Rule 1: The Information Rule:

This rule dictates that all information in our relational database is represented in one way only, via values in tables. We cannot store information in anything other than tables. These tables consist of attributes (columns) and tuples (rows). A table, then, is a logical grouping of related data within these rows and columns. Using a query such as the following, we are capable of accessing any and every piece of data in our table:

SELECT first_name FROM patient WHERE patient_ID=505;

This is on account of the requirement for primary keys. Each tuple in our table is associated with a unique instance of that table's primary key, thereby allowing us to retrieve data by calling the specific primary key value, the table name, and the attribute name.

Rule 2: The Guaranteed Access Rule:

SELECT address FROM patient WHERE Patient_ID=509;

We are able to retrieve a specific value (in this case, a patient's address) simply by knowing that patient's ID and the name of the table within which the data resides. This is due to the fact that the patient_ID attribute has been designation as a primary key within the patient table, thereby ensuring that our query will return the address associated with the patient's unique ID number and no other. Rule 2 is satisfied as, through a combination of table name, primary key, and column name, we would be able to access any atomic value within our database.
If we did not use the primary key in this instance and instead choose to use first name instead (e.g., SELECT Address FROM patient WHERE first_name='Frank';), we could return multiple address if there are multiple instances of the same first name within our table. As we cannot guarantee that each patient would have a unique first name, it would not be suitable primary key.

Rule 3: Systematic Treatment of Null Values:

A condition of the database's unpaid_bill view is that it displays outstanding fees which are greater than 0 (as 0 represents a fully paid bill). In the case that a patient has a bill but has not yet made a payment, a NULL value will be associated with their corresponding outstanding_fee (this is owing to the fact that the unpaid_bill view draws its data from the bill and payment tables, with outstanding_fees being derived from the payment table. While every payment will have a corresponding bill, not even bill will have a corresponding payment (i.e., when a patient has yet to make a payment, fully or partial), hence the potential NULL values).
If we wished to calculate the average outstanding_fee within the view, the following query would do so, regardless of the presence of any null values within our outstanding_fee attribute:

SELECT AVG(Outstanding_fee) FROM unpaid_bill;

Null values are not represented as zeroes or blank spaces in views. They also don't interfere with the calculation of our average outstanding_fee (which would be the case if null was treated as zeroes). Therefore, Rule 3 can be said to have been adhered to.

Rule 4: Dynamic Online Catalog based on the relational model:

SELECT * FROM information_schema.tables WHERE TABLE_SCHEMA='dentist';

The above query allows us to access the metadata of our table by displaying data about our data within our database. More specific information can also be sought out. The query below, for example, displays information regarding the patient table within the dentist database:

```
SELECT * FROM `COLUMNS` WHERE TABLE_NAME='patient';
```

Queries such as these demonstrate that the metadata about our relational database is contained at the same logical level as the regular data and is accessible though the same query language that is used when engaging with the regular data. The database can therefore be thought of as self-describing and in accordance with Rule 4.

Rule 5: The Comprehensive Data Sub Language Rule:

A relational database needs to support at least one language whose statements can handle tasks such as data definition, view definition, data manipulation, and integrity constraints. Examples of SQL queries carrying out these tasks are listed below:

- Data definition:

```
CREATE TABLE `patient` (
        `Patient_ID` int(11) NOT NULL,
        `First_Name` varchar(50) NOT NULL,
        `Last_Name` varchar(50) NOT NULL,
        `Address` varchar(50) NOT NULL,
        `Phone_Number` int(10) UNSIGNED ZEROFILL NOT NULL
);
```

- View definition:

```
CREATE VIEW unpaid_bill AS SELECT bill.Patient_ID, patient.first_name, patient.last_name,
patient.address, bill.Bill_ID, bill.charge_name, payment.Outstanding_fee, charge.fee
FROM (((bill LEFT JOIN payment on bill.bill_ID=payment.bill_ID)
INNER JOIN charge ON bill.Charge_name=charge.Charge_name)
INNER JOIN patient ON bill.Patient_ID=patient.Patient_ID)
WHERE payment.Outstanding_fee IS NULL OR payment.Outstanding_fee > '0.00';
```

- Data manipulation:

```
ALTER TABLE patient
ADD nationality varchar(255);
```

- Integrity constraints:

```
ALTER TABLE appointment
ADD CONSTRAINT appointment_fk1
FOREIGN KEY patient_ID REFERENCES patient(patient_ID);
```

Other requirements of the language are that it’s capable of supporting authorisation (e.g., grant and revoke) and transaction boundaries (e.g., begin, commit and rollback).

Rule 6: The View Updating Rule:

The views we create can be updated in the same manner that the tables upon which the views are based on can be updated. Take the unpaid_bill view as an example. The following query can be used to update the address of the patient with the ID number 520:

UPDATE unpaid_bill SET address='123 Fake Street' WHERE Patient_ID=520;

Patient 520's address will have been successfully updated and this can be verified using the following query:

SELECT address FROM patient WHERE Patient_ID=520;

The address value we updated in our view has indeed been applied in the corresponding section of the patient table.

Note that the following query will not execute as updates to views which affect more than one table cannot be carried out (in this case, the patient table (address) and payment table (outstanding_fee)):

UPDATE unpaid_bill SET address='123 Fake Street', Outstanding_fee='5555.00' WHERE Patient_ID=520;

Rule 7: High Level Insert Update and Delete Rule:

Along with being able to retrieve data, our database manipulation language must be able to insert, update or delete more than one row with a single query. Otherwise, manipulations would need to be carried out on a row-by-row basis. Take the charge table as an example. Say that the dentist wishes to increase the price of white fillings by €10 each. There are three types of fillings offered: 'White Filling (Large), 'White Filling (Regular), and 'White Filling (Small). Rather than updating each price individually, the following query would be able to increase all three prices in one go:

```
UPDATE charge
SET fee = fee + 10.00
WHERE charge_name LIKE 'White Filling%';
```

Rule 8: Physical Data Independence:

The way a relational database's data is physically stored is independent of the logical manner in which it is accessed. This separation of boundaries between the physical and logical aspects of the data means that any modifications made to the physical storage of a database, such as the hardware or disk storage methods, should have no impact on the user's ability to access the data.

Rule 9: Logical Data Independence:

Making changes to the logical structure of the database (e.g., adding a column to an existing table or adding an extra table to the database) does not alter our ability to access and manipulate the data. Logical data independence ensures that when we run the following query, the rest of the information regarding the table is preserved and accessible as it had been beforehand:

ALTER TABLE patient

ADD nationality varchar(255);

Adding a nationality attribute to our patient table does not impact on our ability to access the pre-existing attributes such as first_name or address.

Rule 10: Integrity Independence:

INSERT INTO bill (bill_ID, patient_ID, charge_name, payment_due_date) VALUES (NULL, 510, 'Check-up', '2023-10-10');

The above insertion will not run as attempting to insert a NULL value into the bill_ID column (the primary key for the bill table) goes against the concept of entity integrity, which requires all components of a primary key to have non-NULL values.

INSERT INTO bill (bill_ID, patient_ID, charge_name, payment_due_date) VALUES (10, 555, 'Gold Teeth', '2023-10-10');

An insertion such as the above will also not run as there is no patient with the ID number 555 and there is no charge name called Gold Teeth. Referential integrity dictates that for each distinct non-null foreign key value in a relational data base, there must exist a matching primary key value from the same domain. As patient_ID and charge_name as foreign keys in the bill table, the constraints in the database geared towards ensuring referential integrity prevent this insertion from occurring.

The above demands queries demonstrate that the database is in keeping with the demands of Rule 10.

Rule 11: Distributed Independence:

Even if we were aware of exactly where our database was located, SQL queries do not have any syntax to locate the physical storage of the database. Our queries simply refer to the database on a logical level, with the user offered no indication of where the data is physically stored. A database's data can even be split amongst several physical storage systems and still the user would not be aware of its multiple locations. With the advent of cloud computing, this rule has taken on greater relevance in recent years, with data now being housed potentially anywhere across the globe.

Rule 12 Non Subversion Rule:

There are low-level languages which can be used to alter databases in a way which aren't governed by the constraints that higher-level languages such as SQL would adhere to. Although it has its functions, the use of these low-level languages does leave the door open for potential integrity errors or illicit activity to occur. Rule 12 dictates that no language should be able to bypass integrity constraints or violate the rules of our database in any way. Although Rule 12 does not do away with low-level languages, but it does demand that they be held to the same constraints as our other query languages.