Databases Notes

Hashan Punchihewa

https://hashanp.xyz

Contents

1 Databases **1**
- 1.1 Relational Algebra . . . 1
- 1.2 Functional Dependency . . . 2
- 1.3 Normalisation . . . 4
- 1.4 SQL . . . 5
- 1.5 Entity Relationship Modelling . . . 10
- 1.6 Transactions . . . 14
- 1.7 Storage . . . 15
- 1.8 Indexing . . . 18

Chapter 1

Databases

1.1 Relational Algebra

title:string	year:int	length:int	genre:string
Gone with the Wind	1939	231	Drama
Star Wars	1977	124	SF
Wayne's World	1992	95	Comedy

Various operations can be performed on sets:

- Union. Given two sets R and S, $R \cup S$ is the union of these sets, but the attributes of these two relations must be the same.
- Set difference. Given two sets R and S, $R - S$ is the set difference of these sets, but the attributes of these two relations must be the same.
- Intersection. Given two sets R and S, $R \cap S$ is the union of these sets, but the attributes of these two relations must be the same.
- Projection. Given a set R, $\pi_{a,\ldots,k}(R)$ is equal to modified version of R with only attributes $a \ldots k$.
- Selection. Given a set R, $\sigma_p(R)$, is equal to R with only the tuples that satisfy the predicate p.
- Cartesian Product. Given two sets R and S, $R \times S$ is equal to the Cartesian product of the two relations, but if there are common attributes to both relations, then these attributes are prefixed with the name of the relation.
- Natural Join. Given two sets R and S, $R \bowtie S$, is the Cartesian product, where rows are only taken if the values of the common attributes of any two tuples are the same. Note, that no attributes are prefixed with the name of their original table, as duplicates attributes are made into a single attribute.

- Inner Join. Given two sets R and S, $R \bowtie_{R.a=S.b}$ takes the Cartesian product and then filters where $R.a = S.b$. Note that, all attributes that appear in both tables are prefixed with the name of the their original table.
- Left Outer Join. Given two sets R and S, $R \mathbin{⟕}_{R.a=S.b}$ takes the Cartesian product and then filters where $R.a = S.b$, but if for some row in R, there are no rows in S for which $R.a = S.b$, then this row appears in the output with the columns of S being NULL. Note that, all attributes that appear in both tables are prefixed with the name of the their original table.
- Right Outer Join. Given two sets R and S, $R \bowtie_{R.a=S.b}$ takes the Cartesian product and then filters where $R.a = S.b$, but if for some row in S, there are no rows in R for which $R.a = S.b$, then this row appears in the output with the columns of R being NULL. Note that, all attributes that appear in both tables are prefixed with the name of the their original table.
- Full Outer Join. Given two sets R and S, $R \mathbin{⟗}_{R.a=S.b}$ takes the Cartesian product and then filters where $R.a = S.b$. If for some row in R, there are no rows in S for which $R.a = S.b$, then this row appears in the output with the columns of S being NULL. If for some row in S, there are no rows in R for which $R.a = S.b$, then this row appears in the output with the columns of R being NULL. Note that, all attributes that appear in both tables are prefixed with the name of the their original table.
- Renaming. Given a set R, $\rho_{new_1/old_1,\ldots,new_n/old_n}$, this operations replaces the old attribute names with the new attribute names.

1.2 Functional Dependency

A functional dependency exists between $A_1, \ldots, A_n$ over $B_1, \ldots B_m$, denoted as $A_1, \ldots A_n \rightarrow B_1, \ldots B_m$, if whenever tuples agree on the value of $A_1, \ldots A_n$, they must also agree on the value of $B_1, \ldots, B_m$.

If a set of attributes functionally determines all other attributes, this set of attributes is a superkey. Superkeys, which are as small as possible, i.e. irreducible, are candidate keys. There may exist more than one candidate key.

If L is a set of attributes, and F a set of functional dependencies, the closure of L under F, L^+ is all the attributes functionally determined by L under F. To check if L is a super key of a relation R, check all the attributes of R are a subset of L^+. To check whether $L \rightarrow R_1, \ldots, R_n$ holds, check whether $R_1, \ldots, R_n$ is a subset of L^+.

There exist several rules for manipulating functional dependences:

- Splitting Rule. E.g. if $ABC \rightarrow WXY$, one can split this to $ABC \rightarrow W$, $ABC \rightarrow X$ and $ABC \rightarrow Y$.

- Combining Rule. E.g. if $ABC \rightarrow W$, $ABC \rightarrow X$, $ABC \rightarrow Y$, then one can say $ABC \rightarrow WXY$.
- Trivial Dependency Rule. If some attributes on the left-hand side are on the right-hand side, they can be removed from the right-hand side, $ABC\ toABWXY$, can be transformed to $ABC \rightarrow WXY$.

Armstrong's axioms:

- Reflexivity. $\alpha \rightarrow \beta$, always holds if $\beta \subseteq \alpha$.
- Augmentation. If $\alpha \rightarrow \beta$, then $\alpha\gamma \rightarrow \beta\gamma$.
- Transitivity. If $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.

These additional axioms can be derived from Armstrong's axioms:

- Union. If $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta\ \gamma$.
- Decomposition. If $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.
- Pseudo-transitivity. If $\alpha \rightarrow \beta$, and $\delta\ \beta \rightarrow \gamma$, then $\delta\ \alpha \rightarrow \gamma$

The closure of a functional dependency set is the set of all functional dependencies that can be derived. One approach to obtaining the closure of a functional dependency set:

1. Initialise $F^+ = F$.
2. Apply reflexivity and augmentation axioms.
3. Apply transitivity axioms to appropriate functional dependencies.
4. Repeat from step 2, until there are no further changes.

Two sets of functional dependencies F_1 and F_2 are equivalent if both imply the other. F_1 and F_2 are covers of each other.

A set of functional dependencies is canonical if:

- Each LHS is unique.
- We cannot delete any functional dependency and get an equivalent functional dependency set.
- We can delete any attribute and have an equivalent functional dependency set.

Essentially no functional dependency or functional dependency attribute is unnecessary. To test whether a left-hand side attribute is extraneous, check $RHS \subseteq (LHS \setminus \{X\})^+$, excluding that functional dependency. To test whether a right-hand side attribute is extraneous, check if $X \in LHS^+$, with X removed from the right-hand side of the functional dependency.

To find a canonical cover for a functional dependency set:

- Apply union rule, when possible.

- Remove extraneous attributes.
- Repeat step 1, till nothing changes.

1.3 Normalisation

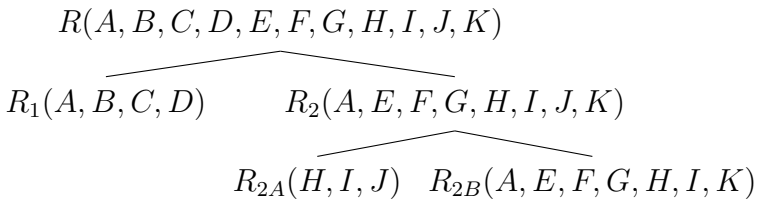
Definition 1.3.1 *A relation is in BCNF (Boyce-Codd Normal Form), if for all non-trivial functional dependencies, including derived dependencies, every left-hand side is a superkey.*

Decomposition into BCNF is recursive. For a violating function dependency $X \to Y$ on a relation A, where $X \cup Y \subseteq A$ and $X \cap Y = \varnothing$, then this is broken down into two relations, $X \cup Y$, and $A \setminus Y$. The decomposition procedure is then recursively applied to these two new relations, until there are no more violating dependencies.

For instance, in the case of $R(A, B, C, D, E, F, G, H, I, J, K)$:

$$\begin{aligned} A &\to BCD \\ HI &\to J \\ AEFG &\to HIK \end{aligned}$$

This can be decomposed as follows:



Definition 1.3.2 *An attribute is prime, if it is a member of any key.*

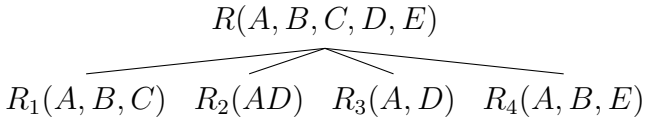
Definition 1.3.3 *A relation is in 3NF (3rd Normal Form), A relation is in BCNF (Boyce-Codd Normal Form), if for all non-trivial functional dependencies, including derived dependencies, every left-hand side is a superkey, or if every attribute on the right-hand side is prime.*

To decompose a relation in 3NF, for a relation A with functional dependencies F, let C be the canonical cover over F. Then for each function dependency $X \to Y$, let $X \cup Y$ be a new relation. Remove any relations created that are subsets of other relations. Then if none of the relations contain a key, add any key as a relation.

For instance, in the case of $R(A, B, C, D, E)$:

$$AB \to C$$
$$C \to B$$
$$A \to D$$

This leads to the keys $\{A, B, E\}$ and $\{A, C, E\}$. This decomposes into:



Note that R_4 could equally be $R_4(A, C, E)$.

1.4 SQL

Some of the types available in SQL:

- `int` (4 bytes), `smallint` (2 bytes), `float(n)`, where n is the number of bits of the mantissa. Note `real` is `float(24)` and `double precision` is `float(53)`. `decimal(p, s)`, where p is the number of decimal digits, and s the number of decimal digits to the right. Arithmetic operators include `+`, `-`, `*`, `/` and `%`.
- `char(n)`, `varchar(n)`, `text`. Strings can be pattern matched with `like`, e.g. `string like 'b%'`, where `%` refers to 0 or more arbitrary characters.
- `bit(n)`, `byte(n)`, `blob`
- `boolean`. Literals are `true` and `false`. SQL comparisons operators are `=`, `<>`, `<=`, `>=`, `<`, `>`. Boolean operators include `and`, `or` and `not`. Also `between x and y`. You can also use `in` and `not in`, i.e. `country in ("UK", "France", "Germany")`. There is also `is null` and `is not null`.
- `date`, `time` and `timestamp`. Dates are specified as `'1998-09-23'`, times as `'12:03:04'` and timestamps as `'1998-09-23 12:03:04'`.

SQL uses 3-valued logic. This is almost identical to standard Boolean logic, except with the additional of a third value: unknown. If unknown appears in a Boolean expression, unless the value of the expression does not depend on the value that is unknown, then the expression is evaluated to be unknown. For instance $\top \vee$ UNKNOWN, will always be $\top$, but in the case of $\bot \vee$ UNKNOWN, then it does depend on UNKNOWN, so the expression evaluates to UNKNOWN. This is as SQL values can be `null`. Comparisons involving `null`, will result in UNKNOWN, whereas arithmetic operations will simply result in `null`.

Example of a select statement:

```
select * from movie where name='Saving Private Ryan'
select * from movie where revenue > 5000000 and title like 'Marvel%'
select title from movie where name='Big Sick'
select title, duration from movie where name='Love Rosie'
select title as name, duration from movie where name='Breakfast Club'
select duration / 60 as hours from movie where name='The Avengers'
select * from movie order by year
select * from movie order by year desc, title asc
```

Example of a Cartesian product and joins:

```
select * from movie, producer
select title, producer.producer_id from movie, producer
select * from movie natural join producer
select title, name from movie natural join producer
select * from movie join producer on movie.producer_id = producer.producer_id
select * from movie join producer using (producer_id, producer_id)
select c.title, d.title movie as c join movie as d on c.sequel = d.title
select * from movie join producer using (producer_id, producer_id)
    join actor on lead_actor = actor.name
select * from movie left outer join producer
    on movie.producer_id = producer.producer_id
```

Other than `left outer join`, you can do `right outer join` and `full outer join`. Note `outer` is optional, and you can write `join` as `inner join`.

5 Aggregation functions:

- `count(*)`
- `sum(prop)`
- `avg(prop)`
- `min(prop)`
- `max(prop)`

They are especially useful whenx combined with `group by`:

```
select
    count(*) as total_films,
    sum(revenue) as total_revenue,
    avg(revenue) as average_revenue,
    min(revenue) as min_revenue,
    max(revenue) as max_revenue
from film
group by genre, producer_id
order by total_revenue desc
```

You can also filter but using `having`:

```
select
    count(*) as total_films,
    sum(revenue) as total_revenue,
    avg(revenue) as average_revenue,
    min(revenue) as min_revenue,
    max(revenue) as max_revenue
from film
group by genre, producer_Id
order by total_revenue desc
having average_revenue > 1000000000
```

```
(select * from movie where revenue > 500000)
    intersect (select * movie where title like 'b%')
```

You can also use `union` and `except`. Note that these all remove duplicates. There exist versions `union all`, `intersect all` and `except all`, which don't.

You can also have subqueries in SQL. A scalar subquery produces a single value. A set subquery produces a set of distinct values. A relation subquery, as seen above produces a relation. Scalar subqueries can be used as follows:

```
select title, year,
    (select count(*) from casting where casting.title=move.title) as num_actors
    from movie
```

This can be made more readable by using joins. In addition, scalar subqueries can be put in `WHERE` clauses. Set subqueries can be used with `in` and `not in`. They can also be used with `some` and `any`:

```
select title, year from movie
    where year < all (select date_of_birth from employee where department='IT')
```

Relational subqueries can also be used with `exists`, `not exists`, `unique` and `not unique`:

```
select title, year from movie
    where exists (select * from employee where date_of_birth=year)
```

Also you can eliminate duplicates by doing `select distinct`.

Here is how to create a table:

```
create table movie (
    title varchar(120),
    year int default 2011,
    duration int not null,
    genre char(20),
    primary key (title, year)
)
```

Unique constraints can also be added as well as general constraints based on expressions:

```
create table movie (
    title varchar(120),
    year int,
    duration int,
    genre char(20),
    ISAN char(24),
    primary key (title, year),
    unique(ISAN),
    check(year > 1900)
);

create table person (
    id int,
    first_name varchar(120),
    last_name varchar(120),
    primary key(id),
    unique (first_name, last_name)
)
```

Groups of values can have a unique index:

```
unique(first_name, second_Name
```

Constraints can be named as follows:

```
constraint primaryKeyCheck primary key(title, year)
constraint uniqueCheck unique(name)
constraint yearCheck check(year > 1900)
```

In addition foreign keys, e.g. in a `casting` table:

```
foreign key person_id references person.id
foreign key (title, year) references movie (title, year)
```

To remove an existing field from a table:

```
alter table movie drop duration;
```

To add a new field to a table:

```
alter table movie add studio char(16);
```

There are 3 possible policies to deal with updates that get rid of referential integrity in relationships. The default policy is to reject any updates that will get rid of referential integrity. The second is 'cascade'. This can be applied to updates and deletions:

```
create table movie (
    title varchar(120),
    year int,
    duration int,
    genre char(20),
```

```
    primary key (title),
    on update cascade on delete cascade
)
```

It is also possible to make assertions over several relations, although this can be quite inefficient:

```
create assertion assert_name check (expr)
```

In this case, if the primary key is changed, this change will be reflected in any other tables that reference this table. Similarly, if the row is deleted, any rows that reference this row will be deleted. The other policy is 'set null', where the reference is set to `null`. Note a different policy can be kept for 'update' and 'delete'.

```
insert into movie(title, released, genre) values ('Lady Bird', 2017, 'Drama')
```

Views can also be declared:

```
create view comedies as select title, year from movie where genre='comedy'
```

Materialised views all where views are stored. This imposes a storage cost, for speeds up access to views.

To update record:

```
update movie
    set name='Star Wars IV: New Hope',
        revenue=2*revenue
    where title='Star Wars'
```

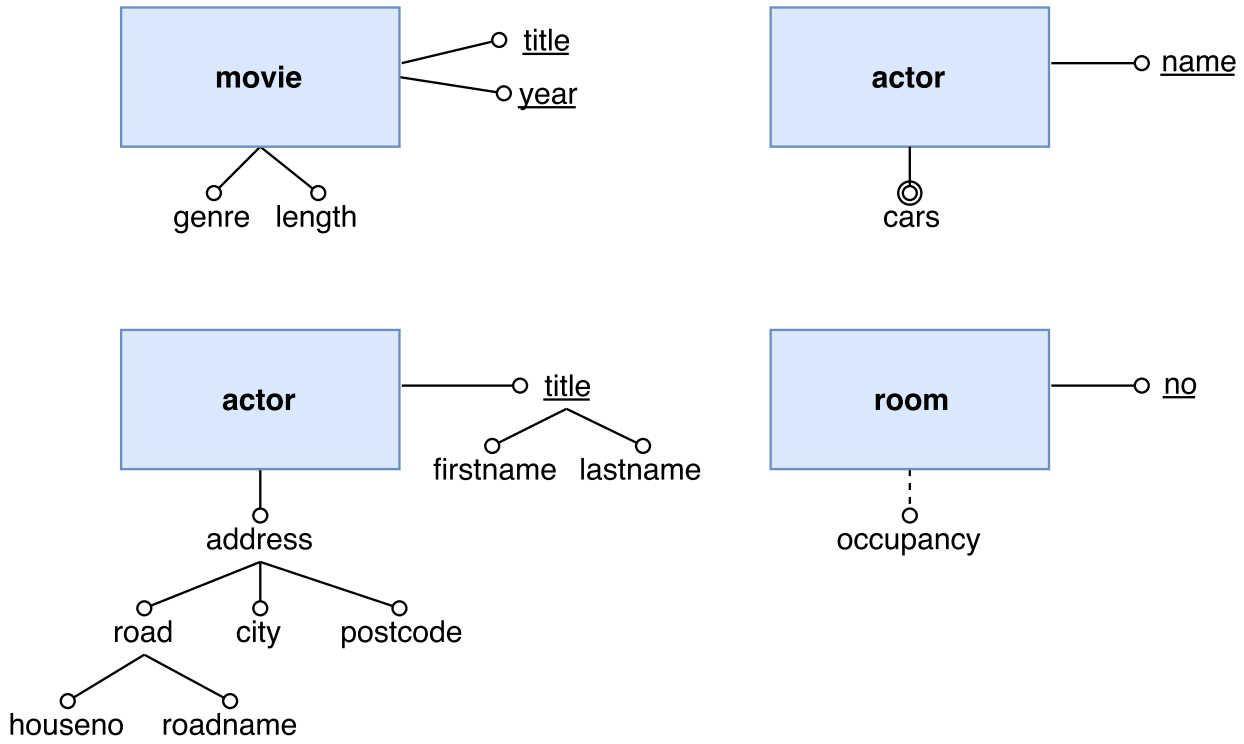
You can also refer to previous values:

```
update employee
    set salary = case
            when salary <= 70000 then salary * 1.02
            when salary <= 80000 then salary * 1.03
            else salary * 1.04
        end,
        last_pay_increase = '2017-04-12'
    where position = 'Professor'
```

Also deletions are performed as so:

```
delete from movie where year < 1990
```

1.5 Entity Relationship Modelling

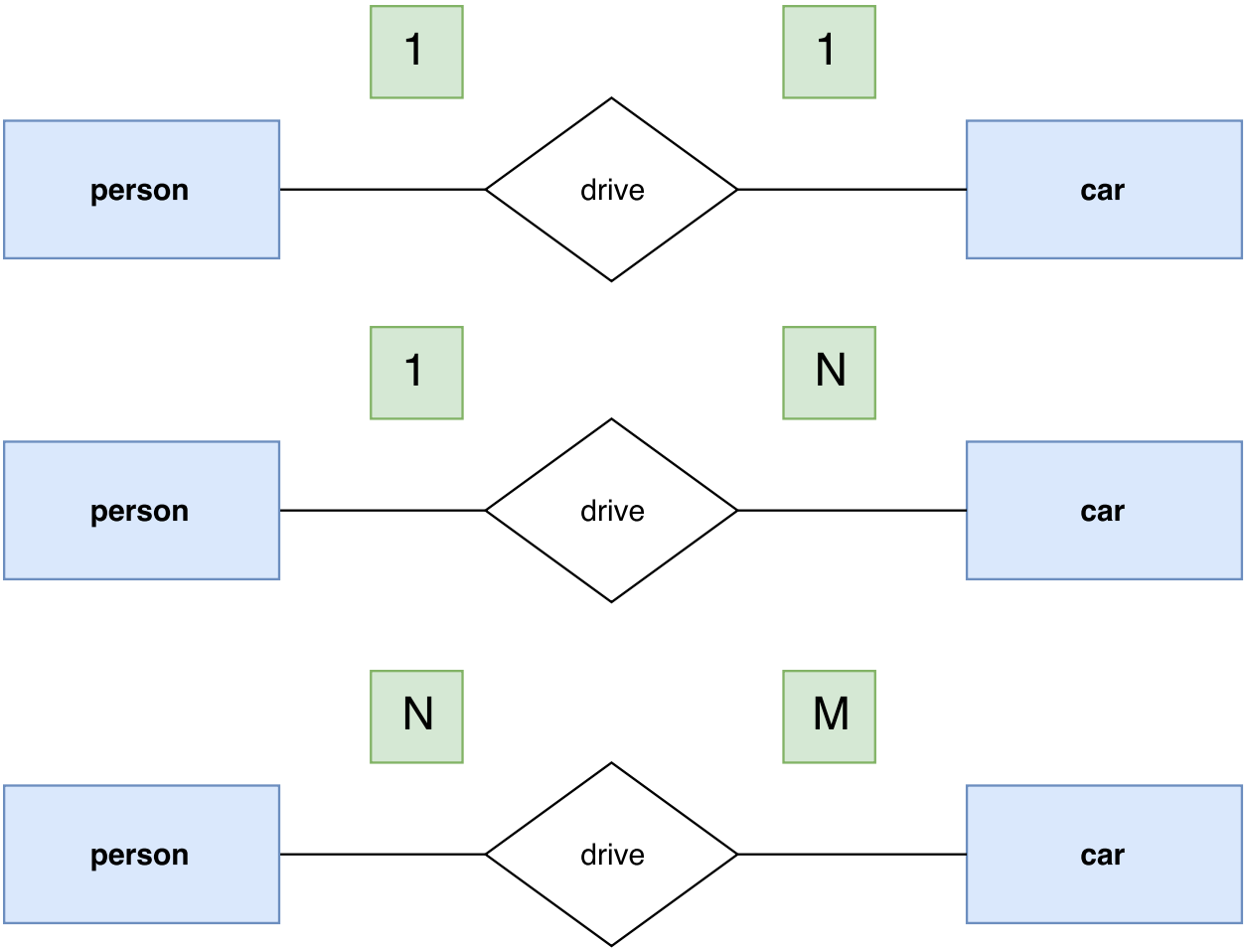


Rules of ER diagrams:

- Rectangles indicate entities.
- Lines with circles indicate attributes.
- Underlined attributes indicate that they are primary keys.
- Composite attributes can be shown with attributes belonging to other attributes.
- If a primary key is a composite attribute, only the primary key and not its constituent attributes.
- Lines with two circles, one inside of another, indicate that the attribute is multivalued.
- Dashed lines indicate derived attributes.

Also note when translating into SQL:

- Only the atomic attributes are kept, composed attributes are not.
- Derived attributes may be included, but when updates are performed, they will need to be updated so that they are consistent. Or they could be computed when needed.
- Multivalued attributes are stored as an additional table for the attribute with a foreign key, referring to the main entity, and a one-to-many relation.



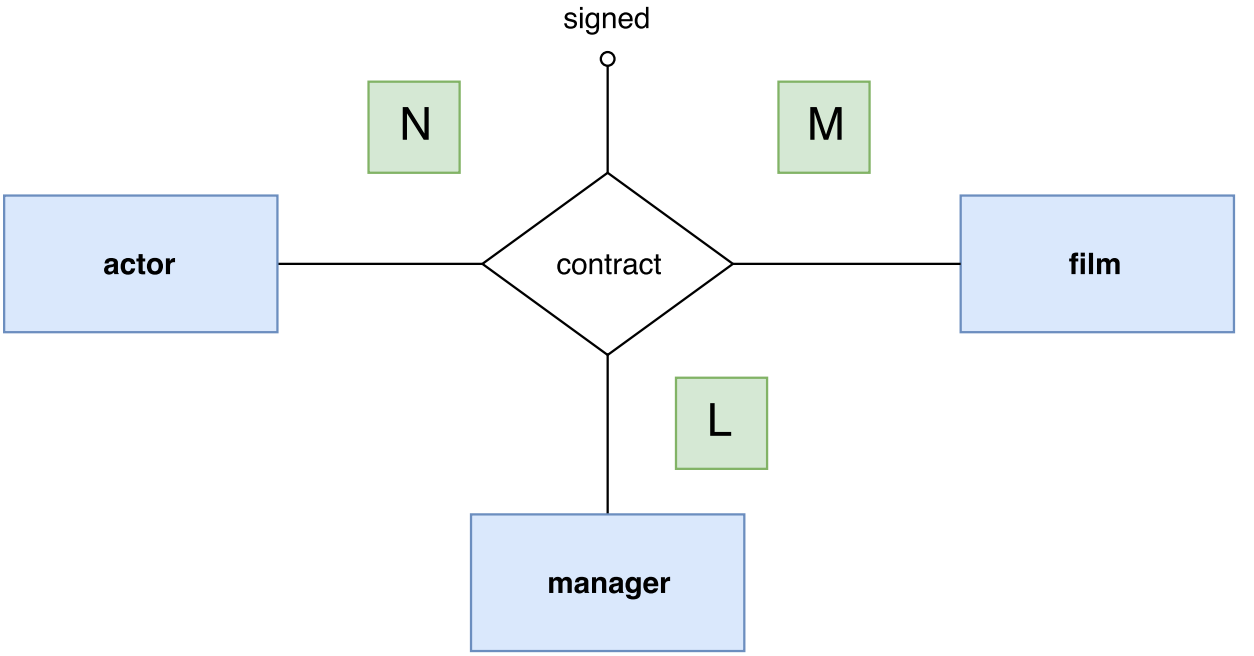
Rules of relationships in ER diagrams:

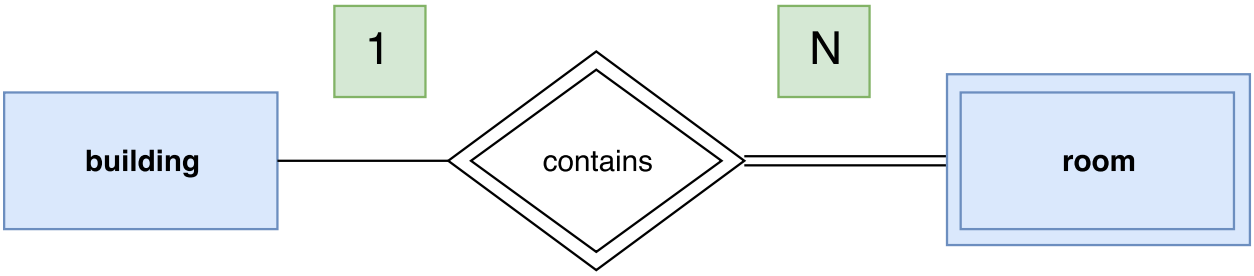
- Double lines means total participation i.e. each entity must participate in the relationship at least once.
- There are two other ways of writing cardinality constraints. Either writing a number e.g. 1 or a letter e.g. N to indicate the maximum number of times the entity participates, or writing a range $1..N$ to indicate the possible range of values. Note a letter means an the entity can participate an arbitrary number of times.
- Fan traps occur if there is an unclear connection between two entities. E.g. if a lecturer belonged to a faculty, and a faculty managed departments, it wouldn't be clear which lecturer belonged to which department.
- Chasm traps occur if a relation between entities is suggested, but it doesn't exist. E.g. if PCs were operated by staff members who belonged to departments. It might seem as if there was a relation between departments and PCs, but what if a PC hadn't been assigned a staff member.
- Weak entities where an entity is dependent on another entity for its existence, is indicated by putting a double border around the rectangle of the entity, and a double border around the diamond with the corresponding strong entity.

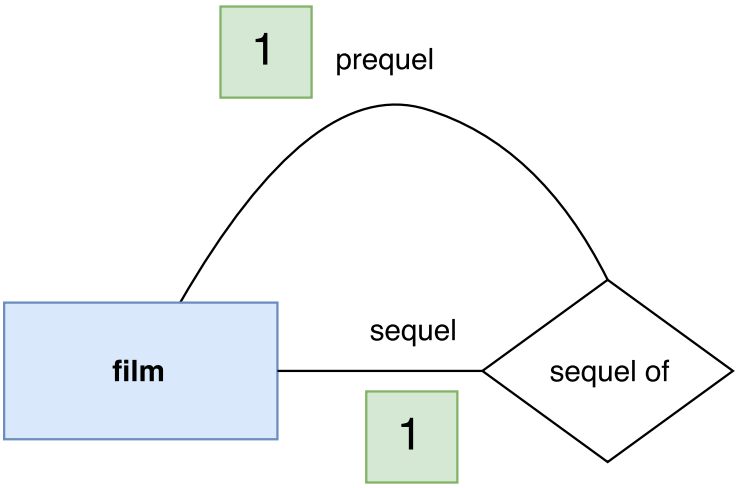
- An is-a relationship can be indicated with an arrow.
- A role is where the same entity plays multiple roles in a single relation. This can be shown by giving each instance of the entity a name.
- Also note, relations can have attributes.

Notes when translating into SQL:

- One-to-one relationships do not need an additional table. They can be implemented as a foreign key, which has a unique constraint in one of the tables.
- One-to-many relationships are implemented as an attribute in the ‘many‘ table.
- A many-to-many relationship is implemented.
- Multiway relationships are also implemented as a separate table, with foreign keys. If a particular row could participate as one of the attributes of the relationship once, then a unique constraint can be used.
- Ensuring total participation on one side in a one-to-one relation is straightforward, by using `not null` and then putting the attribute on the table that must participate. Otherwise constraints have to be used.
- To model roles, another table is usually used.
- Is-a relationships where you have one-entity which is essentially a specific case of another entity can be modelled in different ways:
  - Having separate tables, for the more specific cases that uses foreign keys to reference the more general case. This requires joins though, to retrieve all the information.
  - Having a table for each possible type of entity, where the more specific cases have all the attributes of the more general cases. Note, that tables are needed for combinations of specific cases.
  - Having a single table, with all the attributes of the more specific cases, which are set to `null` if not needed.


signed
N
M
actor
contract
film
L
manager


1
N
building
contains
room


1
prequel
film
sequel
sequel of
1

1.6 Transactions

Relational databases execute instructions in units called transactions. Transactions follow the principles of ACID:

- Atomicity - A transaction happens in its entirety or not at all.
- Consistency - A transaction will only occur if it doesn't bring the database into an inconsistent state.
- Isolation - Concurrent execution of transactions will produce the same result as if the transactions were executed sequentially.
- Durability - Once a transaction it is committed it will remain committed.

This is achieved by having a log, that can be rolled back. In addition, RAID and RAID5 in particular can be used, which increases performance by stripping and also redundancy by mirroring. Also replication can be used. A replica receives updates from the main database, and is ready to take over in case that main database fails.

In SQL a transaction begins with:

```
start transaction
```

Or alternatively:

```
begin
```

To commit a transaction:

```
commit
```

To rollback a transaction:

```
rollback
```

SQL also supports different isolation levels, when performing transactions. If adhering to the principles of ACID, then if two identical database reads are performed in the same transaction, they should provide the same data. However, in reality this reduces concurrency, so it may be better to relax this requirement. This can be done by choosing an isolation level:

- Read uncommitted. Here uncommitted data by other concurrent transactions can be read i.e. dirty reads.
- Read committed. Only data committed by other concurrent processes can be read.
- Repeatable read. Similar to read committed, but if a read is performed, then another identical read will definitely return the same rows. It may return additional rows, i.e. phantom rows, but there will be no rows missed out, that were previously returned. These rows will not be modified in any way. More technically, read rows are "locked" so they cannot be deleted or updated.

- Serialisable. This guarantees isolation as per ACID. When read or write queries are performed, then "range locks" are applied to the range of values that fall under the search conditions of the query, e.g. specified under `WHERE` clauses in SQL. This means that updates, insertions and deletions cannot be performed on rows, that fall under these search conditions, for the duration of the transaction.

Dirty writes are not allowed by any transaction level. This is where data is updated, that has already been updated by not committed or aborted by another transaction. In other words, whenever a write query is performed, the rows affected are locked.

1.7 Storage

In database systems, clients send requests to a query server. Transactions are executed on the server, and sent back to the client. The requests are in SQL and communicated through RPC, remote procedure call. ODBC, Open Database Connectivity is a C API for interacting with databases. JDBC, Java Database Connectivity is a Java equivalent. A query server consists of multiple processes:

- Server processes to handle queries. These processes usually are multithreaded to handle multiple queries concurrently.
- Lock manager process.
- Databases writer process. The database writer process writes changes in memory to disks.
- Log writer process. A log process writes the log buffer, to disk.
- A checkpoint process. A checkpoint is where the current in-memory changes, that haven't been written to disk are written to a disk, so if the database crashes this information can be used. This helps reduce the recovery time, in the event of a failure.
- A process monitor. A process monitor process monitors other processes, and takes actions in case one of them has failed.

Inside a database's shared memory, the following are contained:

- Buffer pool
- Lock table
- Log buffer
- Cached query plans

Either using locks or atomic instructions (test and set) mutual exclusion is achieved to prevent access to the same data structure. The characteristics of storage that matter to databases are:

- Speed
- Cost (per unit)
- Reliability
- Volatility

The types of physical storage in order of fastest to slowest and most expensive to cheapest:

- Cache. This is the fastest memory. It is volatile and has very small capacity.
- Main Memory. This is still very fast. It is volatile, but usually too small to hold the entire database.
- Flash Memory. This usually has fast reads, but writes can be slow. This is as entire memory block must be erased before being rewritten, and erasure is slow. There are also a limited number of write/erase cycles. It is non-volatile.
- Magnetic disk. This typically stores entire database. There is random-access. Disk failure is rare but not impossible.
- Optical storage. Reads and writes are slow and it is non-volatile.
- Tape storage. Serial access means it is very slow. It does have very high capacity, though. Tapes can be removed from drive. This is important as tapes are cheap, but the drives are expensive. It is usually used for backup or archival.

Usually storage is split into;

- Primary storage, which is very fast and volatile storage: Main memory, cache.
- Secondary storage, non-volatile and moderately fast: Flash memory and magnetic disk. Sometimes called on-line storage.
- Tertiary storage, non-volatile and slow access times: Magnetic tape and optical storage. Sometimes called off-line storage.

A magnetic disk consists of one or more spinning platter and read-write heads which are positioned close to the surface of the platter. Usually there are 2 read-write heads for each platter, one on each side. The surface of a platter is divided into tracks. These tracks are essentially concentric circles. Each track is divided into sectors, which are typically 512 bytes. When reading and writing a sector, the disk arm moves to the correct track, and reads the sector as it passes under it.

A disk controller interfaces between hardware and software. It accepts commands to read and write, handles moving the arm to the correct position, computes and attaches checksums, to verify data has been read back correctly, ensures writing is successful by reading afterwards, and perform remapping of bad sectors.

There are two components of access time: seek time, time to position under correct track; rotational latency, time for sector to appear under head. There is also the disk transfer rate, the rate at which data can be read from or written to a disk.

Data is transferred from main memory to the hard drive in blocks. A block is a contiguous sequence of sectors in the same track, usually 4 to 16 kilobytes. A disk-arm-scheduling algorithm order pending accesses to tracks so disk arm movement is minimised. One such algorithm is the elevator algorithm, where the disk arm is moved in one direction until no more requests in that direction, so it reverses direction and repeats.

Block access time can be optimised by storing related information on the same blocks, so the file system attempts to access contiguous chunks of blocks. But files may get fragmented over time, files are stored all over a disk. Defragmentation utilities can help with this. Non-volatile RAM buffers speed up writes by writing to a non-volatile RAM (battery-powered RAM or flash memory). Writes can be reordered for efficient access. Alternatively, a log disk can store a sequential log of block updates, used for like non-volatile RAM, and fast since no seek required. Journalling file systems take advantage of non-volatile RAM or log disks.

To minimise block transfer, a buffer is kept in main memory, which is controlled by a buffer manager. If a block that is needed isn't in main memory, then another block has to thrown away, to fit the new block. It is only written to disk, if it has been modified. Buffer replacement strategies include:

- Least recently used.
- Most recently used.
- Pinned block - Memory block not to be written back to disk.
- Toss immediate - Throw away blocks as soon as not needed.

Databases stored files consisting of records. These records are either fixed-length records or variable-length records. Fixed-length records are easier, as they make accessing the ith record simpler. Deletion works by maintaining a free list. Variable-length pages are handled by splitting the file into pages, with slotted page headers containing the number of entires, the end of free space, and the location and the size of each record in the page. Pointers don't point to the record, but to the entry for the record in the header.

Records can be ordered in a number of ways:

- Heap - Anywhere there is space
- Sequential based on a search key
- Hashing
- Multitable clustering file organisation: Allows records of different tables in one file.

When storing files sequentially, there is a pointer chain, in addition to one used for the free list, and an overflow buffer is used if there is no space for insertions. The pointer chain will then point to the overflow block. From time to time, the file needs to be reorganised to account for insertions and deletions.

Files can either be stored in a row-oriented or column-oriented way, where either the same attributes are stored together, or attributes from the same row are stored together.

Depends on the type of query performed. Sorting also optimises queries filtered based on some range.

1.8 Indexing

Records with fixed length can be represented similar to a `struct` in some programming languages. Variable length fields can be represented with an offset and length pair.

A database index is a data structure to improve the speed to querying for data. An index can be created over properties in SQL as follows:

```
create index index_name on table_name (attribute_1, attribute_2, ...)
```

A unique index can also be used to enforce uniqueness, but is not required if uniqueness is specified in the schema:

```
create unique index index_name on table_name (attribute_1, attribute_2, ...)
```

Similarly an index can be dropped as follows:

```
drop index index_name
```

An index file contains records called index entires, which contain a search key, and a pointer. The search keys are either stored in a sorted order, called order indices, or they are distributed uniformly across buckets, using a hash function. The search key is either an attribute or a set of attributes.

Definition 1.8.1

- *A primary index, in a sequentially ordered file, is the index where the search key specifies the sequential order of the file. This can be called a clustering index.*
- *A secondary index, in a sequentially ordered file, is the index where the search key is different from the sequential order of the file. This can be called a non-clustering index.*
- *An index-sequential file is a ordered sequential file with a primary index.*
- *A dense index is where there is an index entry for each search key.*
- *A sparse index is where there are index entires for only some of the search keys. The actual rows in the database then point to the next entry. Sparse indices take less space and have less maintenance overhead in the event of insertions and deletions. But, they are slower than dense indices.*
- *A multilevel index is where there is a sparse index to the primary index, e.g. if the primary index doesn't fit in memory, a solution is a multi-level index.*

An ideal hash function is uniform, so the same number of search key values are assigned to each bucket from all possible values and random so the same number of records are assigned to each bucket irrespective of the actual distribution of the search key values.

Typically they perform some computation on the internal binary representation of the search key.

Definition 1.8.2 *A B+-tree is a rooted satisfying the following properties. All paths from the root to the leaves are the same length. Each node that isn't a root or a leaf has between* $\lceil n/2 \rceil$ *or* n *children. A leaf has between* $\lceil (n-1)/2 \rceil$ *and* $n-1$ *values. If the root isn't a leaf, it has at least 2 children, and if it is a leaf it has between 0 and* $n-1$ *values.*

A B+-tree node has a collection of search-key values which are ordered and children. Unless the node is a leaf, in which case rather than children, it has references to records or buckets of records. Only need a bucket, if search key is not a primary key. If there are $n+1$ children or pointers , there are n search key, so that there is a search key between each pair of children/pointers . So if the children/pointers are numbers $P_0, \ldots, P_n$, then the search key terms are $K_0 \ldots K_{n-1}$. Therefore, P_0, is followed by K_0, which is followed by P_1, etc. The final items are K_{n-1} followed by P_n. P_0 points to everything with a search key less than K_0. Similarly, P_n points to everything with a search key greater than or equal to K_{n-1}. And for every other i, P_i points to everything whose search key k is $K_{i-1} \leq k < K_i$.

When inserting into a B+-tree, if there is no space in the leaf, then a split occurs. The leaf node is split, such that the parent node points to separate nodes. If there is no space in the parent, then the split propagates upwards. If it gets to the root, this means, that the height of tree will increase by 1. Half the pointers in the original node are put in the new node.

Similarly, if on deletion a node has too few children, then it is merged with another node. And this merge can be propagated upwards, decreasing the height of tree by 1, if necessary. This can only occur if there is a sibling whose children would fit with this node's children. Otherwise, pointers are redistributed. Note that, pointers need to be updated. If the root node only has one child after deletion, it is deleted, with the one child becoming the new root.