Foundations of Computer Science - SQL

Contents

1 Relational Databases

1.1 Introduction

The relational DBs are organized in tables that can be linked together. A table in the relational model represents, itself, a relation. For example in the following database each row describes a single book:

ISBN	**Title**
978-1-449-30321-1	Scaling MongoDB
978-1-491-93200-1	Graph Databases
978-1-449-39041-9	Cassandra: The Definitive Guide
007-709500-6	Database Systems

Table 1: Some sample books

The data in a relational model are organized into columns and each entry(cell) contains a **SINGLE** piece of data, so the main problem of relational model is to handle data with multiple elements in a single column for example a book with two authors.

The solution proposed by the relational model is to link two tables and to do so we need an identifier ID for each row. Here is an example of two linked tables.

book_id	**ISBN**	**Title**
1	978-1-449-30321-1	Scaling MongoDB
2	978-1-491-93200-1	Graph Databases
3	978-1-449-39041-9	Cassandra: The Definitive Guide
4	007-709500-6	Database Systems

Table 2: Books Table with ID

author_id	Name	Surname
1	Ian	Robinson
2	Kristina	Chodorow
3	Riccardo	Torlone
4	Paolo	Atzeni
5	Stefano	Ceri
6	Stefano	Paraboschi
7	Eben	Hewitt

Table 3: Authors Table with ID

To connect these two tables we use a "relation" or "join" table. In the example the "join" table is the following:

book_id	author_id
2	1
3	7
1	2
4	4
4	5
4	6
4	3

Table 4: BookAuthors Table

In the Table 4 we can see the need of using an ID (named Foreign Key) to identify each row, so it can be referenced uniquely. In some cases we might not need a third table (join table): one of these cases is the "One-to-Many" relationship. The One-to-Many relationship also requires that rows in tables have unique IDs, but unlike the join table used in Many-to-Many relationship, the table with the many side of data has a column reserved for the IDs of the one side of data.

The IDs used to uniquely identify the rows described in the tables are called "Primary Keys"(PK). If this primary key is used in another table it is called "Foreign Key"(FK) and it is usually not unique in the new table. The purpose of the foreign key is to link the two tables in one-to-many relationship.

Usually the primary key generation process is left to the RDBMS (it's safer this way), which automatically generates the key and usually it is an ordinary integer. For the join tables, the primary key is a combination of the foreign keys. A primary key comprised of more than one attribute is called a "Composite Primary key"(CPK).

1.2 Normal Forms

The First Normal Form (1NF) says that the types must be atomic (the table must be flat) so we cannot have a cell with multiple data(multi-list) in it. This causes some anomalies such as:

- Redundancy: Same data is repeated multiple times. For example, in a university timetable(we will use this example for the whole list), if every course is in only one room we have two attributes: one with the Course and one with the Room, which is redundant.
- Update anomaly: when we update a value of a cell. For example if we update room name in a cell we don't automatically have an update on all the cell containing that room or the course.
- Delete anomaly: if all the data using a specific information is dropped we lose that specific information too. For example if everyone in drops a class we lose what room the class is in.
- Insert anomaly: we can't use a specific information without some other data using it. Example is similar to delete anomaly: we cannot reserve a room for a course without having students.

To overcome these anomalies other Normal Forms have been created. To introduce them first we need to introduce a definition.

Def. Functional Dependency: Let A, B be sets of attributes, we write $A \to B$ or say A functionally determines B if, for any tuple t_1 and t_2:

$$t_1[A] = t_2[A] \Rightarrow t_1[B] = t_2[B]$$

and we call $A \to B$ a **functional dependency**.

In other words, whenever two tuples agree on A they agree on B. A functional dependency is a form of constraint.

Remark. It is easy to disprove a functional dependency, you need only a instance violating it, but to prove it we need to check every valid instance.

We define the **superkey** as a set of attribute $A_1, \ldots, A_n$ such as for any other attribute B in the relation R, we have $\{A_1, \ldots, A_n\} \to B$. In other words a super key is a set of one or more attributes (columns), which can uniquely identify a row in a table. A **key** is a minimal superkey (only one attribute) i.e. no subset of a key is a superkey.

The idea to have a normalized database is to search for "bad" FDs: if there are any, then decompose the table into sub-tables (linked usually) until no more bad FDs. When done the database schema is normalized. The normal forms distinguish from each other depending on what they define as a "bad" FD.

For the Boyce-Codd Normal Form (BCNF) defines $X \to A$ as a "good" FD if X is a (super)key and "bad" otherwise. BCNF is a simple condition for removing anomalies from the relations: A relation R is in BCNF if $\{A_1, \ldots, A_n\} \to B$ is a non-trivial FD in R then $\{A_1, \ldots, A_n\}$ is a superkey for R.
The problem with BCNF is that to enforce a FD, we must reconstruct original relation on each insert. It's solution is usually a trade-off between redundancy/data anomalies and FD preservation.

2 SQL

2.1 Introduction

SQL stands for Structured Query Language and it is a standard language for querying and manipulating data. It is a very high-level programming language and it is very well optimized.

SQL is a:

- Data Definition Language (DDL):
  It defines relation *schema* and creates/alters/deletes tables and their attributes.
- Data Manipulation Language (DML):
  It Inserts/deletes/modifies tuples in tables and queries one or more tables.

A **relation** (or **table**) in SQL is a multi-set of tuples having the attributes specified by the schema.As **multiset** we mean that is an unordered list (so multiple duplicates instances are allowed) and an **attribute** is a typed data entry present in each tuples in the relation. An attribute must have an atomic type in standard SQL. The atomic types are:

- Characters: CHAR(20), VARCHAR(50)
- Numbers: INT, BIGINT, SMALLINT, FLOAT
- Others: MONEY, DATETIME, etc.

A **tuple (row)** is a single entry in the table having the attributes specified by the schema.

The **schema** of a table is the table name, it's attributes and their types, a key is an attribute whose values are unique.

A **key** is a minimal subset of attributes that acts as a unique identifier for tuples in a relation. A key is an implicit constraint on which tuples can be in the relation: so if two tuples agree on the same value of the key, then they must be the same tuples.

If some informations are missing or not known SQL shows a NULL value. To check if a value is NULL the "IS NULL" SQL command is used. SQL offers the possibility to constrain a column to be NOT NULL or supports other constraints such as maximum number of values per attributes. Thanks to the schema and constraints, the databases understand the semantics of the data.

2.2 Constraints in SQL

A constraint is a relationship among data elements that the DBMS is required to enforce. The triggers are executed when a specified condition occurs. Kinds of constraints are:

- (Foreign) Keys
- Value-Based constraint (constrain a certain value)
- Tuple-Based constraint (relationship among components)
- Assertion (any SQL boolean expression)

When creating a table, we also generate a key using a constraint of UNIQUE or PRIMARY KEY. To declare a Foreign Key, the keyword REFERENCES is placed after an attribute or as an element of the schema and the referenced attribute must be declared (in the table it belongs to) either as a PRIMARY KEY or UNIQUE. Some examples:

```
--For the single attribute key
CREATE TABLE table_1(
   attribute_1 CHAR(20) UNIQUE,
   attribute_2 VARCHAR(20)
);

--For the multi-attribute key
CREATE TABLE table_2(
   attribute_1 CHAR(20),
   attribute_2 VARCHAR(20),
   attribute_3 REAL,
   PRIMARY KEY (attribute_1,attribute_2)
);

--Foreign key reference example
CREATE TABLE table_1(
attribute_1 CHAR(20) UNIQUE,
attribute_2 VARCHAR(20)
);
CREATE TABLE table_2(
attribute_3 CHAR(20),
attribute_4 VARCHAR(20) REFERENCES table_1(attribute_1),
attribute_5 REAL,
);

--Foreign key as schema element
CREATE TABLE table_1(
attribute_1 CHAR(20) UNIQUE,
attribute_2 VARCHAR(20)
);
CREATE TABLE table_2(
attribute_3 CHAR(20),
attribute_4 VARCHAR(20),
attribute_5 REAL,
FOREIGN KEY(attribute_4) REFERENCES table_1(attribute_1)
);
```

Some possible violations may arise from the presence of a foreign key constraint:

1. Insert or update operations onto table_2, can introduce values not found in table_1.
2. Delete or update operations onto table_1, causing some tuples of table_2 to be disrupted.

In the former case, such operation is simply rejected. In the latter case, one of the following policies has to be chosen:

1. Default: Rejects the modification alike in the former case.
2. Cascade: Applies the same changes in table_2 (i.e. in case of deletion (update) of a row of table_1, delete (update) all the related tuples of table_2).
3. Set NULL: Change the involved element to NULL.

The relative declarative syntax is (located after foreign key declaration):
[UPDATE,DELETE] [SET NULL CASCADE] .
If not declared the default option (reject) is intended. Here an example:

```
CREATE TABLE table_1(
attr_1 CHAR(20) UNIQUE,
attr_2 VARCHAR(20)
);
CREATE TABLE table_2(
attr_3 CHAR(20),
attr_4 VARCHAR(20),
attr_5 REAL,
FOREIGN KEY(attr_4)
   REFERENCES table_1(attr_1)
   ON DELETE SET NULL
   ON UPDATE CASCADE
);
```

Some other constraint are Attribute-Based checks to constraint on the value of a particular attribute. In this case add CHECK(<condition>) to the declaration for the attribute. We might use the name of the attribute in the condition but any other relation or attribute name must be in a subquery. The checks are performed only when a value for that attribute is inserted or updated. It can be added as a relation-schema element(like for the foreign key).

The last analysed constrain is the Assertion, a database-schema element. It is syntactically defined by:
CREATE ASSERTION <name> CHECK (<condition>);
the condition may refer to any relation or attribute in the database schema.

For example, if there are two tables, Bars and Drinkers, and we want to assert that there cannot be more bars than drinkers, the code is:

```
   CREATE ASSERTION FewBar CHECK (
   (SELECT COUNT(*) FROM Bars) <=
   (SELECT COUNT(*) FROM DRINKERS)
   );
```

In principle, we must check the assertion after every modification to any relation of the database but a clever system can observe that only certain changes could cause a given assertion to be violated. Problem is the DBMS often can't tell when they need to be checked. The Triggers let the user decide when to check for any condition. An example is when using a foreign key constraint, instead of rejecting insertions with unknown elements, a trigger can add that new element to the original table with other attributes set as NULL.

2.3 SQL Data Manipulation Operations

2.3.1 Basic Commands for a single table

To project a table we use `SELECT FROM`:

```
SELECT <attributes>
FROM <one or more relations>
WHERE [optional]<conditions>
```

After the keyword `SELECT`, we can use * to select all the attributes.

Remark. Commands are case insensitive but the values are case sensitive. Also, when using condition on strings we have to use single quotes 'value' and not the double ones.

Assume we have a table of authors named Authors with the following schema:

id	last_name	first_name	DoB	income	genre
INT	CHAR	CHAR	DATETIME	REAL	CHAR

To handle duplicated values, the keyword `DISTINCT` can be used right after `SELECT`. Associated to the keyword `WHERE` there might be one or more conditions (using `AND` or `OR`).

For conditions on strings we have the following syntax: `%` means "any string", _ (underscore) means "any character" and when using them we need to use `LIKE` in `WHERE` clause. To order them we have to use `ORDER BY (DESC) <attribute>`, `DESC` for descending order by default is ascending, here is an example using all the points:

```
--Distinct Genres of Authors whose name start with S and the third letter is a
SELECT DISTINCT genre
FROM Authors
WHERE first_name LIKE 'S_a%'
ORDER BY DESC genre;
```

To count the number of rows use `count(*)` in the `SELECT` clause. To count the number of values not null in a specific column use `count(column1,column2)` in the `SELECT` clause. Some of the other operations that can be done in the `SELECT` clause are `AVG, SUM, MIN, MAX`.

Those operations are heavily used together with `GROUP BY`. `GROUP BY` is useful to aggregate the data, if we want to calculate the average income for each genre we do it as follows:

```
SELECT genre, AVG(income)
FROM Authors
GROUP BY genre;
```

Attention: we can have in `SELECT` only the attributes present in `GROUP BY` or other attributes with some aggregate operation only, never by themselves. All the operation mentioned were considered in the case of one table.

2.3.2 Basic Commands on two (or more) tables [JOIN/WHERE]

Now let's focus our attention to the case of two tables. Assume having the Authors table, Books table, a BooksAuthors join table and an Editions table with the following schema:

id	last_name	first_name	DoB	income	genre
INT	CHAR	CHAR	DATETIME	REAL	CHAR

Authors Table

id	title	ISBN
INT	CHAR	BIGINT

Books Table

book_id	author_id
INT	INT

BookAuthors Table

edition_id	book_id	date_of_publication	edition_number
INT	INT	INT	INT

Editions Table

We can use the foreign key to connect two tables either in `FROM` using a `JOIN` or in `WHERE`. Here the two methods:

```
--WHERE version to find author of book_id = 1
SELECT first_name, last_name
FROM Authors, BooksAuthors
WHERE BooksAuthors.author_id = Authors.id
   AND book_id = 1;
-- JOIN version to find author of book_id = 1
SELECT first_name, last_name
FROM Authors JOIN BookAuthors
   ON BooksAuthors.author_id = Authors.id
WHERE book_id = 1;
--JOIN version is preferred when only 2 tables are involved
```

Some results of the `JOIN` might be duplicated if the foreign key is displaced more than one time in the table. Thus, it might be useful to use the keyword `DISTINCT`. For example, how to find who has written a book whose ISBN ends with 5 and has last name with exactly 5 characters? We need to use 3 table and the `WHERE` version of linking tables:

```
SELECT Books.id, ISBN, Authors.id, last_name, first_name
FROM Books, Authors, BooksAuthors
WHERE Books.id = BooksAuthors.book_id AND
   BooksAuthors.author_id = Authors.id AND
   last_name LIKE '_____' AND
   isbn LIKE "%5"
```

Whenever it is annoying to frequently write down the table name, **aliases** can be used. For example, let us rename `Books -> b, Authors -> a` and `BooksAuthors -> ba` and rewrite the previous query in the light of aliases:

```
SELECT b.id, ISBN, a.id, last_name, first_name
FROM Books b, Authors a, BooksAuthors ba
WHERE b.id = ba.book_id AND
   ba.author_id = a.id AND
   last_name LIKE '_____' AND
   isbn LIKE '%5'
```

The `JOIN` semantics between two tables `T1` and `T2` is a cross product between `T1` and `T2`, followed by a selection of rows satisfying the `WHERE` clause, and eventually a projection on the columns specified in the `SELECT` clause. Assume that an author has not written any books, when we do join between `Authors` and `BooksAuthors` that author will not appear. If we want to force his presence, we need to use `LEFT JOIN` with `Authors` on the `LEFT`. An example of the outer `JOIN` (which is a union and the inner join is a intersection):

```
SELECT Authors.id, last_name, first_name, Books.id, ISBN
FROM Books, Authors LEFT JOIN BooksAuthors
   ON Authors.id = BooksAuthors.author_id
WHERE Books.id = BooksAuthors.book_id
--Now the result will also show the authors with Books.id and NULL ISBN value.
```

We can also rename the projected columns using aliases. An example where we count the number of books

written by an author, using an outer join to include also the authors with zero associated books:

```
SELECT Authors.id, last_name, first_name ,
   count(Books.id) as number --Here is how to rename a column
FROM Books, Authors LEFT JOIN BooksAuthors
   On Authors.id = BooksAuthors.author_id
WHERE Books.id = BooksAuthors.author_id
GROUP BY Author.id, last_name, first_name; --Here we need all the columns written in the SELECT
```

2.4 SET Operations

The intersection of two or more tables can be obtained by using the INTERSECT clause: it acts as the regular (mathematical) intersection between two sets (where sets correspond to the rows). It is rarely used, since it is often replaced by a more complex `WHERE` condition (for example using `AND`). An example on how to find `Authors` with `Income`>90000 and with the last name containing an "o":

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000
INTERSECT
SELECT last_name, first_name
FROM Authors
WHERE last_name LIKE '%o%'

--Equivalent WHERE / AND condition
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000 AND last_name LIKE '%o%'
```

We can also do the `UNION` between two sets, also this one is rarely used because it can be replaced by a more complex `WHERE` condition if no duplicates are there. To do union with duplicates `UNION ALL` can be used. Here is an example to find authors with `Income`>90000 or name containing a "o":

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000
UNION
SELECT last_name, first_name
FROM Authors
WHERE last_name LIKE '%o%'

--If no duplicates equivalent is the WHERE - OR
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000 OR last_name LIKE '%o%'

--Union with duplicates
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000
UNION ALL --Here is the difference
SELECT last_name, first_name
FROM Authors
WHERE last_name LIKE '%o%'
```

One practically and commonly used command relative to the set operations is the EXCEPT: it is a

difference between two sets (however, it can be replaced with a nested query):

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000
EXCEPT
SELECT last_name, first_name
FROM Authors
WHERE last_name LIKE '%o%'
```

The code above will return all the author with `Income`>`90000` and the last name not containing a "`o`".

2.5 Nested Queries

As aforementioned, we can replace some of the SET operations with nested queries.

To replace `INTERSECT` we can proceed as follows:

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000 AND last_name IN
   (SELECT last_name, first_name
   FROM Authors
   WHERE last_name LIKE '%o%')
```

For the `EXCEPT`:

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000 AND last_name NOT IN
   (SELECT last_name, first_name
   FROM Authors
   WHERE last_name LIKE '%o%')
```

Nested queries has many other uses, for example finding the author with maximum income:

```
SELECT last_name, first_name
FROM Authors A1
WHERE A1.income >= ALL (SELECT A2.income
                        FROM Authors)
```

Another command that can be used after `WHERE` is the `ANY` (instead of `ALL`). In this case the result will include every row satisfying the condition for at least one other row of the nested query (if `ALL` is used, the result is the set of rows that satisfies the condition for all of the rows in the nested query).

In SQLite the command ALL is replaced by MAX.