1 Database Modifications

1.1 Insertion

The basic form of insertion statement is:

`INSERT INTO R(`$A_1, \ldots, A_n$`) VALUES (`$v_1, \ldots, v_n$`)`

A row is created using the value v_i for attribute A_i, for $i = 1, 2, \ldots, n$. If the list of attributes does not include all attributes of the relation R, then the row created has default values for all missing attributes.

Suppose we have the following relation:

```
StarsIn(movieTitle, movieYear, starName)
```

We wish to add Sydney Greenstreet to the list of stars of *The Maltese Falcon*. Then we say:

```
INSERT INTO StarsIn(movieTitle, movieYear, starName)
VALUE("The Maltese Falcon" 1942, "Sydney Greenstreet")
```

The effect of executing this statement is that a row with the three components in line 2 is inserted into the relation `StarsIn`. Since all attributes of `StarsIn` are mentioned in line 1, there is no need to add default components. The values in line 2 are matched with the attributes in line 1 in the order given. Note that if we provide values for all attributes in a relation, then we may omit the list of attributes that follows the relation name. However, if we take this option, we must ensure that the order of the values is the same as the standard order of attributes for the relation:

```
INSERT INTO StarsIn
VALUE("The Maltese Falcon", 1942, "Sydney Greenstreet")
```

If unsure of declared order for attributes, it is best to list them in the `INSERT` clause for their values in the `VALUES` clause.

We can also compute a set of rows to be inserted, using a subquery. This subquery replaces the keyword `VALUES` and the row expression in the `INSERT` statement form described above.

Suppose we want to add to the relation

`Studio(name, address, presC#)`

all movie studios that are mentioned in the relation

`Movies(title, year, length, genre, studioName, producerC#)`

but do not appear in `Studio`. Since there is no way to determine an address or a president for such a studio, we shall have NULL for attributes `address` and `presC#` in the inserted `Studio` rows. A way to make this insertion is as follows

```
INSERT INTO Studio(name)
SELECT DISTINCT studioName
FROM Movies
WHERE studioName NOT IN
               (SELECT name
                FROM Studio);
```

1.2 Deletion

The form of a deletion is

`DELETE FROM R WHERE <conditions>;`

so every row satisfying the condition will be deleted from relation `R`

We can delete from relation

`StarsIn(movieTitle, movieYear,starName)`

the fact that Sydney Greenstreet was a star in *The Maltese Falcon* by:

```
DELETE FROM StarsIn
WHERE movieTitle = "The Maltese Falcon" AND
     movieYear = 1942 AND
     starName = "Sydney Greenstreet";
```

Unlike insertion statements, we cannot specify exactly a row to be deleted. Rather, we must describe the row by a `WHERE` clause. With this logic, we can deleted several rows at once easily. For instance the query below deletes all movie executives whose net worth is less than ten million dollars.

```
DELETE FROM MovieExec
WHERE netWorth < 10000000;
```

1.3 Foreign Keys

The order of insertions and deletions should follow the hierarchy of parent tables and child tables. Suppose we have two tables `Artist` and `Album`. `Album` has a foreign key of `artistID`, and an album can not exist if there is not a matching artist in the `Artist` table. To insert a new album, first we would need to insert the artist into the `Artist` table, and then insert the new album into the `Album` table. If we switch the order of insertion we will receive an error.

Likewise, if we try to delete an artist without first deleting the artist's albums, we may encounter different results depending on the `artistID` foreign key setting:

- `RESTRICT` the deletion would be blocked
- `CASCADE` The corresponding albums will also be deleted
- `SET NULL` The corresponding albums will have `artistID` set to `NULL`

1.4 Updates

If one or more rows that already exist in the database need some of their components changed then we must use the `UPDATE` command. The general form of an update statement is:

```
UPDATE R SET <new-value assignment> WHERE <condition>;
```

Each new-value assignment looks like: `ATTRIBUTE = EXPRESSION`. If there is more than one assignment, they are separated by commas. The effect of this statement is to find all rows in `R` that satisfy the condition. Each of these rows is then modified by evaluating `EXPRESSION` and assigning the result to the corresponding `ATTRIBUTE`.

Lets modify the relation

MovieExec(name, address, cert#, netWorth)

by attaching the title `Pres.` in front of the name of every movie executive who is the president of a studio. The condition the desired rows satisfy is that their certificate number appear in the `presC#` component of some rows in the `Studio` relation.

Studio(name, address, presC#)

We express this update as:

```
UPDATE MovieExec
SET name ="Pres. " || name
WHERE cert# IN (SELECT presC# FROM Studio)
```

1.5 Creating Tables

```
CREATE TABLE <table-name> (
    <column-name1> <type> <column-constraint>,
    <column-name2> <type> <column-constraint>,
    <column-name3> <type> <column-constraint>,
    ...
    <table-constriaint>,
    <table-constriaint>,
    ...
);
```

`column-constraint` can specify many things, like `PRIMARY KEY`, `REFERENCES` for defining foreign keys, `DEFAULT` to set the default value if it is not provided at row insertion. This constraint is isolated to only one column.

`table-constraint` starts with `CONSTRAINT` and applies to multiple columns. For example, this can be useful when defining composite keys.

1.6 Index Definition

Create an index with the `CREATE INDEX` command:

`CREATE INDEX <index-name> ON <relation-name> (<attribute-list>);`

The *attribute-list* is the list of attributes of the relations that form the search key for the index. So, to define an index named *dept_index* on the *instructor* relation with *dept_name* as the search key, we write:

`CREATE INDEX` *dept_index* `ON` *instructor*(*dept_name*)`;`

If we wish to declare that the search key is a candidate key[1] , we add the attribute **UNIQUE** to the index definition:

`CREATE UNIQUE INDEX` *dept_index* `ON` *instructor*(*dept_name*)`;`

This declares *dept_name* to be a candidate key for *instructor* relation. If, at the time we enter `CREATE UNIQUE INDEX` command and *dept_name* is not a candidate key then the system will display an error message and the attempt will fail. If index-creation attempt succeeds, any subsequent attempt to insert a row that violates the key declaration will fail.

To drop an index use this command:

`DROP INDEX <index-name>;`

[1]Candidate Key – A Candidate Key can be any column or a combination of columns that can qualify as unique key in database. There can be multiple Candidate Keys in one table. Each Candidate Key can qualify as Primary Key.