Databases

Ivin Lee

October 20, 2020

1 Introduction

1.1 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.

1.2 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

1.3 SQL

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

1.4 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

1.5 Distributing Data

- Replication
- Partitioning

1.6 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.

1.7 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.

1.8 BASE

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

This is an area of ongoing research.

1.9 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 Entity-Relationship Diagrams

- Entities (squares): nouns of the model
  - Weak entities:
    - existence depends on existence of another entity
  - Sub-entities: inherit all attributes of the parent entity

- Attributes(ovals): represent properties
- Key (underlined): attribute whose value uniquely identifies an entity instance
- Relationship (diamond): verbs, relation between entities
  - Cardinality:
    - one-to-many: arrow to one
    - many-to-one
    - many-to-many
    - one-to-one
  - May also have attributes
  - Can be more than 2 parties, i.e. ternary relationship

3 Relational Algebra

This will probably clash with the lectures on discrete mathematics but I'll still type them here

3.1 Mathematical Relations

Suppose S and T are sets. The Cartesian product is the set

$$S \times T = \{(s,t)|s \in S, t \in T\}$$

A binary relation over $S \times T$ is any set R with

$$\begin{aligned} & R \subseteq S \times T \\ \iff & (s,t) \in R \\ \iff & sRt \end{aligned}$$

S and T are domains. We are interested in finite relations R that can be stored.

n-ary relations

$$R \subseteq S_1 \times S_2 \times ... \times S_n = \{(s_1, s_2, ...s_n)|s_i \in S_i\}$$

Each relation is a row in a table. We can pair the column name (attribute name) with each element of S_i, so we can tell that a value 6 in an element

of a relation refers to, say, `age: 6` or `height: 6`. This way, column order doesn't matter.

A database relation R is a finite set

$$R \subseteq \{\{(A_1, s_1), (A_2, s_2), ..., (A_n, s_n)\} | s_i \in S_i\}$$

We specify R's schema as $R(A_1 : S_1, A_2 : S_2, ..., A_n : S_n)$. Each element of R is a **record**. Each S_n could be a `string`, `integer` etc.

3.2 Database Query Language

Input: a collection of relation instances
Output: a single relation instance

$$R_1, R_2, ..., R_k \Rightarrow Q(R_1, R_2, ..., R_k)$$

3.3 Relational Algebra

Q	::=	R	base relation
	—	$\sigma_p(Q)$	selection
	—	$\pi_X(Q)$	projection
	—	$Q \times Q$	product
	—	$Q - Q$	difference
	—	$Q \cup Q$	union
	—	$Q \cap Q$	intersection
	—	$\rho_M(Q)$	renaming

$X = \{A_1, A_2, ..., A_k\}$ is a set of attributes.
$M = \{A_1 \mapsto B_1, A_2 \mapsto B_2, ..., A_k \mapsto B_k\}$ is a renaming map

A query must be well-formed: all column names must be distinct

3.3.1 Selection

Selecting one row:

$\sigma_{A>12}(R)$
```
select distinct * from R where R.A > 12
```

Reason for `distinct` is because SQL works with multisets so to follow set

properties we need to use it. The choice of multisets is because there may be times when we need to sum these values, so we need to preserve duplicates.

3.3.2 Projection

Selecting multiple columns
$\pi_{B,C}(R)$
`select distinct B, C from R`

3.3.3 Renaming

$\rho_{\{B \mapsto E, D \mapsto F\}}(R)$
`select A, B as E, c, D as F from R`

3.3.4 Union

$R \cup S$
`(select * from R) union (select * from S)`

3.3.5 Intersection

$R \cap S$
`(select * from R) intersect (select * from S)`

3.3.6 Difference

$R - S$
`(select * from R) except (select * from S)`

3.3.7 Product

$R \times S$
`select A, B, C, D from R cross join S`
`select A, B, C, D from R, S`
Note that after crossing, the pair of pairs is flattened

3.3.8 Natural Join

Notation
Relational Schema: $R(A)$, where $A = \{A_1, A_2, ..., A_n\}$ are the attribute names.
$R(A, B)$ means $R(A \cup B)$ and $A \cap B = \phi$

$u.[A] = v.[A]$ abbreviates $(u.A_1 = v.A_1) \wedge ... \wedge (u.A_n = v.A_n)$

Given $R(A, B)$ and $S(B, C)$, we define natural join as a relation over attributes A, B, C as

$$R \bowtie S \equiv \{t | \exists u \in R, v \in S, u.[B] = v.[B] \wedge t = u.[A] \cup u.[B] \cup v.[C]\}$$
$$R \bowtie S = \pi_{A,B,C}(\sigma_{B=B'}(R \times \rho_{B \mapsto B'}(S)))$$

So natural join is a composite function.

4 Implementing ER with SQL

The entity-relationship model does not dictate implementation. However some methods are better than others...

Lumping all data in one big table leads to:

- Insertion anomalies:
  - if some of the columns of the data to be inserted are supposed to have the same values as some of the columns in the table already, how do we check if they are the same?
  - Insert data where some columns are empty
- Deletion anomalies:
  - deleting a row will delete more information than we should
  - the other method is to make those columns null
- Update anomalies:
  - Many rows to update if the rows share some column values
- a transaction might take very long are it needs to scan through multiple rows when information is duplicated in the table (i.e. checks for correctness)
- this database cannot support many concurrent updates

A better idea is to break tables down, having tables for both entities and relationships. Then to retrieve the table, we can join the smaller tables up.

- Upside: reduce update anomalies
- Downside: a lot of effort to combine information when making queries

4.1 Keys

To make sure that the relational implementation is correct, we can use keys and foreign keys.

Suppose $R(X)$ is a relational schema with $Z \subseteq X$. If for any records u and v in any instance of R we have

$$u.[Z] = v.[Z] \Rightarrow u.[X] = v.[X] \tag{1}$$

(i.e. Z uniquely identifies a single record), then Z is a superkey for R. If no proper subset of Z is a superkey, (i.e. Z is the smallest combination of attributes needed to uniquely identify a record), then Z is a key for R. We write $R(\underline{Z}, Y)$ to indicate that Z is a key for $R(Z \cup Y)$.

Suppose we have $R(\underline{Z}, Y)$. Furthermore, let $S(W)$ be a relational schema with $Z \subseteq W$. We say that Z represents a foreign key in S for R if for any instance we have $\pi_Z(S) \subseteq \pi_Z(R)$ (i.e. all records of attribute Z in S point to a record with the same attribute Z in R).

A database is said to have referential integrity when all foreign key constraints are satisfied.

```
1  create table genres (
2      genre_id integer NOT NULL,
3      genre varchar(100) NOT NULL,
4      PRIMARY KEY (genre_id));
5
6  create table has_genre (
7      movie_id varchar(16) NOT NULL
8          REFERENCES movies (movie_id),
9      genre_id integer NOT NULL
10         REFERENCES genres (genre_id),
11     PRIMARY KEY (movie_id, genre_id));
```

SQL allows you to specify these constraints, and will prevent any violations of referential integrity (i.e. preventing you from deleting a row with a primary key when another table references that primary key as a foreign key)

4.2 Implementing Relationships

- Many to many: $R(\underline{X, Z}, U)$

- one to many: $R(\underline{X}, Z, U)$
- many to one: $R(X, \underline{Z}, U)$

In a many to many relationship, it is intuitive that you can identify each record only you have the details of X and Z.

In a one to many relationship (i.e. one department to many employees), then for each employee the department may be repeated so only the employee's identification should be a primary key.

Following the previous example, we might want to include the relationship between employees and departments into the table for employees. This is especially so if the entity is a weak entity.

- Pros: Less complex queries
- Cons: If the schema needs to change (i.e. one to many to many to many) migration is needed, instead of just needing to add more records to the relationship table.

Multiple relationships in a single table

If S and T are related by 2 relationships, a workaround could be to have one relationship table, with an indicator variable to specify whether we are referring to the first or second relationship. Then we have 2 more columns, one of them null and the other is the value of the correct relationship.

$$\text{Instead of:} \tag{2}$$
$$R(\underline{X, Z}, U) \tag{3}$$
$$Q(\underline{X, Z}, V) \tag{4}$$
$$\text{We could have:} \tag{5}$$
$$RQ(\underline{X, Z, type}, U, V) \tag{6}$$

5 SQL Commands

Aggregate Commands

```
1 select position, count(*) as total
2 from has_position
3 group by position
4 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
---------------   ---------------------------------------
    ----
```

```
3 Lisbeth Salander  The Girl Who Kicked the Hornet's Nest
      2009
4 Lisbeth Salander  The Girl Who Kicked the Hornet's Nest
      2009
5 Lisbeth Salander  The Girl Who Played with Fire
                2009
6 Lisbeth Salander  The Girl Who Played with Fire
                2009
7 Lisbeth Salander  The Girl with the Dragon Tattoo
              2009
8 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?)

```
1 select distinct r1.role as role, m1.title as title, m1.
     year as year
2 from plays_role as r1
3 join plays_role as r2 on r2.person_id = r1.person_id
4 join movies as m1 on m1.movie_id = r1.movie_id
5 join movies as m2 on m2.movie_id = r2.movie_id
6 join people as p on p.person_id = r1.person_id
7 where p.name = 'Noomi Rapace'
8       and r1.role = r2.role
9       and m1.movie_id <> m2.movie_id
10 order by m1.title, r1.role, m1.year;
```

```
1 ROLE              TITLE
                                        YEAR
2 ----------------  ------------------------------------
      ----
3 Lisbeth Salander  The Girl Who Kicked the Hornet's Nest
      2009
4 Lisbeth Salander  The Girl Who Played with Fire
                2009
5 Lisbeth Salander  The Girl with the Dragon Tattoo
              2009
```