Intro to databases

Lecture 1:

CSV data model – "columns" are separated by commas, easy to put invalid data types in column or forget to add them in.

A transaction is an indivisible amount of work done by a database software.

Transactions all have the ACID property.

Atomicity – you cannot have a half transaction done. Either its fully completed or not at all.

Consistency – if the data makes sense before a transaction then it will make sense after a transition.

Isolation - ???????????????

Durability-??????????????????

Relational Algebra:

A relation takes the form R(A, B…..) where R is the name of the relation and A and B are attributes of the relation (can also be written as $R(\vec{A})$ where the A represents the set of attributes). Such as name or DOB.

The number of attributes is the rarity of the relation.

Attrs(R) will return A, B….

We can represent the whole relation (effectively the table) with a list of tuples with the same order as they appear in the relation tuple. We can also call this the extent of R.

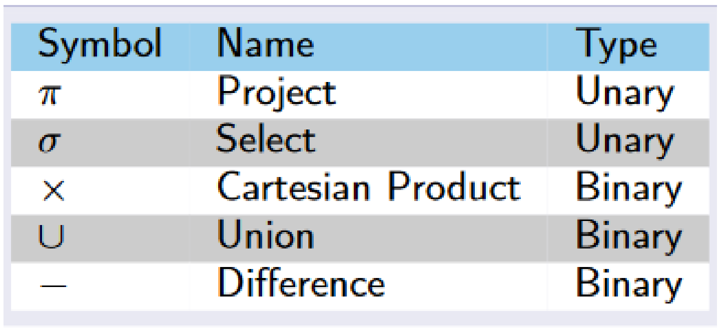
A key of a relation is a subset of Attrs(R) that are unique for all values in the extent.

Every R has at least one key which is just the whole of Attrs(R)

The minimal key is a set of attributes where no subset is also a key.

When we describe a foreign key the fancy way of writing it is $R(\vec{X}) \overset{fk}{\Rrightarrow} S(\vec{Y})$. This means that $\vec{Y}$ is a key of S. If $\vec{Y}$ appears in R that means it is a foreign key.

Primitive operators take in two relations and spit one out.



Symbol	Name	Type
π	Project	Unary
σ	Select	Unary
$\times$	Cartesian Product	Binary
$\cup$	Union	Binary
$-$	Difference	Binary

Project takes in one relation and the columns that you want to represent and shows that.

Select takes in one relation and the rows you want to show by using a bool for the column and shows whatever rows satisfy that.

Cartesian product takes in two relations and then show the pairs of all the rows.

Union and difference are pretty self-explanatory.

We need to be careful using these queries and make sure that we are using them correctly by making sure that they are well formed.

If we are using a union or a difference, then the relations must have the same number of attributes.

A natural join is effectively a more efficient cartesian product. You can just set two attributes with the same name to be equal. JOIN IS LEFT ASSOCIATIVE. Cl I don't really know what that means effectively if we have A B C it's the same as A(BC).

We can combine primitive operators to make derived operators. The first example is union which is pretty easy. R ∩ S can be defined as R – (R – S).

We can also define divide which is slightly weird. A couple of predicates before you begin. If you are doing R÷S then R must have more attributes than S. The definition is below.

$$R \div S = \pi_{Attrs(R)-Attrs(S)} R - \pi_{Attrs(R)-Attrs(S)}((\pi_{Attrs(R)-Attrs(S)} R \times S) - R)$$

So when we divide the two relations. We are going to be projecting the difference between the attributes. Then we are going to get the Cartesian product of R and S and we are going to take it away from R. This is going to return pairs that do not exist in R. Then we get R and take away the "fake" pairs and we have the division.
It kinda spits out which tuples have all the types of the column in S (or the sets of columns).

Datalog:

Datalog is another way of representing data in a database, I guess. Data is held in "extensional predicates" where the data is just held in a long tuple that takes the name of the table. For example, you could have branch(56,"Goodge Street", 123456) places of the data should correspond to the tuples or whatever I don't know.
There is also this kinda implies things that looks like this head:- body. What this does is that if the body holds then generate the head. So if we wanted all the names of people who had current accounts we could do current_account_names(names):-account(_,'current',_,_,_) and it will return all the names that have current accounts.
The minimal model is like the spanning set of the answers to a predicate thing you know what im talking about.
Safe negation is just a negation. If you want to use it, you need to make sure that any variables that you have negated appear earlier in the body.

SQL:

You need to able to translate RA into SQL code. Here's the cheat sheet.
When you use these it will generate a bag set (a set with duplicates) and you need to use the DISTINCT keyword to make it a set.
Difference will only take out the first occurrence of a certain row.

RA and SQL

RA Operator	SQL Operator
π	SELECT
σ	WHERE
$R_1 \times R_2$	FROM R_1, R_2 *or* FROM R_1 CROSS JOIN R_2
$R_1 \bowtie R_2$	FROM R_1 NATURAL JOIN R_2
$R_1 \overset{\theta}{\bowtie} R_2$	FROM R_1 JOIN R_2 ON θ
$R_1 - R_2$	R_1 EXCEPT R_2
$R_1 \cup R_2$	R_1 UNION R_2
$R_1 \cap R_2$	R_1 INTERSECT R_2

We have some different operators. For example we can define what we want in a certain column by using the IN keyword. We can either define the exact values or define a table that we want the value to exist in.

```
SELECT  *
FROM    account
WHERE   type='current'
AND     no IN (100,101)
```

```
SELECT  no
FROM    account
WHERE   type='current'
AND     no IN (SELECT  no
               FROM    movement
               WHERE   amount>500)
```

Above we can see two examples of the IN keyword.
The EXISTS keyword is used to select if an inner query has is empty before you complete a whole query.
We can also have a for all and exists function by using the ALL or SOME keywords respectively.

```
SELECT  bname
FROM    branch
WHERE   'current'=ALL (SELECT  type
                       FROM    account
                       WHERE   branch.sortcode=account.sortcode)
```

This query takes the branch names from branch and is looking for branches where all the accounts at that branch are current accounts.

SQL Nulls:

Small distraction from talking about keywords. Null in SQL kinda represents a value that might be in the table but we don't know their current value.

SQL uses tree Boolean values for where clauses, the obvious T and ⊥ and Unknown.

Formula	Result
x=null	UNKNOWN
null=null	UNKNOWN
x IS NULL	TRUE if x has a null value, FALSE otherwise
x IS NOT NULL	TRUE if x does not have a null value, FALSE otherwise

You can't really equate things with null but you can use IS NULL to check for null. Since WHERE only returns true results a WHERE that includes =null will return nothing unless it is OR'd with a true. NOT UNKNOWN = UNKNOWN.

AND

P_1 AND P_2		P_2		
		TRUE	UNKNOWN	FALSE
P_1	TRUE	TRUE	UNKNOWN	FALSE
	UNKNOWN	UNKNOWN	UNKNOWN	FALSE
	FALSE	FALSE	FALSE	FALSE

OR

P_1 OR P_2		P_2		
		TRUE	UNKNOWN	FALSE
P_1	TRUE	TRUE	TRUE	TRUE
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
	FALSE	TRUE	UNKNOWN	FALSE

Back to keywords:

A left join is kinda like a cartesian product but with the added perk that if a certain row can't have a certain attribute instead of throwing it away, you just put null values as placeholders. A right join does the same but for the right side of the operation.

To the right is an example of a left join. The left join matches up the no key and then all the rows where the movement amount is less than zero. For all the rows where it doesn't satisfy this, it returns a null THIS ONLY APPLIES TO PREDICATES IN THE AND IF IT IS THE WHERE THEN IT JUST GETS THROWN AWAY.

```
SELECT  account.no,
        movement.amount
FROM    account LEFT JOIN movement
                ON    account.no=movement.no
                AND   movement.amount<0
```

If you think that I am gonna do all the fancy inner joins and outer joins you're wrong.

ER Modelling:

Designing relational databases:

Up until now we have looked at doing queries on relational databases. We're gonna start off with ER schemas whatever they are.

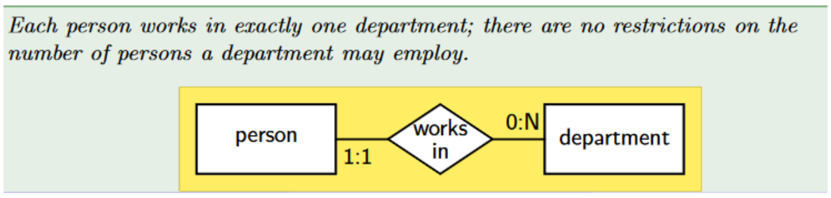
$\mathcal{E}$ represents entities in our relational database. An entity is a group of items that are of the same type. For example you can have people or shoes. In a description of a database an entity will be a noun. Proper nouns will be instances of the entity.

Now we have attributes and there are three types:

- Mandatory Attributes
  It is a function that maps items from the entity set E to the value set V. They have to be unique so if I have (e,v1) and (e,v2) that means v1= v2.
- Optional Attributes
  These are attributes that can be null.
- Key Attributes
  These are attributes where all entities have a different value.

$\mathcal{R}$ represents relations. These are defined as tuples in which each item in the tuple is of an entity and there is some type of association between the two. In descriptions verbs will show you what the relations are.

We need to have some sort of amount relation (known as a cardinality constraint) for each entity set in a relation.

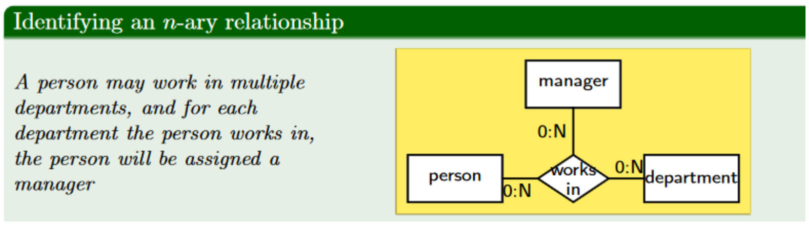


Each person works in exactly one department; there are no restrictions on the number of persons a department may employ.

The number on the left signifies the lower bound on the relation and the number on the right signifies the upper bound.
We also have subsets, represented by a fancy S to the power of the ER. A subset is shown by an arrow going from one entity to another and the arrow points to the superset. Entites in the subset have all of the same attributes from the smaller set.

We can have some sticky n-ary relationships

Nested relationship comes from a description that implies a relationship on a relationship.



Identifying an n-ary relationship

A person may work in multiple departments, and for each department the person works in, the person will be assigned a manager

Functional Dependencies:

An fd in a table is two attributes that are represented X -> Y where if I have two X's that are the same then the Y's must be the same.
Here are some rules:

- If we have attributes X and Y and X ⊆Y then X -> Y (reflexivity)
- If I know that X -> Y then I can say that XZ -> YZ (augmentation)
- Also, if I have X->Z and Z->Y then we also have X->Y (transitivity)
- If we have X->Y and Y->Z we can then have XZ->YZ (transitivity) and also have X->XZ (augmentation) so by transitivity we have X->YZ which is effectively a union. X->Y and X->Z give X->YZ.
- We can use reflexivity and transitivity to get X->Y and WY -> Z and have XW->WY and get XW-> Z.
- And finally, if we have X->Y Y⊆Z then we have X->Z

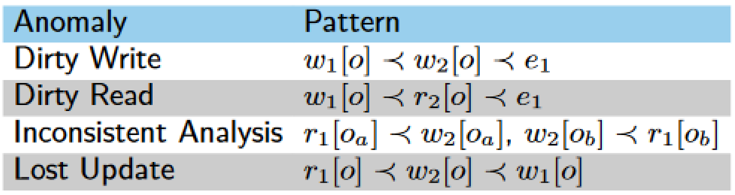
From FDs we can get keys of a relation. If we have a set of attributes that can infer the rest of the relation then they are a super key. If I can't remove any attribute and infer all of the attributes then I have a minimum key.

Minimal Cover:

To get a minimal cover, you need to start off by making sure all the FDs have only one attribute on the RHS. So just break them up. Then you need to look at any FD with two attributes on the right and then get the closure of all the individual attributes and if it includes any attrivute that is also on the LHS you can remove it. After that you need to look at all the FDs and get the closure of the LHS but not using the FD you are looking at. If you can find the RHS in the closure then you can remove it as well. Then you have the minimal cover.
Now let's talk about 1st normal form. This is when all of the attributes come from one key.
Now we move onto prime and non-prime attributes. A prime attribute is an attribute that is in a candidate key. We can generate a set of relations that are in 3NF by looking at our set of FDs. First we need to make sure that our set of FDs is a minimal cover. Then we look at the set of FDs (in the from X->A) and we see if there are any that break 3NF. If they do then we can decompose them into R_1(Attr(R) – A)) and R_2(XA).
Let's talk about getting our FDs to decompose. We want them to join back up without adding in any phantom columns. We do this by only decomposing on relations. The tactic is to find an FD and then write the FD as a relation and what you should be left over with is another relation where the only missing attributes are the ones on the right side of the FD.
Boyce-Codd NF is just when everything is inferred from the super key.

FINAL TOPIC: CONCURRENCY

Now we can make concurrent databases. Great. So we have four types of anomalies for this stuff. If you read an object after another transaction has written to it without that transaction ending that is a dirty read there's also a dirty write. You have lost update when an object is read by a transaction and then changed by another and then written by the first transaction. To the side is a chart describing them all. The weird less than sign just means and then after.



Anomaly	Pattern
Dirty Write	$w_1[o] \prec w_2[o] \prec e_1$
Dirty Read	$w_1[o] \prec r_2[o] \prec e_1$
Inconsistent Analysis	$r_1[o_a] \prec w_2[o_a],\ w_2[o_b] \prec r_1[o_b]$
Lost Update	$r_1[o] \prec w_2[o] \prec w_1[o]$

Now we move onto conflicts. There are two conditions for conflicts. Either a transaction reads an object and then before the transaction writes to it another transaction writes to it. And then the other one that is when a transaction writes to an object and another transaction writes to the same object. We can find conflicts in a history and we can use them to see if a history is serializable by making a directed graph of all the conflicts and if there exists a cycle in the graph then it is not serializable.