SQL Odds and Ends

In this section we briefly discuss other important SQL-related concepts that we don't have time to explore more fully.

- **Indexes** are used in databases to speed up queries.
- **Views** are essentially stored SELECT queries.
- **ORM** allows users in programming languages like Python to interact with a database using objects and their relations rather than direct SQL queries.

Reading

- Introduction to SQL, sections 4.11, 4.12, chapters 20, 21[1]
- Wikipedia entry on Indexes[2]
- Wikipedia entry on Views[3]
- SQLAlchemy ORM tutorial[4]

Reading questions

- What are the advantages of using indexes? What are the disadvantages?
- Can we use in an index columns coming from different tables?
- Does the order of columns in an index matter? Why?
- Can we add an index on a table with existing data?
- How can we remove an index after it's created?
- Does SQL generate certain indexes automatically?
- Are the rows in a view actually stored on a physical storage device?
- Come up with example applications of Views, at least 2 for each of the different usages of views.

Notes

Indexes

An **index** is essentially an extra structure that is stored along with your database table, and facilitates certain searches. Each index is essentially tailor-made for a particular kind of search that uses a certain set of columns.

An index effectively maintains an efficient arrangement of the rows according to their values in a certain column. Think of it similar to the index at the back of a book: It offers you quick access to occurences of important words in the text. Similarly,

[1] http://learning.acm.org/books/book_detail.cfm?id=1208031&type=safari
[2] https://en.wikipedia.org/wiki/Database_index
[3] https://en.wikipedia.org/wiki/View_(SQL)
[4] http://docs.sqlalchemy.org/en/latest/orm/tutorial.html

an index on the variable “age” would for instance offer you quick access to the rows corresponding to students of a particular age, rather than having to go through all rows to find them.

This power comes at a cost: Every time you want to add a new entry or change an existing entry, the corresponding index needs to change to accomodate this new value. As a result, *indexes make the INSERT, DELETE and UPDATE queries slower, in order to make certain SELECT queries faster*.

As a concrete example, consider the following table:

```
CREATE TABLE `dummy_vals` (
   `id`          bigint(20) NOT NULL       AUTO_INCREMENT,
   `created`     timestamp  NULL           DEFAULT CURRENT_TIMESTAMP,
   `anint`       INT(11),
  PRIMARY KEY (id)
);
```

This table has an auto-generated key called id, and it is also a primary key. Any primary key is automatically part of an index. This means that any queries that address the table via the id will be very efficient. In order to work with the table, we will add a fairly large number of data to it. This uses a procedure, and you don’t need to understand all the details:

```
DELIMITER $$
CREATE PROCEDURE generate_data()
BEGIN
  DECLARE i INT DEFAULT 0;
  WHILE i < 100000 DO
    INSERT INTO `dummy_vals` (`datetime`,`anint`) VALUES (
      FROM_UNIXTIME(UNIX_TIMESTAMP('2014-01-01 01:00:00')+FLOOR(RAND()*31536000)),
      ROUND(RAND()*50,2)
    );
    SET i = i + 1;
  END WHILE;
END$$
DELIMITER ;

CALL generate_data();
```

This generates 50000 “dummy” entries. As we say, these are by default indexed based on the id. So we could do quite efficiently a query like the following:

```
SELECT SQL_NO_CACHE *
FROM dummy_vals
WHERE id > 25000 AND id < 29000
ORDER BY id;
```

This takes a mere 0.004-0.007 seconds to execute. But a query on the integer field, matching about the same number of cases, would be slower:

```
SELECT SQL_NO_CACHE *
FROM dummy_vals
WHERE anint > 250 and anint < 270
ORDER BY anint;
```

This takes 0.03-0.5 seconds to execute, about 7-8 times slower. If we were to add an index to it, like so:

```
ALTER TABLE `dummy_vals`
ADD INDEX `anint_index` (anint);
```

Then the same anint-based query as above takes only 0.005-0.008 seconds, very close to the primary key-based query.

Indexing on multiple columns only helps if you use all the columns, or at least the first few columns. For example, if we take out that last index and add a new one:

```
ALTER TABLE `dummy_vals`
DROP INDEX `anint_index`,
ADD INDEX `combo_index` (created, anint);
```

Then the anint-based query is again slow, because this index requires a created clause before it can be used. Reversing the order helps:

```
ALTER TABLE `dummy_vals`
DROP INDEX `combo_index`,
ADD INDEX `combo_index2` (anint, created);
```

Now the anint-based query is again fast.

Some takeaways:

1. Indexes *may* speed up SELECT queries.
2. Indexes take up additional storage.
3. Indexes slow down CREATE, UPDATE and DELETE queries, as they need to be updated along with the database.
4. Having many indexes may slightly slow down a SELECT query as the engine has to choose which index to use for the specific query.
5. Indexes typically benefit queries that would return a relatively small number of hits compared to the table size. If almost all values are to be returned, an index won't help much.

Views

Views are a bit like "virtual tables". A view is associated with a SELECT query, and it effectively stores the query for future use. In the future you can directly try to access the view as if it was a table, without needing to explicitly run the query again. There are two main uses of this:

1. For frequently used queries that bring many tables together.
2. For computations relying on existing data. For instance computing the number of credits and gpa that a student has based on their course records. We could have this dynamically computed with a view rather than updating the student table every time a new grade is inserted.

3. When we want to restrict access. For instance we might need to provide someone access to student names and addresses, but not other more sensitive information stored in their account, like their GPA, number of credits, account holds etc. We can create a view that only contains specific columns from a table, and then give select users view access to this view, but not the table it came from.

Read more about views in the links above.

ORM

We now turn to an important topic that is more higher-level, namely **Object-Relational Mapping**. The idea is simple. Assume we are working in an object-oriented language like Python.

- We create custom Python classes to represent the objects we want to persist/store.
- We describe to the SQL engine (SQLAlchemy in our case) how these Python objects would correspond to table entries in the database.
- We specify **relationships** between objects of these classes, that effectively correspond to foreign keys. There is typically an **owning side** to each relationship. We change the relatioship by making a change on the object on that side. There are a number of different possible relationships:
  - one-to-one relationships. They also come in two variables, where the lack of a relationship is allowed and where it is not. Imagine an "order" object. It may or may not be associated to a unique "invoice" object. An order may or may not have an associated invoice object, and an invoice object can only be associated with one order object.

    Another example is relating a student-class enrollment object to an evaluation object, should we choose to do so. An evaluation is associated to a unique enrollment, and an enrollment is associated to at most one evaluation (but none if the student has not completed one).

    In SQL terms, this association is achieved via a FOREIGN KEY that is also UNIQUE, and possibly NOT NULL.
  - one-to-many relationships. These relate an object of one class with possibly many objects of another class. Almost universally the owning side is the "many" side. The "many" objects are typically stored in a collection structure like a Python list.

    As an example, a course has a one-many relationship with its sections. This is represented by a foreign key on the sections table pointing to the corresponding course.

    In SQL these relationships are achieved via a standard FOREIGN KEY.
  - many-to-many relationships. Each object of the one class may be related to many objects of the other class, and vice versa. For example in a mailing

center we might have accounts and addresses. An account may be associated with many addresses, and an address may correspond to multiple accounts.

In SQL these kinds of relationships are typically achieved via a third "intersection" table that contains pairs of ids from the two tables corresponding to the classes. It effectively turns the many-to-many relationship in two many-to-one relationships through the intermediary intersection table.

Some times we go further and explicitly make this its own entity/class, when we want to store further information along. For example students have a many-to-many relationship with course sections, but this relationship becomes explicit via the enrollment table, as we want to store further information along, namely whether the student has completed an evaluation yet or not.

- From that point on we manipulate these Python objects, creating new, deleting some or changing some, and let the SQL engine decide how persist these objects on the database and what queries to perform to do so.
- We (almost) never write our own SQL queries this way.
- The interaction of the ORM with that database is often done via **Sessions**. The ORM system might accumulate many changes, and only talk to the database when needed. For instance, you can:
  - Create a new object and *add* it to the session.
  - Use the session to perform a query and return some objects from the database.
  - Change an object's attribute values, and trust that the session will create a suitable UPDATE query.
  - *Commit* all the changes from the session.
  - *Rollback* changes from the session to discard them.
- A process known as *identity map* ensures that objects we retrieve via queries are really the same objects, even if they were loaded via multiple instructions. When we for instance search for the user "Haris" and store it in the variable `user1`, then later on search for the user "Haris" again and store it in the variable `user2`, then the objects stored in the variables `user1` and `user2` are in fact the exact same object, and not two different copies.