` value is required to fall inside (0.0, 4.0]; in the `Apply` table and the value of `decision` should be either `Y`, `N` or `Null`. Constraints can be logically more complicated, e.g. the condition that no decisions have been made for CS applicants can be expressed as the constraint `major = 'CS'` $\Rightarrow$ `decision = Null`.

Constrains are enforced for several purposes:

- To automatically catch data-entry error when doing insertions.
- As correctness criteria when doing updates.
- To enforce consistency.
- To inform the system of properties of the data so as to facilitate better storage and queries.

Constraints are classified into:

- Non-null constraints. An attribute can be forbidden from having null value.
- Key constraints. Attributes specified as key should have unique value combinations.

- Referential integrity (foreign key).
- Attribute-based constraints.
- Tuple-based constraints.
- General assertions.

Constraints can be declared either when the schema is created or later. In the former case, it is checked after bulk loading, while in the latter case it is checked on the state of the DB at the time of the declaration.

Normally a constraint should be checked after every modification that may cause a violation to it. Constraint checking can also be carried out only after each transaction, which is called deferred constraint checking: constraints are allowed to be violated during the transaction, but if any violation still exists after the transaction is completed, the whole transaction gets rolled back.

6.1.2 Trigger

Trigger is a more dynamic concept than constraints. Constraints put restrictions on each state of the DB, while triggers limit how the DB evolves. Triggers are also called "Event-Condition-Action Rules" because they usually have the form

When *event* occurs, check *condition*; If true, do *action*.

For example, a trigger can be set up to accept applications from student with a GPA higher than 3.95 automatically: `insert application with GPA>3.95 ⇒ accept`.

Triggers are enforced for the following purposes:

- To move logic from application into DBMS.
- To enforce constraints. In mainstream implementations, the trigger feature is more expressive than the constraint feature. Moreover, triggers make it possible to repair the DB when a constraint violation is found.

6.2 SQL for Constraints

6.2.1 Non-null Constraints

To add a non-null constraint to an attribute when creating the table, we need to add the keyword `not null` to the attribute.

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

6.2.2 Key Constraints

A key constraint requires that values of the specified attribute be unique.

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

Suppose the table contains two tuples:

```
1 234 Bob 3.6 1500
2 123 Amy 3.9 1000
```

The statement

```
1 update Student set sID = sID - 111;
```

is possible to cause an error because Bob will have the same sID as Amy if it gets updated first. This demonstrates the situation in which deferred constraint checking should be used.

There can be only one primary key in each table. If attributes other than the primary key is supposed to be unique, the keyword `unique` should be used.

```
1 create table Student(sID int primary key, sName text unique, GPA
      real, sizeHS int);
```

The following examples demonstrate the case in which a combination of attributes is used as the primary key.

```
1 create table College(cName text, state text, enrollment int,
      primary key (cName,state));
2 create table Apply(sID int, cName text, major text, decision text,
      primary(sID, cName), unique(sID, major));
```

Note that Null values are handled specially. SQL standard and most implementations allow repeated Null values even if the column is declared as unique. However repeated Null values are usually not permitted for the primary key.

6.2.3 Attribute/Tuple Based Constraints

Attribute-based constraints are often used to catch data entry errors concerning ranges of attributes. They are check whenever a tuple is inserted or updated. In the following example we put limits to the range of GPA and sizeHS.

```
1 create table Student(
2   sID int, sName text,
3   GPA read check(GPA <= 4.0 and GPA > 0.0),
4   sizeHS int check(sizeHS < 5000));
```

Tuple-Based constraints also catches data entry errors, but it puts limits to relationship of different attributes inside a tuple. In the following example, the tuple-based constraint rules out successful applications for CS major of Stanford.

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

Attribute/Tuple based constraints may be syntactically allowed but not enforced in MySQL. Currently sub-queries are allowed in check constraints in no implementation, although they are listed in SQL standard and might be syntactically permitted in certain implementations.

6.2.4 General Assertion

General assertions are very powerful, but unfortunately not supported in any mainstream DBMS. In the SQL standard it takes the form

```
--A is unique
create assertion Key
check ((select count(distinct A) from T) =
       (select count(*) from T));
--Apply only contains sID from Student
create assertion ReferentialIntegrity
check (not exists (select * from Apply
  where sID not in (select sID from Student)));
--Average GPA of admitted students higher than 3.0
create assertion AvgAccept
check (3.0 < (select avg(GPA) from Student
  where sID in (select sID from Apply
    where decision = 'Y')));
```

The complexity of implementing general assertion comes from the requirement that it has to be checked each time any modification that can possibly cause violation is made to the DB.

6.2.5 Referential Integrity

Referential integrity handles the integrity of references in a database. No "dangling pointer" is allowed. In our example, it should be guaranteed that an `sID` in `Apply` refers to a student in `Student`, and that a `cName` in `Apply` refers to a college in `College`, i.e. we have referential integrities `Apply.sID to Student.sID` and `Apply.cName to College.cName`, as shown below. Note that referential integrity is a directional constraint.

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

In a referential integrity `R.A to S.B`, `A` is called the **foreign key**. `B` is usually required to be the primary key of `S`, or at least a `unique` attribute. Multi-attribute foreign key is allowed.

The enforcement of referential integrity `R.A to S.B` should consider possible violations by the following operations:

- insert into `R`;
- delete from `S`;
- update `R.A`;
- update `S.B`.

For insertion to `R` and update of `R.A`(the referencing table), the referential integrity is simply checked. The referential integrity also prevents us from dropping table `S`. For deletion from `S`, three types of actions are possible:

Restrict(default) Generate an error.

Set Null Set referencing tuples in `R` to have Null value for `R.A`.

Cascade Delete all referencing tuples in `R`.

Actions for update of `S.B`(the referenced table) are similar, except that for cascade, the same update is propagated to `R.A` in `R`. The actions can be set in the form

```
1 create table Apply(sID int references Student(sID) on delete set
      null,
2         cName text references College(cName) on update cascade,
3         major text, decision text);
```

We can have referential integrities within the same table:

```
1 create table T(A int, B int, C int, D int, primary key(A,B),
2   foreign key (B,C) references T(A,B) on delete cascade);
```

6.3 SQL for Triggers

Implementations of trigger vary significantly across different systems. We will talk about triggers in the SQL standard. The syntax of trigger is as follow.

```
1 Create Trigger name
2 Before | After | Instead of events
3 [referencing-variables]
4 [For Each Row]
5 when (condition)
6 action
```

The trigger should be activated `before/after/instead of` the `events` listed in the events clause, which can be insert on table T, delete on table T or update of a few columns(optional) on table T. Each event can involve several tuples. If `For Each Row` is specified, the trigger is activated for each of these

tuples, while the trigger is activated only once if there is no `For Each Row`. The `referencing-variables` clause provides a method to access the data whose modification activated the trigger. Old rows, new rows, old tables and new tables can be referenced, with clauses of the form `old row/new row/old table/new table as var`. Note that `old/new row` can only be referenced when the event is row-level (its contrary is statement-level), i.e. when `For Each Row` is enabled. `condition` and `action` are self-explaining.

A simple example is provided below. It implements the referential integrity `R.A to S.B` with cascaded deletion.

```
1  --row-level
2  Create Trigger Cascade
3  After Delete On S
4  Referencing Old Row as O
5  For Each Row
6  Delete From R where A = O.B;
7
8  --statement-level
9  Create Trigger Cascade
10 After Delete On S
11 Referencing Old Table as OT
12 Delete From R where A in (select B from OT);
```

There are plenty of tricky issues when handling triggers.

- Row-level v.s. statement-level. `Old/New Row` can only be referenced when the trigger is row-level. The difference is intuitive when the trigger is activated after the event, but it can get trickier when the activation happens before or instead of the event.
- Multiple triggers can be activated at the same time. The order of the activations matters.
- The action of one trigger may activate another trigger, causing a chaining effect. A trigger that can activate itself leads to self-triggering, and a few triggers can form a cycle. In both cases, termination of the chain becomes an issue. Moreover, a trigger may take several actions that activates different triggers, resulting in nested invocations of triggers.
- Conditions can be put either in the `when` clause or as part of the `action`. Sometimes the efficiency differs a lot among different choices.

The following complex example using Table `T(K,V)` (`K` is the key) demonstrates the subtlety of row-level v.s. statement-level. Each time an insertion is performed on table `T`, the trigger checks for each newly inserted tuple whether the average of `V` in table `T` is smaller than the average of `V` in the inserted tuples. If so, then `V` of the current tuple being processed s incremented by 10.

```
1  Create Trigger IncreaseInserts
2  After Insert on T
3  Referencing New Row as NR, New Table as NT
4  For Each Row
```

```
When (select avg(V) from T) < (select avg(V) from NT)
Update T set V=V+10 where K=NR.K;
```

There is no statement-level equivalent to this row-level trigger. Moreover, its final state is nondeterministic because its behavior depends on the order in which the inserted tuples are checked. We will illustrate different aspects of trigger with a few examples using SQLite, which differs from the SQL standard in the following aspects:

- Only row-level trigger is implemented. `for each row` is implicit if not specified.
- Triggers are activated immediately after each row-level change instead of after the whole statement as specified in the standard. As a result, there is no reference to `old table` or `new table`. Actually there is no referencing clause. `Old` and `New` are predefined for `Old Row` and `New Row`.
- The `action` in a trigger is contained in a `begin-end` block.

```
--All newly inserted students with GPA in (3.3,3.6] apply for
    geology of Stanford and biology of MIT.
create trigger R1
after insert on Student
for each row
when New.GPA > 3.3 and New.GPA <= 3.6
begin
  insert into Apply values(New.sID,'Stanford','geology',null);
  insert into Apply values(New.sID,'MIT','biology',null);
end;
```

```
--cascade deletion in Student to Apply
create trigger R2
after delete on Student
for each row
begin
  delete from Apply where sID = Old.sID;
end;
```

```
--cascade update in College to Apply
create trigger R3
after update of cName on College
for each row
begin
  update Apply
  set cName = New.cName where cName = Old.cName;
end;
```

```
--enforce key constraint on College
create trigger R4
before insert on College
for each row
when exists (select * from College where cName = New.cName)
```

```
begin
  select raise(ignore);--ignore the current modification
end;

create trigger R5
before update of cName on College
for each row
when exists (select * from College where cName = New.cName)
begin
  select raise(ignore);--ignore the current modification
end;
```

```
--update college name when number of applications > 10
create trigger R6
after insert on Apply
for each row
when (select count(*) from Apply where cName = New.cName) > 10
begin
  update College set cName=cName||'-Done' where cName=New.cName;
end;
```

```
--limit sizeHS
create trigger R7
after insert on Student
for each row
when New.sizeHS < 100 or New.sizeHS > 5000
begin
  delete from Student where sID = New.sID;
end;
```

Note that R7 and R1 can be activated at the same time. If we insert a student with GPA = 3.6 and sizeHS = 10000, the applications generated by R1 will stay in Apply, while the tuple inserted into Student will be deleted according to R7.[1]

```
--process some applications when enrollment passes threshold of
    16000.
create trigger TooMany
after update of enrollment on College
for each row
when (Old.enrollment <= 16000 and New.enrollment > 16000)
begin
  delete from Apply where cName = New.cName and major = 'EE';
  update Apply
    set decision = 'U'
    where cName = New.cName and decision = 'Y';
end;
```

TooMany takes action based on the dynamic behavior of the DB, which cannot be done with constraints.

The follow examples use tables Ti(A:Int), i = 1,2,3,4.

```
--self-triggering
```

[1] R2 is not taken into account.

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

The trigger above will end up in infinite loop if no limit is put on the times for which self-triggering can happen. The parameter `recursive_triggers` controls this behavior. By default it is off, and the trigger can be activated at most once. If we turn it on with

```
pragma recursivetriggers = on;
```

the activation can happen multiple times. A limit has to be put in the `when` clause to avoid infinite loop error.

```
--two triggers are actiavetd at the same time and act on the same
    data
create trigger R1
after insert on T1
for each row
begin
  update T1 set A = 2;
end;

create trigger R2
after insert on T1
for each row
when exists (select * from T1 where A = 2)
begin
  update T1 set A = 3;
end
```

The two triggers above are activated at the same time and they both modify the value of `A`. `A` may end up with value 2 or 3, depending on the order in which the two triggers are defined.