Databases

Ivin Lee

October 8, 2020

1 Lecture 1

Database Systems in course

- HyperSQL
- DOCtor Who
- Neo4j

Interface
This course will deal with Database Management Systems from the perspective of an application developer, and will teach the interfaces of these systems.

Database Management System
Implements CRUD operations

- Create
- Read
- Update
- Delete

Implements ACID transactions for concurrent updates

- Atomicity: either all actions carried out or none carried out
- Consistency: database is consistent before and after transactions
- Isolation: every transaction happens independently of other transactions: allows for concurrency

- Durability: if transaction happens successfully, then its effects will persist

SQL
HyperSql - just a java file, overhead of using is low

NoSQL
Not only SQL - non-relational, distributed, open-source, horizontally scalable.

- Scalability: data stored on multiple machines
- Fault Tolerance: service can survive failure of some machines
- Lower Latency: data located closer to widely distributed users

Distributing Data

- Replication
- Partitioning

CAP Concepts

- Consistency: all reads return data up-to-date
- Availability: all clients can fine some replica of the data
- Partition Tolerance: system can still operate despite loss or failure of part of the system

Cannot achieve all of these, some balance.

CAP Principle
In a highly distributed system,

- Assume that network partitions and other connectivity problems will occur
- Implementing ACID transactions is very difficult and slow
- trade-off between availability and consistency

Eventual consistency - if update activity ceases, then the system will reach a consistent state.

BASE

- BA: basically available
- S: soft state
- E: eventual consistency

This is an area of ongoing research.

Polyglot Persistence
Using various Database Management Systems to manage different sets of data

Conclusion: there will be a lot of changes in Database Management Systems.

2 SQL Commands

Aggregate Commands

```
select position, count(*) as total
from has_position
group by position
order by total desc;
```

SQL creates new rows where each row 'contains' all the rows which have the same `position`. Then, each column in the new table can only be aggregate commands like `count(*)`, `min(...)`.

Join

```
select title, genre
from movies
join has_genre on has_genre.movie_id = movies.movie_id
join genres on genres.genre_id = has_genre.genre_id
where year = 2017
limit 20;
```

```
TITLE                    GENRE
-----------------------  ---------
Wonder Woman             Fantasy
Wonder Woman             Action
Wonder Woman             Adventure
It                       Horror
The Greatest Showman     Drama
The Greatest Showman     Musical
```

```
The Greatest Showman    Biography
The Foreigner           Action
The Foreigner           Thriller
A Dog's Purpose         Drama
A Dog's Purpose         Comedy
A Dog's Purpose         Adventure
Blade Runner 2049       Drama
Blade Runner 2049       Action
Blade Runner 2049       Mystery
Spider-Man: Homecoming  Action
Spider-Man: Homecoming  Sci-Fi
Spider-Man: Homecoming  Adventure
Coco                    Animation
Coco                    Adventure
```

Joining duplicates entries. Since some movies might have more than 1 genre, identifying these movies requires work as SQL evaluates each row separately so `select * from movies join genres where genre = genre_1 and genre = genre_2` doesn't work.

Multiple Joins

The way around this problem is to just join the `genres` table multiple times so each row has two `genres` columns!

```
select title, year, rating, votes
from movies as m
join has_genre as hg1 on hg1.movie_id = m.movie_id
join has_genre as hg2 on hg2.movie_id = m.movie_id
join genres as g1 on g1.genre_id = hg1.genre_id
join genres as g2 on g2.genre_id = hg2.genre_id
where m.votes > 100000 and g1.genre = 'Romance' and g2.
   genre = 'Comedy'
order by votes desc;
```

If you are discerning enough, you'll realise that there are actually 4 rows for each original row! This results in a bigger search time complexity. Example rows are shown below.

```
TITLE                 GENRE      GENRE
--------------------  ---------  ---------
In the Mood for Love  Drama      Drama
In the Mood for Love  Drama      Romance
In the Mood for Love  Romance    Drama
In the Mood for Love  Romance    Romance
Chicken Run           Animation  Animation
```

```
Chicken Run             Animation   Comedy
Chicken Run             Animation   Adventure
Chicken Run             Comedy      Animation
Chicken Run             Comedy      Comedy
Chicken Run             Comedy      Adventure
```

Nesting

Possible to nest queries, i.e. search a sample space generated from another select statement.

```
select m1.title, m1.year, g.genre, m1.rating, m1.votes
from movies as m1
join has_genre as hg on hg.movie_id = m1.movie_id
join genres as g on g.genre_id = hg.genre_id
where m1.votes > 100000 and (not (g.genre = 'Romance' or
    g.genre = 'Comedy'))
      and m1.movie_id in
         (select m2.movie_id
         from movies as m2
         join has_genre as hg1 on hg1.movie_id = m2.
            movie_id
         join has_genre as hg2 on hg2.movie_id = m2.
            movie_id
         join genres as g1 on g1.genre_id = hg1.genre_id
         join genres as g2 on g2.genre_id = hg2.genre_id
         where g1.genre = 'Romance' and g2.genre = '
            Comedy')
order by m1.votes desc
limit 10;
```

Views

Instead of nesting select statements, it is possible to create a `view`, which can be used in the same way as a table.

```
drop view if exists romcom_ids;

create view romcom_ids as
    select m.movie_id as movie_id
    from movies as m
    join has_genre as hg1 on hg1.movie_id = m.movie_id
    join has_genre as hg2 on hg2.movie_id = m.movie_id
    join genres as g1 on g1.genre_id = hg1.genre_id
    join genres as g2 on g2.genre_id = hg2.genre_id
    where g1.genre = 'Romance' and g2.genre = 'Comedy';

```

```
select m.title, m.year, g.genre, m.rating, m.votes
from romcom_ids as r
join has_genre as hg on hg.movie_id = r.movie_id
join genres as g on g.genre_id = hg.genre_id
join movies as m on m.movie_id = r.movie_id
where m.votes > 100000 and (not (g.genre = 'Romance' or
   g.genre = 'Comedy'))
order by m.votes desc
limit 10;
```

Three-valued Logic

SQL allows columns to be `null`. This means that statements can evaluate to `true`, `false` and `null`.

```
select count(*)
from people
where deathYear = null

returns 0
```

`* = null` returns null, and `select ... where` only returns records when the `where` clause returns `true`.

To solve this, SQL introduced `is null`.

```
select count(*)
from people
where deathYear is null

returns 5919
```

Left Join

The way `join` works is to strike off the rows which cannot match the other table. However, in some situations we would like to preserve these rows, and we can do so using `left join` or `right join`.

```
select name, position
from people as p
join has_position as c on p.person_id = c.person_id
join movies as m on c.movie_id = m.movie_id
where title = 'Silver Linings Playbook';
```

```
NAME                POSITION
------------------  --------
Robert De Niro      actor
Bruce Cohen         producer
```

```
Bradley Cooper      actor
Donna Gigliotti     producer
Jonathan Gordon     producer
David O. Russell    director
Jacki Weaver        actor
Jennifer Lawrence   actor
Matthew Quick       writer
```

Joining this table with the roles table gives:

```
select name, position, role
from people as p
join has_position as c on p.person_id = c.person_id
join movies as m on c.movie_id = m.movie_id
join plays_role as r on r.movie_id = m.movie_id and r.
   person_id = c.person_id
where title = 'Silver Linings Playbook';
```

```
NAME                POSITION  ROLE
------------------  --------  -------
Robert De Niro      actor     Pat Sr.
Bradley Cooper      actor     Pat
Jacki Weaver        actor     Dolores
Jennifer Lawrence   actor     Tiffany
```

So some entries in the first table are not shown because only actors have roles. In this case, using a left join will help:

```
select name, position, role
from people as p
join has_position as c on p.person_id = c.person_id
join movies as m on c.movie_id = m.movie_id
left join plays_role as r on r.movie_id = m.movie_id and
    r.person_id = c.person_id
where title = 'Silver Linings Playbook';
```

```
NAME                POSITION  ROLE
------------------  --------  -------
Robert De Niro      actor     Pat Sr.
Bruce Cohen         producer  [null]
Bradley Cooper      actor     Pat
Donna Gigliotti     producer  [null]
Jonathan Gordon     producer  [null]
David O. Russell    director  [null]
Jacki Weaver        actor     Dolores
Jennifer Lawrence   actor     Tiffany
```

```
Matthew Quick       writer     [null]
```

Note that `null` is displayed as `[null]` to distinguish between the empty string and actual `null`.

Select Distinct

SQL is based on multisets of records (i.e. sets containing duplicates, also called bags) so the following query will result in multiple repeated rows:

```
select r1.role as role, m1.title as title, m1.year as
   year
from plays_role as r1
join plays_role as r2 on r2.person_id = r1.person_id
join movies as m1 on m1.movie_id = r1.movie_id
join movies as m2 on m2.movie_id = r2.movie_id
join people as p on p.person_id = r1.person_id
where p.name = 'Noomi Rapace'
      and r1.role = r2.role
      and m1.movie_id <> m2.movie_id
order by m1.title, r1.role, m1.year;
```

```
ROLE               TITLE
                                        YEAR
----------------   ------------------------------------
    ----
Lisbeth Salander   The Girl Who Kicked the Hornet's Nest
    2009
Lisbeth Salander   The Girl Who Kicked the Hornet's Nest
    2009
Lisbeth Salander   The Girl Who Played with Fire
            2009
Lisbeth Salander   The Girl Who Played with Fire
            2009
Lisbeth Salander   The Girl with the Dragon Tattoo
          2009
Lisbeth Salander   The Girl with the Dragon Tattoo
          2009
```

They keyword `select distinct` collapses the repeated rows. Note that the table with repeated listings is first generated, then the repeats removed, so there is no improvement in query time (I believe?)

```
select distinct r1.role as role, m1.title as title, m1.
   year as year
from plays_role as r1
```

```
join plays_role as r2 on r2.person_id = r1.person_id
join movies as m1 on m1.movie_id = r1.movie_id
join movies as m2 on m2.movie_id = r2.movie_id
join people as p on p.person_id = r1.person_id
where p.name = 'Noomi Rapace'
      and r1.role = r2.role
      and m1.movie_id <> m2.movie_id
order by m1.title, r1.role, m1.year;
```

```
ROLE              TITLE
                                    YEAR
----------------  ------------------------------------
    ----
Lisbeth Salander  The Girl Who Kicked the Hornet's Nest
    2009
Lisbeth Salander  The Girl Who Played with Fire
             2009
Lisbeth Salander  The Girl with the Dragon Tattoo
          2009
```