Lecture 1

Every modern application has Large amount of data

This needs to be

- Stored (typically on a disk drive)
- Manipulated (efficiently and usefully)
- Shared (by many users concurrently)
    - THESE ARE ALL HANDLED BY databases
- Transmitted (all around the internet)
    - This is handled by networks

We use databases to deal with LARGE data.

Challenges in building effective databases: efficiency, security, scalability, maintainability, availability, integration, new media types (e.g., music), ...

The field of databases deals with

- data representing the application scenarios
    - e.g. YouTube videos on YouTube
- relationships amongst data items
- constraints on data and relationships
- redundancy one source for each data item
- data manipulation, declarative or procedural
- transactions, multiple actions, atomic effect
    - we want to make sure all actions only yield one effect e.g. Don't want to crash system after use withdraws money from online banking
- concurrency, multiple users sharing data
- scalability for massive amounts of data

A database system is a collection of software tools that help manipulate data.

Data can be defined as known facts that can be recorded and have no implicit (just exists) meaning.

Database is a collection of related data and the data items alone are relatively useless, we need to provide a structure for the data. We can combine tools to create a DBMS (database management system).

A database is a mix of

- DBMS
- Database systems (the database and DBMS together)

Database modelling and design

- Requirements and mapping of requirements
    - ER model
    - ER relational diagrams
- Database application development
    - SQL views, stored procedure, triggers, aggregates
    - SQLite: sqlite3 (a SQL shell)
    - PostgreSQL: psql (a SQL shell)

- Programming languages that have access to databases (PHP, ORMS)

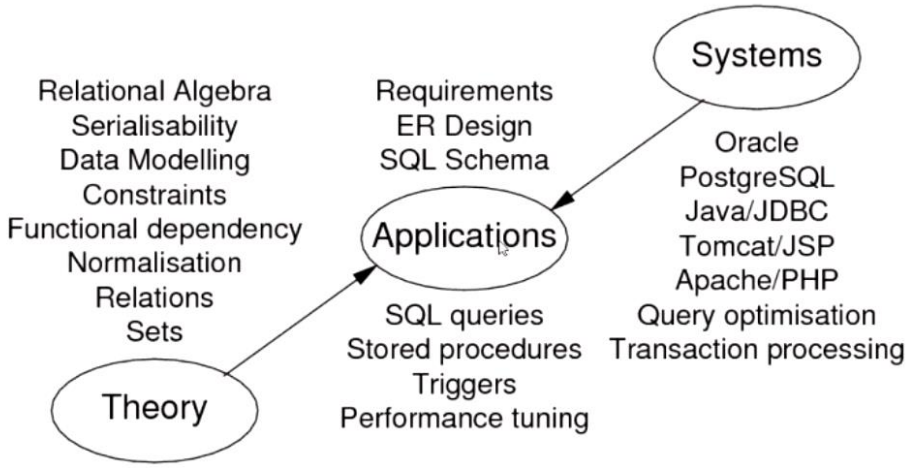
Database Management Systems (DBMS)

Comments on PostgreSQL vs Oracle:

- Oracle is resource-intensive (>800MB vs <200MB for PostgreSQL)
- PostgreSQL is a commercial-strength (ACID) RDBMS
  ... but, being open source, you can see how it works
- PostgreSQL has been object-relational longer than Oracle
  ... and its extensibility model is better than Oracle's
- PostgreSQL is more flexible than Oracle
  ... allows stored procedures via a range of programming languages

But note: PostgreSQL and Oracle have very close SQL and PL/SQL languages

Note make sure compatible with PSQL9.4 PHP 5.4 and SQLite 3.8

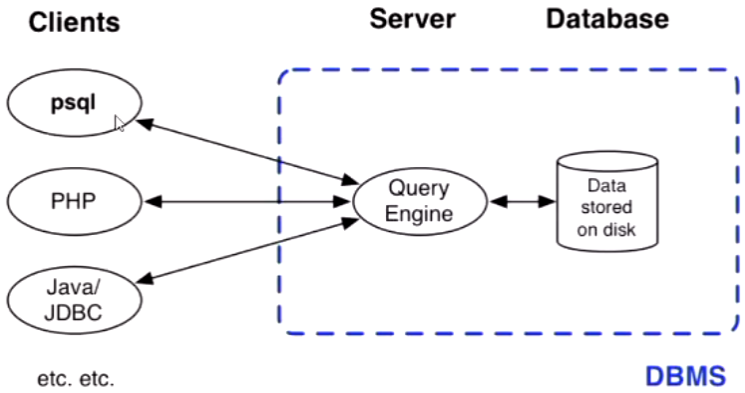


When developing a database application

1. Analyse application requirement
2. Develop a data model to meet the requirements (ER)
3. Define operations (transactions) on this model
4. Implement the data model as relational schema
5. Implement the transactions via SQL and procedural PLs
6. Construct an interface to these transactions
7. Populate database

Architecture

The typical environment for a modern DBMS is:



All clients use the same query engine to retrieve data stored on the disk.

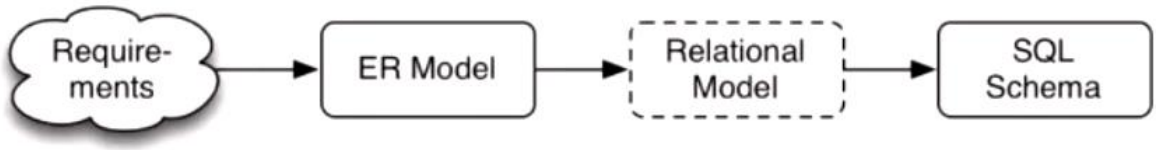
Data Modelling

- We want to describe what information is inside the database, relationships between data and constraints on the data.
- Data modelling is a design process
    - Converts requirements into a data model

We have different kinds of data models

- Logical model (abstract for conceptual design for clients
- Physical: recorder based for implementation, relational for programmers

We want to convert our requirements to an ER model and then convert it into a relational model which can be transformed into a SQL schema



The schema is our physical model which will be the structure/skeleton of our database.

There is no “best” design for a given application

Most important aspects of a design (data model)

- Correctness (satisfies requirements accurately)
- Completeness (all reqs covered, all assumptions explicit)
- Consistency (no contradictory statements)

We also want to not have

- Omit information that needs to be included
- Contain redundant information which leads to inconsistency
- Inefficiency

ER models

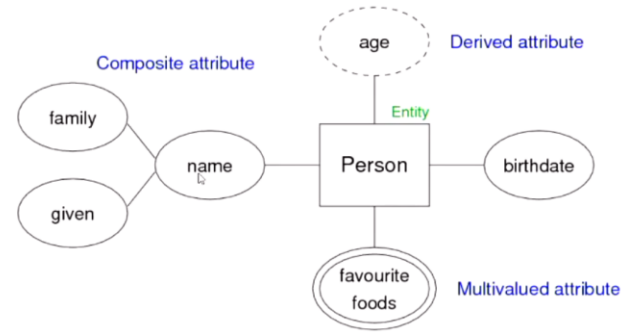
ER has three major modelling constructs

- Attributes
    - Data item describing property of interest e.g. Name, address etc
- Entity
    - Collection of attributes describing object of interest
- Relationship
    - Association between entities (objects)

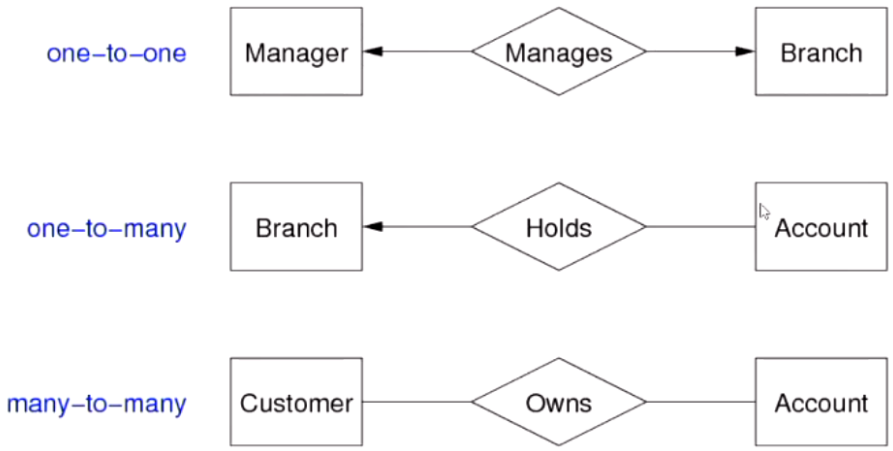
It provides a graphical tool for modelling data.

PROPERTIES ARE AS FOLLOWS

- Circles = attributes able to be composite too
- Square = entity
- Double circle (multivalued attribute) can have multiple values like a list
- Derived attributes (dotted circles) can be calculated doesn't need to be stored
- Diamond boxes are relationships
- Thick line = total participation, thin line = partial participation.



relationship cardinality



An entity set can be viewed as either:

- a set of entities with the same set of attributes (extensional)
- an abstract description of a class of entities (intensional)

Key (superkey): any set of attributes

- whose set of values are distinct over entity set
- natural (e.g., name+address+birthday) or artificial (e.g., SSN)

Candidate key = minimal superkey (no subset is a key)

Primary key = candidate key chosen by DB designer

Keys are indicated in ER diagrams by underlining

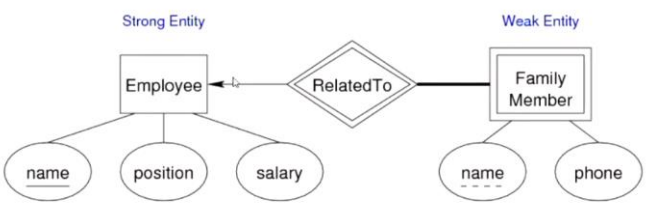
Keys are the foundations of relational databases. A key is an identifier to a unique tuple inside the database. SSN is used to be a distinct key to distinguish 100% between people or license plates. Candidate key is the minimal keys that need to identify a tuple it's the minimal key to identify a tuple in a database. We can have multiple candidate keys so a primary key is the candidate key chosen by the database designer. We can have phone number, email etc. which are all enough to identify a row, we have to choose one. Primary keys are underlined in our design.

Double diamond/double rectangle represents weak strong coupling. The dashed line in name on the weak entity is our discriminator for our ERD. If we remove our strong entity, we also need to remove the weak entity.

Weak entities

- exist only because of association with strong entities.
- have no key of their own; have a discriminator

Example:



Given an employee name and a family member name we can determine a row on DB

Lecture 2

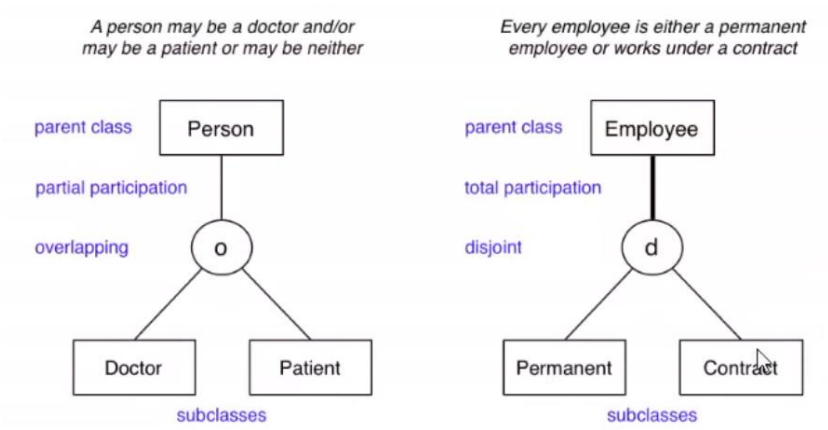
Subclasses and inheritance

A subclass of an entity is a set of entities which have all attributes of A and its own attributes. It is involved in all of A's relationships and its own.

Properties of subclasses

- Overlapping or disjoint
- Total or partial

For example (o = overlap, d = disjoint)



Thicker line means total participation, thin line means partial participation.

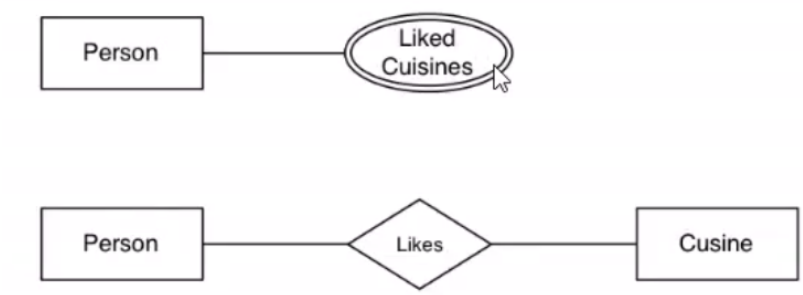
ER Models

Considerations for ER models

- Should an object be an attribute or entity?
- Is a concept best expressed as an entity or relationship?
- Should we use n-way relationship or several 2-way relationships?
- Is an object a strong or weak entity?
- Are there subclasses or superclasses within the entities

These are all different depending on the system we are modelling.

For example, two different models representing the same thing



We can select one or the other based on the application we are making. If our application is based on the person solely the first option is more viable, since it is a property of the person, however if

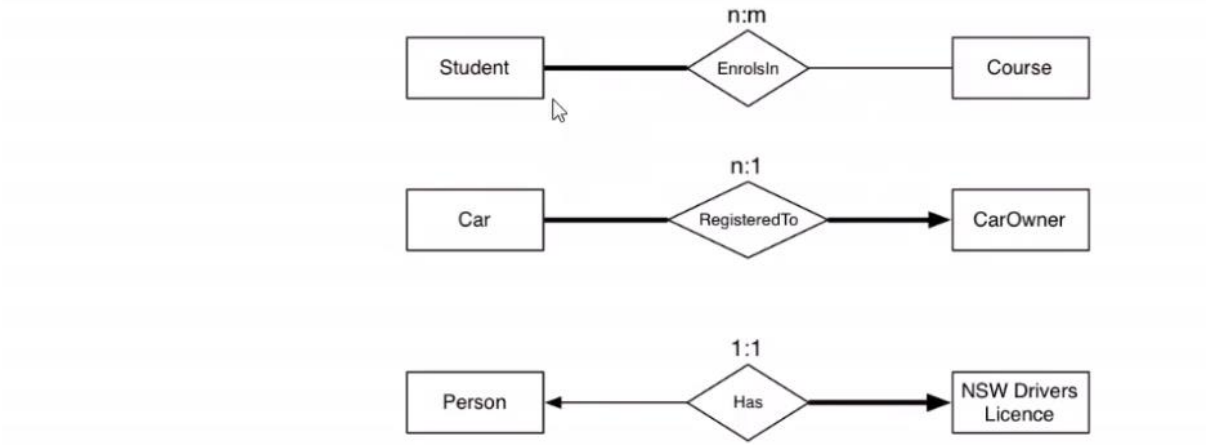
our database was based on cuisine or not based on the person, the second ERD would be preferable. Because then we can put properties to our foods. This goes to show that our ERD is based specifically on client needs.

In real world scenarios ER diagrams are too large to fit on a single screen. We usually link different pages together. One technique is to

- Define entity sets separately with just attributes
- Combine entities and relationships on a single diagram without showing attributes to save space
- If the design is very large, we can use several linked diagrams

Exercises

Describe the precise semantics of each of the following:



1. All students enrol in 0 to many courses, not all courses have students enrolled
2. All cars have one car owner, a car owner can own 0 to many cars
3. One person owns one NSW drivers license, all NSW drivers' licenses are owned by 1 person

Many (more than 1) = no arrow

Thick line = total participation, thin line means some participation.

Exercise: Medical Information

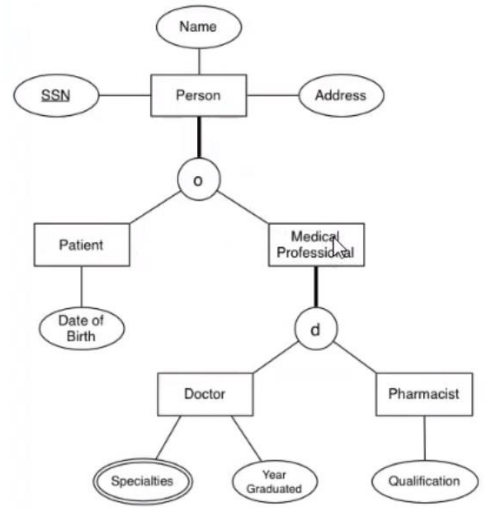
Develop an ER design for the following scenario:

- Patients are identified by an SSN, and their name, address and date of birth must be recorded.
- Doctors are identified by an SSN. For each doctor, we record their name, address, specialties and year of graduation.
- A pharmacist is identified by an SSN, and we also record their name, address and qualification.
- A medical professional is either a doctor or pharmacist, but not both. Anyone can be a patient.
- For each drug, the trade name and formula must be recorded.
- Every patient has a primary physician. Every doctor has at least one patient.
- Each pharmacy has a name, address and phone number. A pharmacy must have a manager (who is a pharmacist). A pharmacist cannot work in or manage more than one pharmacy.
- Each pharmacy sells several drugs, and has a price for each. A drug could be sold at several pharmacies, and the price could vary between pharmacies.
- Doctors prescribe drugs for patients. A doctor could prescribe one or more drugs for several patients, and a patient could obtain prescriptions from several doctors. Each prescription has a date and quantity associated with it.

[Solution]

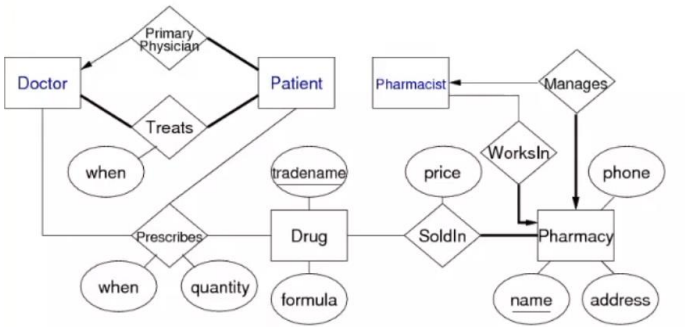
Solution

People subclasses



Now that we have our definitions for our entities, we can then put our relationships without including attributes to save space

Relationships



Also note we use common sense to include things that were left out of the requirements that are useful e.g. the treats relationship needs the when attribute to know when the patient was treated.

Do this on my own!

Exercise: Book Publishing

Develop an ER design for the following scenario:

- for each person, we need to record their tax file number (TFN), their real name, and their address
- authors write books, and may publish books using a "pen-name" (a name, different to their real name, which they use as author of books)
- editors ensure that books are written in a manner that is suitable for publication
- every editor works for just one publisher
- editors and authors have quite different skills; someone who is an editor cannot be an author, and vice versa
- a book may have several authors, just one author, or no authors (published anonymously)
- every book has one editor assigned to it, who liaises with the author(s) in getting the book ready for publication
- each book has a title, and an edition number (e.g. 1st, 2nd, 3rd)
- each published book is assigned a unique 13-digit number (its ISBN); different editions of the same book will have different ISBNs
- publishers are companies that publish (market/distribute) books
- each publisher is required to have a unique Australian business number (ABN)
- a publisher also has a name and address that need to be recorded
- a particular edition of a book is published by exactly one publisher

[Solution]

Relational Data Model

Relational data model describes things as inter-connected relations (tables).

The goal is to

- A simple general data modelling formalism which maps easily to file structures

Relational model terminology can be

- Mathematical: tuple, relation, attribute
- Data-oriented: table, record, field/column

A relation model has one structuring mechanism

- A relation corresponds to a mathematical relation
- A relation can also be viewed as a table

Each relation has

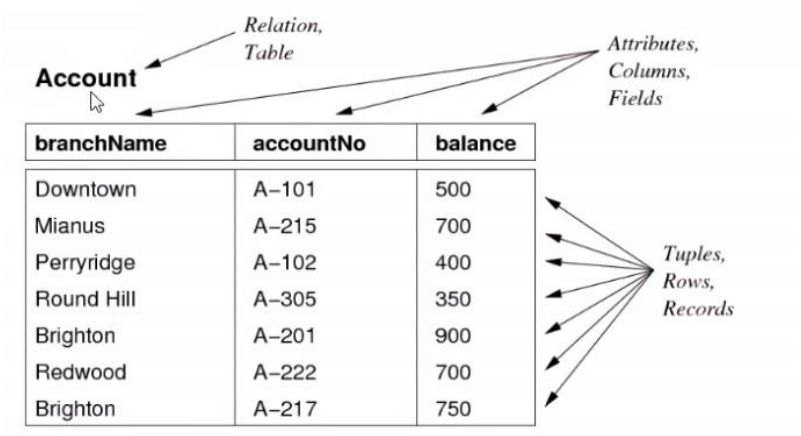
- A name
- A set of attributes (table name + rows)

Each attribute has

- A name (row name)
- An associate domain (allowed values)

DB uses logical constraints to restrict values.

Example Bank Account



branchName	accountNo	balance
Downtown	A–101	500
Mianus	A–215	700
Perryridge	A–102	400
Round Hill	A–305	350
Brighton	A–201	900
Redwood	A–222	700
Brighton	A–217	750

In a database

- All attribute values are atomic (no composite or multi-valued attributes)
- A special value NULL (literally definition of NULL)
- Each relation has a key (subset of attributes unique for each tuple)

Consider relation R with attributes $a_1, a_2, \ldots a_n$

Relation schema of R : $\boldsymbol{R}(a_1{:}D_1, a_2{:}D_2, \ldots a_n{:}D_n)$

Tuple of R : an element of $D_1 \times D_2 \times \ldots \times D_n$ (i.e. list of values)

Instance of R : subset of $D_1 \times D_2 \times \ldots \times D_n$ (i.e. set of tuples)

Database schema : a collection of relation schemas.

Database (instance) : a collection of relation instances.

In a mathematical sense

- Relation schema is a relation between an attribute which can be abstracted to a type
- Tuple is an element of all the types together (list of elements)
- An instance is a database filled with values

Relational database management systems (RDBMS)

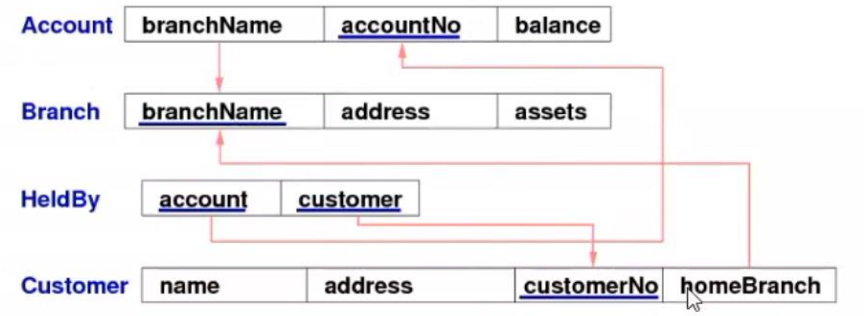
- Uses sql as language for data definition, queries and updates
    - Relations are not required to have a key (can have duplicates)
    - Relations are bags rather than sets (collections)
- SQL does not maintain a set for efficiency sake, O(n) for every query is too slow, and all insertion is also too slow
    - If application needs no duplicates, we use DISTINCT
    - Distinct will also sort the result, since the fastest way to get distinct is to sort and check neighbours
- Implements relation model

DBMS have multiple namespaces:

- DBMS-level (database names must be unique)
- Database level (schema names must be unique)
- Schema-level (table names must be unique)
- Table level (attribute names must be unique)

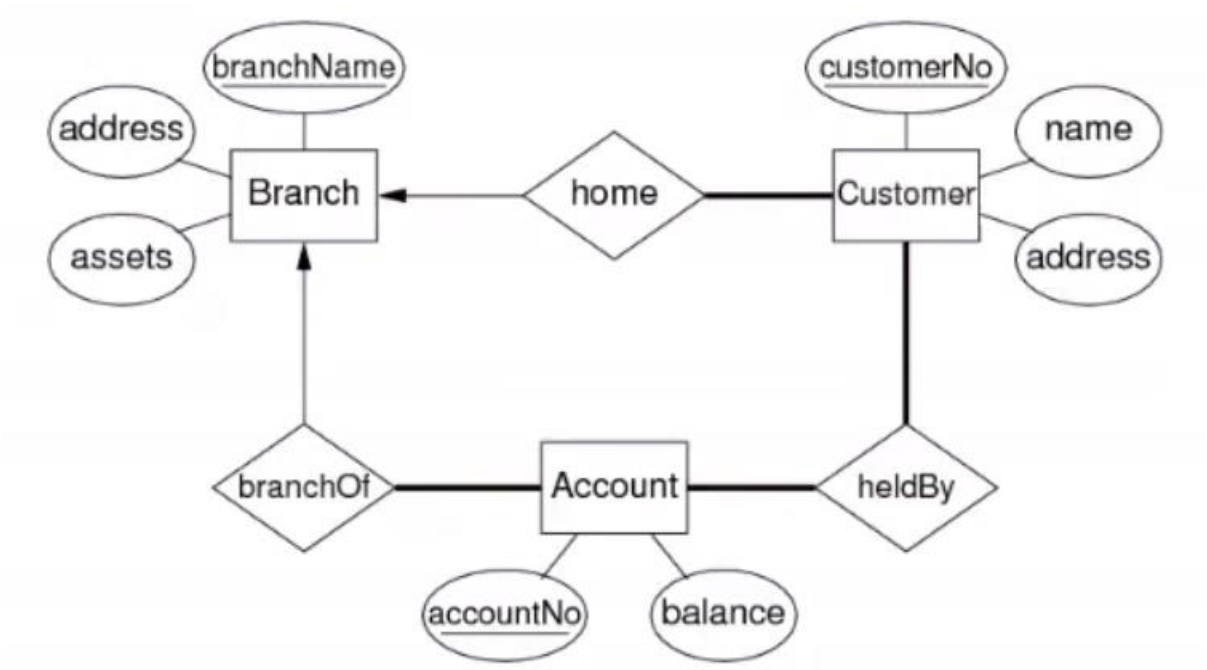
It can be convenient to use the same name in several tables e.g. Id is a good primary key. We distinguish which attribute we use the table name, e.g. Movie.id is different to actor.id.

Database schema. Primary key is underlined since they are unique identifiers. Sometimes we need two attributes to create a primary key set since one attribute might not be enough to distinguish. In example the Held By table might have multiple same accounts or multiple same customers.



In the held by table since it is a many-many relationship we need both columns to be primary keys. Searches are much faster for primary keys since they are indexed and hashed/stored in a tree.

The corresponding ER diagram for this database system is the following



First, we get our requirements, then we create a corresponding schema. Finally, we map out the ERD to display to our client that this is what they want.

Domains

Domains will limit the set of values attributes can take, we need to describe

- What values are and aren't allowed
- What combinations of values are/aren't allowed?

Constraints are logical statements to do this

- Domain
- Key
- Entity
- Integrity
- Referential integrity

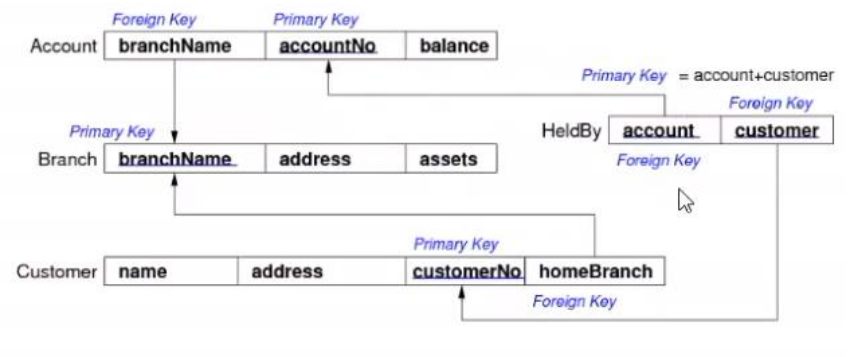
Example

- **Data constraints**
  - Employee. Age is typically an integer
  - Can add extra constraint (15 < age < 66)
- **Key constraints**
  - Key constraints Student (id, …) guarantees unique id key
  - Class (…, day, time, location …) guarantees unique tuple keys (the day, time, location) tuple all unique
- **Entity Integrity**
  - Class (…, Mon, 2pm, Lyre, …) is well defined
  - Class (…, NULL, 2pm, Lyre, …) is not well-defined

Referential

We can define references between tables by using the notion of a foreign key (FK)

Example



These are rows that link two different tables together on unique values. We normally perform joins using the foreign key with another tables primary key.

Foreign Keys

A set of attributes F in relation R1 is a foreign key for R2 IF

- The attributes in F correspond to the primary key of R2
    - Example actor id and acting.actor_id is same value
- The value for F in each tuple of R1
    - Occurs as a primary key in R2
    - Or is entirely NULL
- Foreign keys are critical in RDBMS
    - They link tables together like glue
    - They allow us to perform joins between tables

Summary

A relational database schema is

- A set of relation schemas {R1, R2, …., Rn} and
- A set of integrity constraints

A relational database INSTANCE is

- A set of relation instances {r1(R1), r2(R2), …, rn (Rn)}
- Where all the constraints are satisfied from the schema

One of the most important functions of a RDBMS is to ensure that all data is satisfying the constraints of our requirements.

Lecture 3

Foreign key constraint says that we cannot add a foreign key that is not a primary key in the corresponding table. For example, if we have a soccer database, and we have a player and a team, we cannot put a player in the database for a team that doesn't exist. (player has team_plays_for (foreign key) and team has id (primary key)). Database check the integrity of our data.

Key constraints make values unique for identification e.g. Id's to identify a tuple in our record set.

We sometimes create artificial keys as primary keys like id to increase efficiency, it's a lot faster than using natural list of keys as a primary key.

Foreign keys allow us to prevent redundancy by using foreign keys to reference other tables.

If we want to formalise our relation schema, SQL provides a Data Definition Language (DDL) for this.

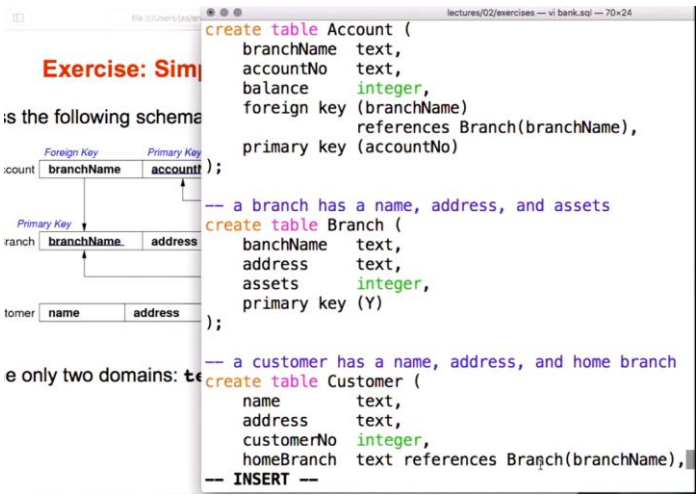
- We give the name for the table
- Create attributes for table
- We give the domain and constraints on the attributes for table
- Then we add primary and foreign keys

```
CREATE TABLE TableName (
    attrName1 domain1 constraints1 ,
    attrName2 domain2 constraints2 ,
    ...
    PRIMARY KEY (attri,attrj,...)
    FOREIGN KEY (attrx,attry,...)
                REFERENCES
                OtherTable (attrm,attrn,...)
);
```

SQL syntax

- Identifiers the " " quotes allow us to create identifiers
- Reserved words such as create, select, table, replace
- Strings ' ' quotes, does not have '\n'
- Numeric
- Types: integer, floats, char(n), varchar(n), date, time, timestamp, text
    - Char(10) creates a fixed length string that has 10 chars, char[n] in c (pads rest with spaces)
    - Varchar (10) allows us to give upper bound for maximum length of string, and if our string is less than the upper bound, we only allocate for the current length, for example
        - Varchar (20) s = 'poo'
        - Since poo has 3 characters, we only allocate for 3 characters rather then pad with spaces like char does.
- Operators: =, <>, <, <=, >, >=, AND, OR, NOT, IS NULL, WHERE
- This is DBMS specific

For the bank account example, we can create the following sql schema.



Every table needs a primary key.

We can also enforce other constraints e.g.

- Don't allow overdraw
- Customer numbers seven-digit integers
- Account numbers are in the form a-xxx
- The assets of the branch are the sum of the balances in all the accounts held in that branch.

To do this we can add constraints inside our table shown below

```
create table Account (
    branchName  text,
    accountNo   text check (accountNo ~ '[A-Z]-[0-9]{3}'),
    balance     integer,
    foreign key (branchName)
                references Branch(branchName),
    primary key (accountNo)
);
```

We can even create domains for our variable types. (note first one is typo needs check like 2nd one). Defining domains allows consistent usage of same variable types and restriction across multiple tables.

```
create domain AcctNumType as
    char(5) check (value ~ '[A-Z]-[0-9]{3}');
create domain CustNumType as
    char(7) check (value ~ '[0-9]{7}');
-- alternatively
create domain AcctNumType as
    text check (value ~ '^[A-Z]-[0-9]{3}$');
create domain CustNumType as
    text check (value ~ '^[0-9]{7}$');

-- accounts are held at branches and have a balance
create table Account (
    branchName  text,
    accountNo   AcctNumType,
    balance     integer check (balance >= 0),
    foreign key (branchName) references Branch(branchName),
    primary key (accountNo)
);
```

We can also define constraints for our datatype outside the schema.

```
-- Another example ...
create table t (
    x integer,
    y integer,
    constraint xBiggerThanY check (x > y)
);
```

\dt will show all tables in our database

\d(table) shows the specific table)

NOTE we can use an extra constraint to make sure data is input in the following

```
create table Customer (
    custNo      integer,
    name        text not null,
    address     text
    primary key (XX)
);
```

Mapping ER designs to Relational Schemas

For database design a useful approach could be

- Perform initial data modelling using ER or OO (conceptual)
    - Early stage for design avoids bad designs!
- Transform conceptual design into relational model (implementation)

Correspondences between relational and ER data models:

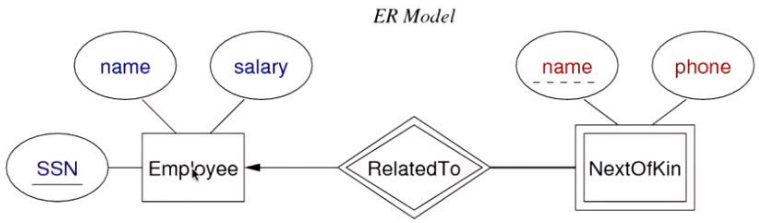
- attribute(ER) ≅ attribute(Rel), entity(ER) ≅ tuple(Rel)
- entity set(ER) ≅ relation(Rel), relationship(ER) ≅ relation(Rel)

Differences between relational and ER models:

- Rel uses relations to model entities *and* relationships
- Rel has no composite or multi-valued attributes (only atomic)
- Rel has no object-oriented notions (e.g. subclasses, inheritance)

Example

Example:



Relational Version

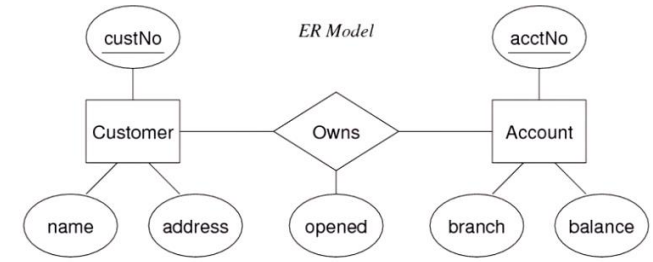
Employee

SSN	name	salary

NextOfKin

SSN	name	phone

Another n:m example



Customer

custNo	name	address

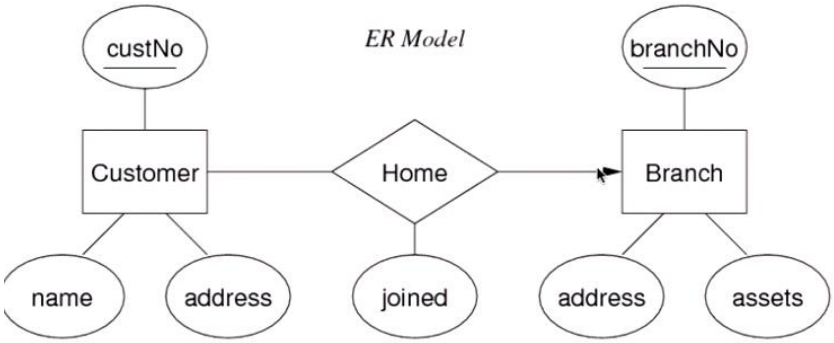
Account

acctNo	branch	balance

Owns

custNo	acctNo	opened

1: n relationship example



Customer

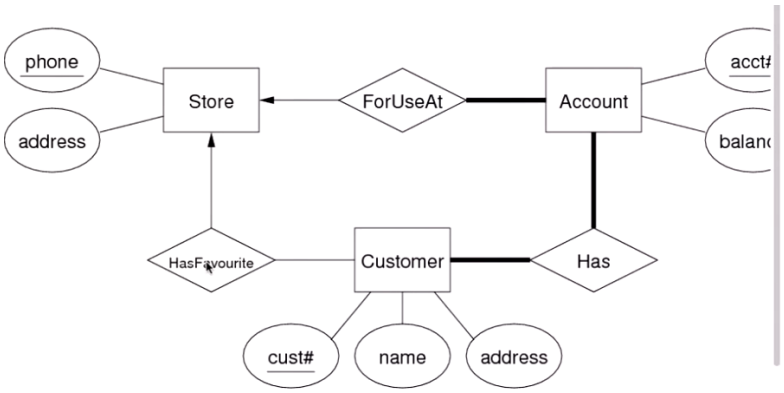
custNo	name	address	branchNo	joined

Branch

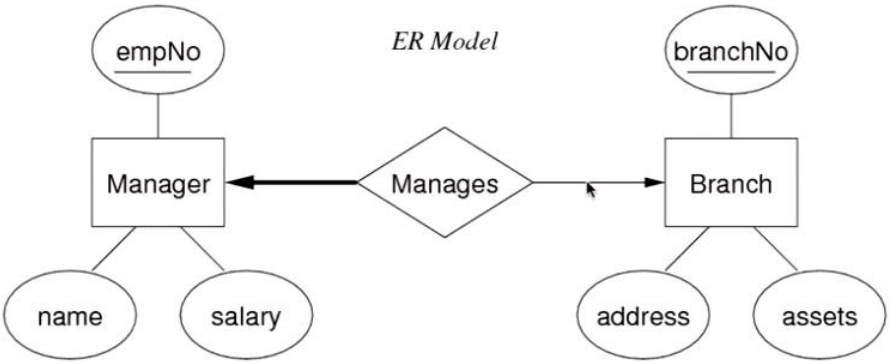
branchNo	address	assets

So, in this case we put the foreign key into the customer since each customer only associated to one branch.

Shopping Database



1:1 example



The side with total participation takes the foreign key! This is why we don't include list of managers for the branch.

Manager

empNo	name	salary	branchNo

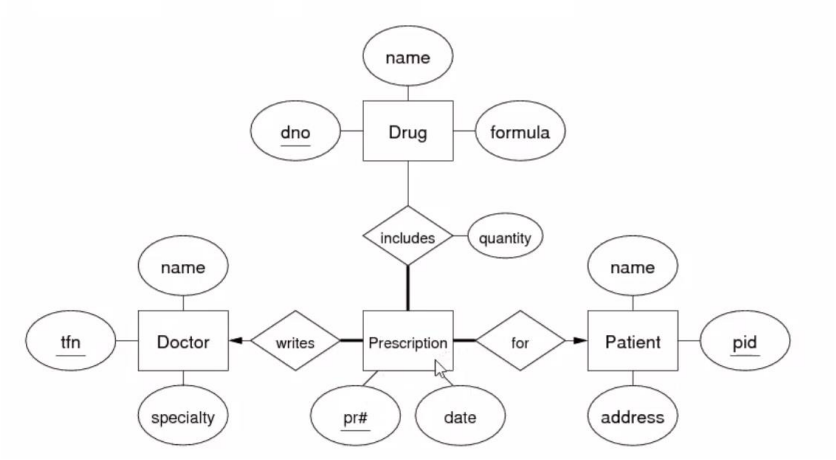
Branch

branchNo	address	assets

Lecture 4

When we have a 1: n:m relationship, we still need a separate table to map the relationship.

Advanced example



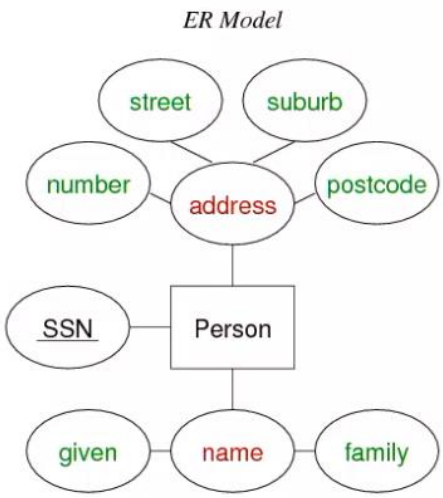
Any n-way relationship can be represented by multiple binary relationships. However, this will come with the implication of creating more tables.

In the above we have 4 tables each for our entities, then since prescription->doctor is many-1 relationship the doctor tfn is in the prescription table, similarly for prescription->patient the patient id is in the prescription table, however since prescription-drug is a many:many relationship, we need to create another table for the includes relationship.

Total: 5 tables

Mapping Composite Attributes

Since composite attributes cannot be done in sql (no array types), we can represent them by either, concatenation or flattening.



Relational Version #1

Person

SSN	name	address

Relational Version #2

Person

SSN	givenName	familyName	...

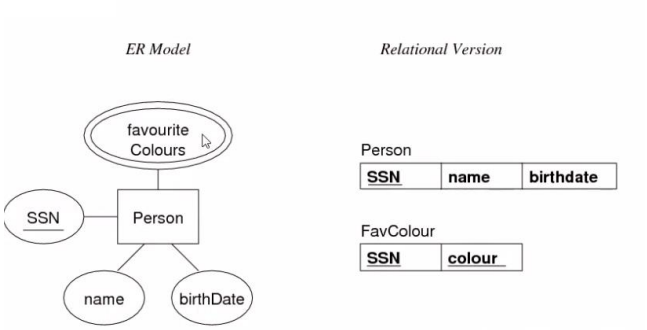
...	number	street	suburb	postcode

- We can split the attributes up individually e.g. Address → suburb, street, number, postcode
- Or we can concatenate each individual sub attribute together. There is a trade-off for each approach
    - Updating details is a lot harder for composite approach, and also makes it a lot harder for queries e.g. Finding countries with postcode in certain regions. It also makes sorting by certain aspects of address harder, e.g. Sort by addresses in postcode 2222. Filtering is almost impossible for concatenating. (we can do smarter concatenation). However, by concatenating we slow down queries because we have to match substrings which requires more computation.
    - But by flattening we have to, combine attributes to create an address, if we separate too much the information might not be stored in right field when we perform queries, e.g. If we want name search for all people who have bob in their name, we usually search first and last name not middle name.

In database systems, size is never the issue, the main issue is speed, we can buy size but we cannot buy speed. If we tell our boss we can upgrade DB by buying more space they won't care, but if we can say we can speed up queries by 5x more impressive.

Multi-valued Attributes (MVA)

MVA's are mapped by a new table linking values to their entity



Now in our favourite colour table we can have MULTIPLE matches to the same SSN as the primary key is the combination of SSN and colour. To match a person to his favourite colours we can use JOINs and SQL queries to perform this. Databases are atomic so we can't store multiple values in one attribute

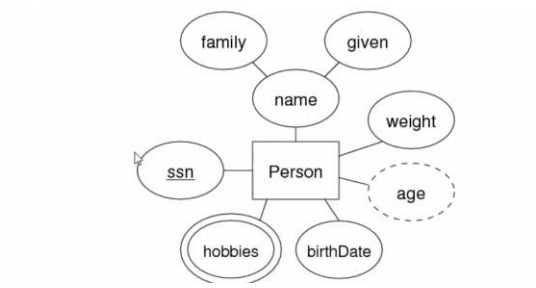
Example: the two entities

```
Person(12345, John, 12-feb-1990, [red,green,blue])
Person(54321, Jane, 25-dec-1990, [green,purple])
```

would be represented as

```
Person(12345, John, 12-feb-1990)
Person(54321, Jane, 25-dec-1990)
FavColour(12345, red)
FavColour(12345, green)
FavColour(12345, blue)
FavColour(54321, green)
FavColour(54321, purple)
```

Convert this ER design to relational form:

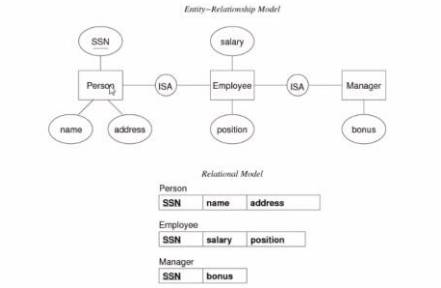


Note age is a derived attribute from birthdate. 4 attributes if we combine name, 5 if we don't combine name. We don't store derived attributes in our table, we can derive them inside queries.

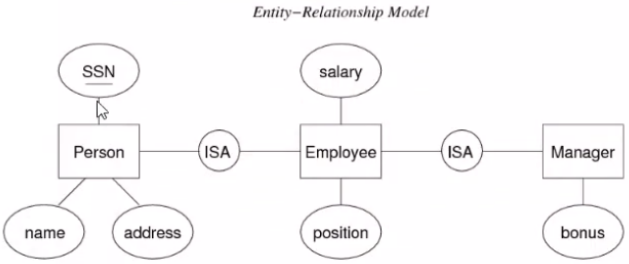
Mapping Subclasses

Three different styles

- ER style
  - Each entity becomes a separate table
  - Containing attributes of subclass + FK to highest superclass
  - Example of ER-style mapping:



  - Note we store the foreign key SSN to all tables since we can reference all these data from SSN by performing joins
- Object-oriented
  - Each entity becomes a separate table
  - Inherits all attributes from all superclasses



Relational Model

Person

SSN	name	address

Employee

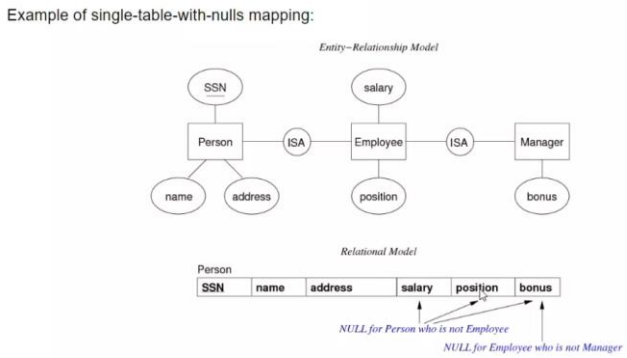
SSN	name	address	salary	position

Manager

SSN	name	address	salary	position	bonus

- Single table with nulls
    - Whole class hierarchy becomes one table
    - Containing all attributes of subclasses (null if unused)
    - One giga table

Example of single-table-with-nulls mapping:



    - NULL value if not there e.g. If null bonus means not manager!

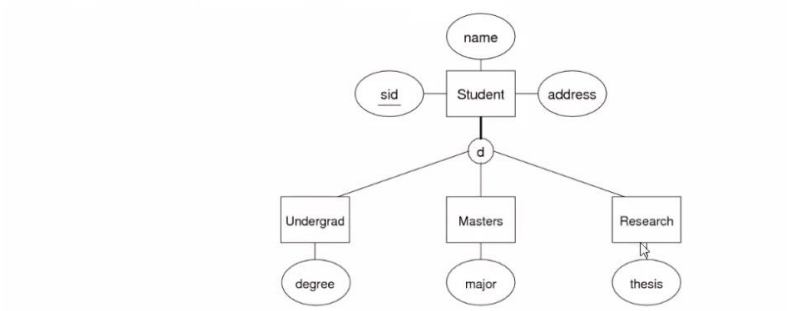
Advantages and disadvantages

- Single table is good in cases when you want to look up specifics for a certain subclass, because all the information is present there e.g. Find whether a manager lives in barker street, since we do our look up directly, we can check if not null for bonus and do one query, on other approaches we would require joins.
- The first approach would require joins to perform queries however it also avoids redundancy duplicate information. It is also a lot easier to add a new entity e.g. add a new category CEO would require only the attributes of CEO and SSN. In all other approaches we get redundancy, since a manager is an employee and a person he will be in both tables!
- The first approach is much better than OOP approach for updating values, as with object-oriented approach we need to update values for all superclasses too but in the first approach we only update in one table since there is no redundancy

We want a balance between complexity of database and searching. The object-oriented approach is kind of bad because of redundancy but, it does allow us to perform queries on only one smaller table rather than larger tables. Each approach has its own use and times where it is better.

Disjoint subclass

For example, undergrad, masters, research



Use (a) ER-mapping, (b) OO-mapping, (c) 1-table-mapping

Are there aspects of the ER design that can't be mapped?

Total participation and we can only pick 1!

To do this we can use all 3 approaches

Exact same as above! This is also same for overlapping, it can be implemented in the exact same way. We can also enforce disjoin with triggers or with constraints!

```
      constraint DisjointTotal check
      ((degree is not null and major is null and thesis is
null)
       or
       (degree is null and major is not null and thesis is
null)
       or
       (degree is null and major is null and thesis is not
null))
```

With overlapping we just make sure all 3 fields are not null, but can allow for overlapping! We cannot force disjoint property with just schema conditions, we use triggers!! (executive example for assignment 1).

```
---- ER mapping of subclasses ----

-- as specified, only properly handles (overlapping, partial)
case
-- to make it handle other cases correctly requires triggers
```

We cannot enforce that a student must be undergrad/masters/research or that a student can only be 1 of the three subclasses, to do this we use triggers!

The only thing we can specify between tables is foreign keys. We only have referential integrity to work with

RDBMS

An RDBMS

- Software design supports high-scale data applications
- Allows high level description of data with tables and constraints
- High level access to the data in SQL
- Provides efficient storage and retrieval
- Supports multiple simultaneous users with different privileges
- Allows multiple simultaneous operations with transactions and concurrency
- Maintains reliable access to the stored data with backup and recovery.
- Persistent storage of information

The database systems implement a relational model however

- A table in the database doesn't need a requirement of keys
- Bag storage rather than set storage for more efficiency
- No standard support for multi-table constraints.

RDBMS have the following operations can be seen in Postgres or most db. systems

- Create/remove a database or schema
- Create/remove/alter tables within a schema
- Insert/delete/update tuples in a table
- Perform queries on data and also define named queries which are views
- Transactional behaviour

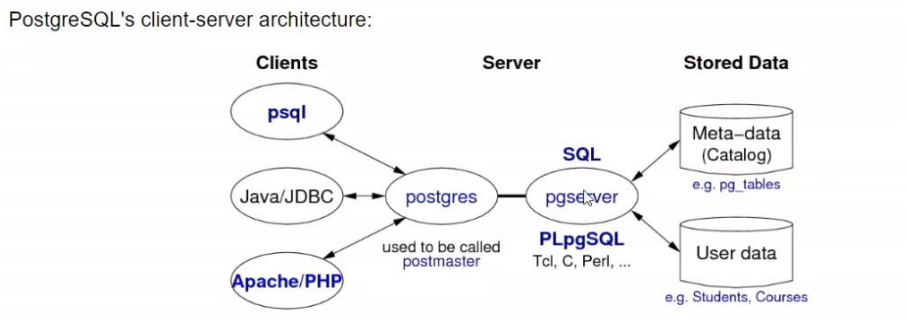
Some database also provides us ways of

- Creating/managing users of the database
- Defining storing procedural code to manipulate data
- Implementing complex constraints within triggers
- Defining new data types and operators (cancer in plpgsql)

A good programmer does not write redundant code.

Java is one of the only languages that have direct DB support, when you run java code it runs on VM which also runs on top of a database, so companies which use java can allow for management for programmers.

POSTGRESQL Architecture



Meta-data is all system related data, permissions, system controls etc.

Clients run commands with psql and the request is sent to pgserver by postgres. It is a typical client server architecture.

DB commands

In shell to create/destroy a db

- Createdb/destroydb

In SQL statements

- CREATE DATABASE/ DROP DATABASE

To dump/restore a database

- Pg_dump “filename”
  - Dumps a database into a file
- Psql “dbname” -f “dumpfile”
  - Creates a database from dump file

SQL statements (used in *dumpfile*):

- **CREATE TABLE** *table (Attributes+Constraints)*
- **ALTER TABLE** *table TableSchemaChanges*
- **COPY** *table (AttributeNames)* **FROM STDIN**

SQL statements:

- **ALTER TABLE** *table TableSchemaChanges*
- **DROP TABLE** *table(s)* [**CASCADE**]
- **TRUNCATE TABLE** *table(s)* [**CASCADE**]

(All conform to SQL standard, but all also have extensions)

DROP..CASCADE drops objects which depend on the table

TRUNCATE..CASCADE truncates tables which refer to the table

Best code always simplest - Raymond.

- Alter table
    - Changes structure of table, attributes etc. without destroying the table
    - Useful in industry if u got large db being used for weeks with millions of records
- Drop table
    - Destroys the entire table completely
- Truncate Table
    - Only gets rid of data in table, table reset
- Optional keyword cascade
    - If we try to remove a table which will break referential integrity our database will say no you can't do that, this is because a table can rely on the table, we are trying to delete
    - CASCADE will perform the operation on all tables with dependency, recursively traces back
    - Use carefully can be extremely dangerous

Lecture 5

Managing tuples with SQL

- **INSERT INTO** *table (attrs)* **VALUES** *tuple(s)*
- **DELETE FROM** *table* **WHERE** *condition*
- **UPDATE** *table* **SET** *AttrValueChanges* **WHERE** *condition*

All 3 sql commands perform the exact behaviour described in common English.

Note that attrvaluechanges is a comma-separated list of attributes we are changing.

Generating IDs (NOTE SERIAL POSTGRES)

```
create table T (
    id serial primary key,
    x  integer,
    y  varchar(10)
);
```

Postgres provides a special type known as serial which will be a unique value that updates based on the current serial's in the table. This allows us to ignore the process of generating the unique id's and let postgres manage it.

Creating unique id's is not as simple as it seems. Just doing select max(id) from T, and incrementing by 1 won't work.

Serial works very special, it is a table managed by progress that stores a unique set of values, and when we insert a value postgres' backend will then mark it as taken so we select a unique value next. To make it work with concurrency efficiently, postgres will pass around id's making sure each tuple gets a unique value, these ids are pre-determined for tuples so that is why sometimes we get gaps in our id. This is a very simple watered-down explanation but serial data type helps us avoid a lot of headaches.

Note the id does not need to be provided when inserting, we only need to insert x and y, the system takes care of id with serial.

DB objects

Databases contain objects other than tables and tuples

- Views
- Functions
- Triggers
- Sequences
- Types
- Indexes
- Roles

We use the commands to create/destroy these objects

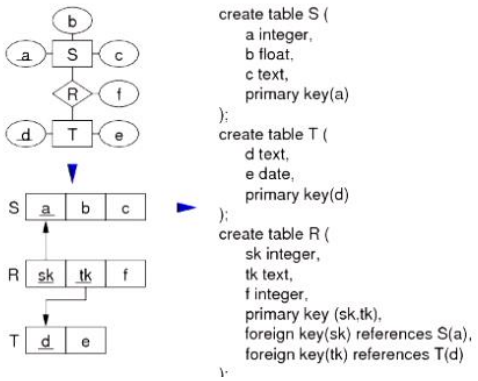
- Create object name
- Drop object name

ONLY IN POSTGRESQL

- Create or replace view/functions

If we try to create an object that already exists PostgreSQL will spew out an error. Because we can't create what exists, so what we do is drop it if it exists.

Summary of ER to Relational to SQL




```
create table S (
    a integer,
    b float,
    c text,
    primary key(a)
);
create table T (
    d text,
    e date,
    primary key(d)
);
create table R (
    sk integer,
    tk text,
    f integer,
    primary key (sk,tk),
    foreign key(sk) references S(a),
    foreign key(tk) references T(d)
);
```

SQL has several sub-languages:

- meta-data definition language (e.g. CREATE TABLE)
- meta-data update language (e.g. ALTER TABLE)
- data update language (e.g. INSERT, UPDATE, DELETE)
- query language (SQL) (e.g. SELECT)

Meta-data languages manage the schema.

Data languages manipulate (sets of) tuples.

Query languages are based on relational algebra.

The whole point of a database is to efficiently Query the data

Note in SQL that the double quotes " allow us to create identifiers that also escape keywords e.g. Date/table/create.

More SQL Language

Identifiers

- Tables
- Attributes
- Views
- Functions
- Types
- Constraints

Code conventions

- Relations names all start with capital letter
- Attribute name (camel/snake case)
- Foreign keys named based on table referenced
- Write all SQL keywords in all caps

Types

- Numeric types
  - Integer
  - Real
  - Numeric (digits, decimal)
- String types
  - Char (pads extra slots with spaces)
  - Varchar (custom allocation)
  - Text POSTGRESQL ONLY!!!!

PostgreSQL provides extended strings containing \ escapes, e.g.

```
E'\n'   E'O\'Brien'   E'[A-Z]{4}\\d{4}'   E'John'
```

Type-casting via *Expr*`::`*Type* (e.g. `'10'::integer`)

-
- Logical types
  - Boolean
- Time-related types
  - Date
  - Time
  - Timestamp
  - Interval
    - Subtraction from timestamp will give us intervals!
    - Now()::TIMESTAMP – birthdate::Timestamp
- EXTRAS FOR POSTGRESQL
  - Geometric
  - Currency
  - IP
  - XML
  - Object IDs
  - Non-standard types

We can define domains or types in the following ways

```
-- domains: constrained version of existing type

CREATE DOMAIN Name AS Type CHECK ( Constraint )

-- tuple types: defined for each table

CREATE TYPE Name AS ( AttrName AttrType, ... )

-- enumerated type: specify elements and ordering

CREATE TYPE Name AS ENUM ( 'Label', ... )
```

SELF EXERCISE

```
create domain PositiveIntegerValue as
      integer check (value > 0);

create domain PersonAge as
      integer check (value >= 0 and value <= 200);
--    integer check (value between 0 and 200);

create domain UnswCourseCode as
      char(8) check (value ~ '[A-Z]{4}[0-9]{4}');
--    text check (value ~ '^[A-Z]{4}[0-9]{4}$');

create domain UnswSID as
      char(7) check (value ~ '[0-9]{7}');
--    integer check (value >= 1000000 and value <= 9999999);

create domain Colour as
      char(7) check (value ~ '#[0-9,A-F]{6}');

create type IntegerPair as
      (x integer, y integer);

create domain UnswGradesDomain as
      char(2) check (value in ('FL','PS','CR','DN','HD'))
      -- CR < DN < FL < HD < PS

create type UnswGradesType as
      enum ('FL','PS','CR','DN','HD');
      -- FL < PS < CR < DN < HD
```

Give suitable domain definitions for the following:

- positive integers
- a person's age
- a UNSW course code
- a UNSW student/staff ID
- colours (as used in HTML/CSS)
- pairs of integers *(x,y)*
- standard UNSW grades (FL,PS,CR,DN,HD)

Note that enum is different to using text, as enum can be used to artificially create order with meaning, if we string compare PS and HD, PS > HD but we all know 50 < 85

ENUM will store as number rather than text!

From context we can distinguish between MSQL sets and tuples

INSERT INTO TABLE () VALUES () ← this is an example of tuples we are inserting

However, if we had

CONSTRAINT CHECK gender IN ('male', 'Female') ← SET

Comparison Operators

- <
- >
- <=
- >=
- =
- <> (!=)
- AND
- OR
- NOT
- IS NULL
- String1 > string2 compares alphabetically
- LIKE/ILIKE regex matching
    - % is like .* (epsilon characters around pattern)
    - _ matches any single character so like .

Note that type is very important to operation (sometimes we might need to type-cast)

Examples (using SQL92 pattern matching):

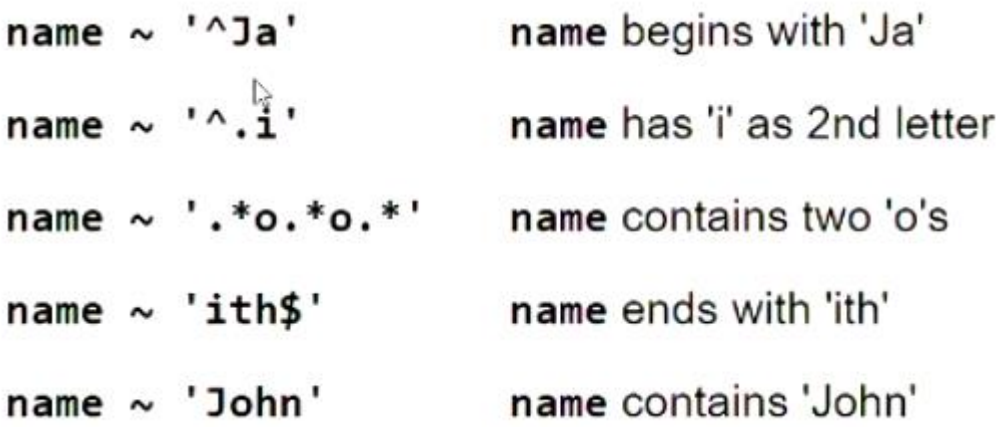
`name LIKE 'Ja%'`	`name` begins with 'Ja'
`name LIKE '_i%'`	`name` has 'i' as 2nd letter
`name LIKE '%o%o%'`	`name` contains two 'o's
`name LIKE '%ith'`	`name` ends with 'ith'
`name LIKE 'John'`	`name` equals 'John'

PostgreSQL also supports case-insensitive match: `ILIKE`

Most DBMS also allow us to use regex using

- ~ 'regex' matches regex
- !~ 'regex' matches not regex
- ~* (case insensitive)
- !~* (case insensitive)

Same example with regex



Same regex rules as grep/egrep/sed

String manipulation

- || concatenates 2 strings together → str1||str2
- Lower(str) will lowercase the value of str
- Substring (str, start, count) will extract substring between index specified
- Note null will null out any result from operator!
    - E.g. if a is null → a||' '||b will result in null since a is null

Arithmetic operators

- +/-/*/divide/abs/ceil/floor/power/sqrt/sin
- Aggregations to summarise (group) a column of numbers in a relation
    - Count(attr) → counts number of rows in the column attr
    - Sum(attr) → sum value of all attr
    - Avg(attr) → takes average value for all attr
    - Min/max(attr) → finds min/max of attr

The NULL Value

Expressions containing **NULL** generally yield **NULL**.

However, boolean expressions use three-valued logic:

a	*b*	*a* **AND** *b*	*a* **OR** *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

Think of null as unknown, this makes it a lot easier to understand

True and unknown == unknown because we don't know the value of our unknown, however true or unknown is true since one of them is true

False and unknown == FALSE since we know already know one side is false other side doesn't matter, but we can't perform or and get definite answer since 1 side is null.

To check whether a variable is or isn't null we use the special keywords

- Variable IS NULL
- Variable IS NOT NULL

Coalesce

In order to display null values on a table to a default value we can use coalesce to default to a value

- Coalesce (value1, value2, value3 … valueN) note that these can also be literal values e.g. "0"
- The first non-null value is returned
- So, we can do Coalesce (mark, 0)
    - So, if a student does not submit, we don't want to display NULL, we display 0 instead.

E.g. `select coalesce(mark,'??') from Marks ...`

Null if operator

- Nullif(val1, val2)
- Returns NULL if val1 is equal to val2
- The inverse of coalesce, dual(coalesce)

E.g. `nullif(mark,'??')`

Conditional expressions

SQL also provides a generalised conditional expression:

```
CASE
   WHEN test1 THEN result1
   WHEN test2 THEN result2
   ...
   ELSE resultn
END
```

E.g. `case when mark>=85 then 'HD' ... else '??' end`

Tests that yield **NULL** are treated as **FALSE**

If no **ELSE**, and all tests fail, **CASE** yields **NULL**

We can assign our case to our value and then test our value to all cases (sort of like a switch statement). The last ELSE is the default in our switch.

Primary Keys

Primary keys must be non null unique values to identify a tuple

E.g. CREATE TABLE R (id INTEGER PRIMARY KEY);

However we will have to do the generation of unique values in this case

MOST DBMS allow us to generate sequences of unique values to ensure that two tuples don't get assigned the same value

In Postgres

```
CREATE TABLE R ( id SERIAL PRIMARY KEY, ... );
INSERT INTO R VALUES ( DEFAULT, ...);
```

Foreign keys

Declaring foreign keys will allow the DBMS to enforce referential integrity

e.g. Account.branch text references Branch(name)

Every Account tuple must contain a value that exists within the branch name,

In a more relatable example, if we are making a soccer db and we make soccer players and assign them to a team, we have to assign them to a team that exists, it makes no sense to say soccer player belongs to a team that does not exist.

Now when we try to delete tuples from our Branch or Team

- DBMS will reject deletion to maintain referential integrity
- We could
    - Set the foreign key to be NULL in the Account/Team table
    - Cascade deletion, use with caution

Circular dependancy

Consider the following schema:

```
create table R (
   id integer primary key,
   s  char(1) references S(id)
);
create table S (
   id char(1) primary key,
   r  integer references R(id)
);
```

Devise a method to:

- load the schema
- **INSERT** data into the tables

Advanced: what if both foreign keys were **NOT NULL**.

Insert null data for references, then update after → naïve approach

One approach could be to change the schema, so we can remove the foreign key from R, and alter the table in the following

```
create table R (
      id integer primary key,
      s char(1)
);

create table S (
      id char(1) primary key,
      r integer references R(id)
);

alter table R add foreign key (s) references S(id);
```

Another thing we can do is add the keyword “deferrable” at the end behind the semi-colon. This allows us to do this

```
begin;
set constraints all deferred;
insert into R values (1,'a');
insert into S values ('a',2);
insert into R values (2,'b');
insert into S values ('b',2);
commit;
```

This will allow us to ignore the constraints until we insert ALL values. This bypasses the circular dependancies and removes the need to insert then update. This will delay the constraint checking.

Lecture 6

We can only create constraints on table based on the value of our attribute, we cannot create a constraint based on a query like shown below.

```
CREATE TABLE Example (
   gender char(1) CHECK (gender IN ('M','F')),
   Xval   integer NOT NULL,
   Yval   integer CONSTRAINT isPos CHECK (Yval > 0),
   Zval   real    DEFAULT 100.0,
   CONSTRAINT     XgtY CHECK (Xval > Yval),
   CONSTRAINT     Zcondition CHECK
                  (Zval >
                    (SELECT MAX(price) FROM Sells)
                  )
);
```

The only thing we can cross-reference to another table is foreign key references. We do this using triggers. This is because it will drastically slow down your table defying the purpose of the database.

This is also to simplify the table part and put the work on plpgsql. Since it is a constraint DBMS must uphold it.

Queries

SQL queries are high-level data manipulation. It is simply a means for retrieving data efficiently! So, we combine SQL queries with other programming languages to create complete software.

An SQL query consists of a sequence of clauses:

```
SELECT   projectionList
FROM     relations/joins
WHERE    condition
GROUP BY groupingAttributes
HAVING   groupCondition
```

FROM, **WHERE**, **GROUP BY**, **HAVING** clauses are optional.

Result of query: a relation, typically displayed as a table.

Result could be just one tuple with one attribute (i.e. one value) or even empty

Group by will group values with the same attributes specified, and then allows us to perform aggregate functions on those groups. To condense the result to one entry.

To do tasks in SQL we need to

1. Understand the schema properly and all relations
2. Look for keywords to see what is required from us
3. Check related tables to perform our queries as required

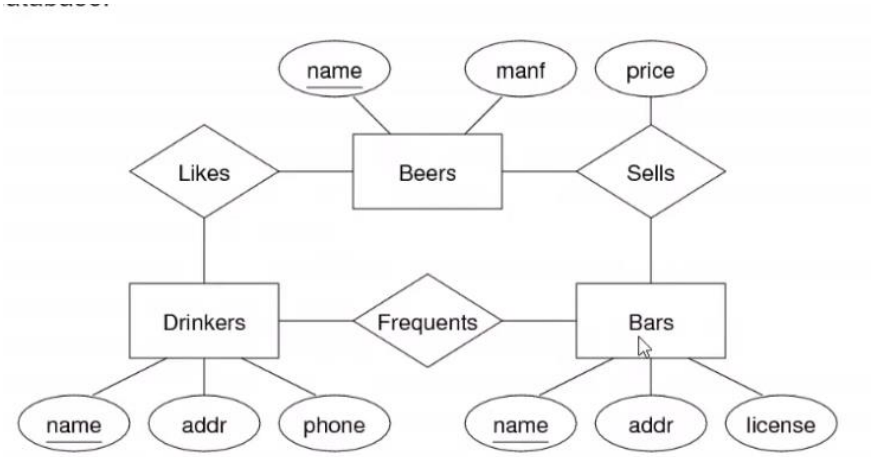
To create SQL queries, we need to

- Relate requirements to attributes in schema
- Identify the tables we need to perform our queries on
- Combine the data from relevant tables using joins
- Specify the conditions (filtering) using where clause
- Optionally we can group our results by attributes

Views

Views allow us to put a name to a query that will be called each time we access the view. Instead of creating new tables for each request, we can use a named query (view). It's like labelling a query. Views are like fake (virtual) tables.

Example Beers



1. What beers are made by Toohey's?
2. Show beers with headings "Beer", "Brewer".
3. Find the brewers whose beers John likes.
4. Find pairs of beers by the same manufacturer.
5. Find beers that are the only one by their brewer.
6. Find the beers sold at bars where John drinks.
7. How many different beers are there?
8. How many different brewers are there?

1. Select * from beers where manf = 'Toohey'
2. Skip
3. Select DISTINCT Beers.manf as "brewer" from likes join Beers on Likes.beer = Beer.name Where Likes.drinker = 'John'
4. Select DISTINCT beer1.name as beer1, beer2.name as beer2 from beer1 Beers join beer2 Beers on beer1.manf = beer2.manf Where beer1.name < beer2.name
5. Skip can do ez
6. Skip ez
7. Select count(DISTINCT *) Beers;

Joins

```
insert into R values (1,'abc');
insert into R values (2,'def');
insert into R values (3,'ghi');

create table S (
      z  char(1) primary key,
      x  integer references R(x)
);

insert into S values ('a',1);
insert into S values ('b',3);
insert into S values ('c',1);
insert into S values ('d',null);
```

Natural Joins

When we do natural join, it will automatically join based on the primary key having same value as foreign key in the other table. We do not need to specify the join condition. Natural join will assume primary and foreign key are linked together and join on that.

```
-- select * from R natural join S;
x     y     z
1     abc   a
1     abc   c
3     ghi   b
```

Inner Join (join)

Inner join will allow us to join on a condition where we can set a custom condition for our inner join.

```
-- select * from R join S on (R.x = S.x);  -- join means inner
join (inner is optional and is the default)
x     y     z     x
1     abc   a     1
1     abc   c     1
3     ghi   b     3
```

Left/right and full outer join

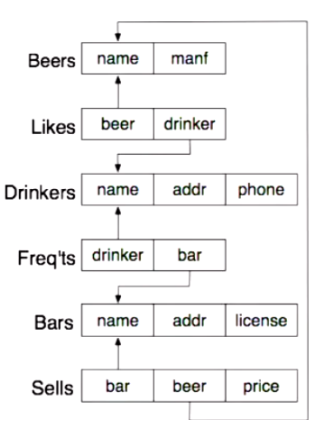
Matches all values from the left/right table and will fill in null where there is no join value

```
-- select * from R left outer join S on (R.x = S.x);  -- outer
not compulsory when left, right, and full are used
x     y     z     x
1     abc   a     1
1     abc   c     1
2     def
3     ghi   b     3
```

```
-- select * from R full outer join S on (R.x = S.x);
x     y     z     x
1     abc   a     1
1     abc   c     1
2     def
3     ghi   b     3
            d
```

Lecture 7

More Queries



More queries on the Beer database:

9. How many beers does each brewer make?
10. Which brewer makes the most beers?
11. Bars where either Gernot or John drink.
12. Bars where both Gernot and John drink.
13. Find bars that serve New at the same price as the Coogee Bay Hotel charges for VB.
14. Find the average price of common beers (i.e. served in more than two hotels).
15. Which bar sells 'New' cheapest?

8. Create or replace view beercount as Select count(name), manf from Beers Group by manf;
9. Select max(count) as max from beercount join beercount b on b.count = max, can use this as well if not like joins
    a.
```
select brewer
from   BrewersBeers
where  nbeers = (select max(nbeers) from BrewersBeers);
```
10. easy
11. easy use join/intersection (can create 1 view for gernet, 1 view for john and join
    a.
```
(select bar from frequents where drinker='Gernot')
intersect
(select bar from frequents where drinker='John');
```
    b. Can use union for previous one too. Note intersect/union remove all duplicate since set operations

A harder question, which beer is sold at all bars

```
select name
from   Beers b
where  not exists (                                    --
isEmpty
      (select name from Bars)                          -- AB
      except                                           -- set diff
      (select bar from Sells where beer = b.name)      -- BB
      );
```

My approach would be to count the number of bars, and then create a view which will count how many distinct bars a beer is sold at, and then join the two tables together on counts.

```
select beer, count(bar)
from   Sells
group  by beer
having count(bar) = (select count(*) from Bars);

-- Note: the above approach only works in some cases
-- The set-based approach works in all cases
```

← my above works

Harder: How many bars in suburbs where drinkers live

We need to use left join since we have cases where drinker does not have bar in his suburb

```
select d.addr, count(b.name)
from   Drinkers d left outer join Bars b on (d.addr = b.addr)
group  by d.addr;
```

Harder Queries self-exercise

16. Which bar is most popular? (Most drinkers)
17. Which bar is most expensive? (Highest average price)
18. Which beers are sold at all bars?
19. Price of cheapest beer at each bar?
20. Name of cheapest beer at each bar?
21. How many drinkers are in each suburb?
22. How many bars in suburbs where drinkers live? (Must include suburbs with no bars)

Stored procedures

- Functions stored in DB along with data
- Written in a language that combines SQL and procedural programming
- Provides ways to do stuff in database that can't be done in vanilla SQL
- Executed within the DBMS so it is DBMS specific

Allows us to perform procedural operations to minimise amount of code required from client. As Database management operator, we want to perform MOST the task on database side, doing insane data transfers, and harder scans is a lot slower on the client side. Data transfer is also slow.

Plpgsql is cancer just use queries lol.

```
CREATE OR REPLACE FUNCTION
   funcName(arg1, arg2, ....) RETURNS retType
AS $$
String containing function definition
$$ LANGUAGE funcDefLanguage;
```

- Function definition languages are SQL, Plpgsql, Python
- Depends on database which languages are supported

Return types

- Void
- An atomic data type, so a single integer, a text string
- A tuple (row from table)
- A set of atomic values (table columns)
- A set of tuples (table)

Note that if we are returning a set of tuples, it is almost the same as creating a view so use wisely.

```
create function factorial(integer) returns integer ...
create function EmployeeOfMonth(date) returns Employee ...
create function allSalaries() returns setof float ...
create function OlderEmployees() returns setof Employee ...
```

Their examples

```
select factorial(8);  -- returns one integer
select EmployeeOfMonth('2008-04-01');  -- returns (x,y,z)
select * from EmployeeOfMonth('2008-04-01'); -- one-row table
select * from allSalaries();  -- single-column table
select * from OlderEmployees();  -- subset of Employees
```

Note the employee of month will do the same thing since it will treat result as tuple.

These functions can also be used in views!

SQL defined functions

Arguments in sql functions can be accessed by $1, $2 and so on, or we can name our arguments and their types. In SQL the return value is the result of the last SQL statement. The return type can be any of the data types mentioned above. Generic structure below

```
CREATE OR REPLACE FUNCTION
   funcName(arg1type, arg2type, ....)
   RETURNS rettype
AS $$
   SQL statements
$$ LANGUAGE sql;
```

Example using beers

```
-- max price of specified beer
create or replace function
    maxPrice(text) returns float
as $$
select max(price) from Sells where beer = $1;
$$ language sql;

-- usage examples
select maxPrice('New');
 maxprice
----------
      2.8

select bar,price from sells
where beer='New' and price=maxPrice('New');
    bar     | price
```

Plpgsql aka cancer

Plpgsql means procedural language for PostgreSQL.

This is a mixture of procedural programming and SQL and allows us to

- Implement constraint checking e.g. triggers
- Recursive queries
- Computation for column values
- Detailed control of displayed results
- Return composite types from tables
- For loops, if statements the whole deal

Generic structure for Plpgsql function

```
CREATE OR REPLACE FUNCTION
   funcName(param1, param2, ....)
   RETURNS rettype
AS $$
DECLARE
   variable declarations
BEGIN
   code for function
END;
$$ LANGUAGE plpgsql;
```

Very simple example

old-style function ("a","b") → "a'b"

```
CREATE OR REPLACE FUNCTION
    cat(text, text) RETURNS text
AS '
DECLARE
    x alias for $1;   -- alias for parameter
    y alias for $2;   -- alias for parameter
    result text;      -- local variable
BEGIN
    result := x||''''''''||y;
    return result;
END;
' LANGUAGE 'plpgsql';
```

Add “ ‘ “ characters till it works LUL.

Newer plpgsql uses double dollar sign

```
CREATE OR REPLACE FUNCTION
    add(x text, y text) RETURNS text
AS $add$
DECLARE
    result text;      -- local variable
BEGIN
    result := x||''''||y;
    return result;
END;
$add$ LANGUAGE 'plpgsql';
```

DO NOT GIVE PARAMETERS THE SAME NAME AS ATTRIBUTES SINCE WE CAN ALSO DO QUERIES IN THE FUNCTION. GOOD PRACTICE FOR THIS COURSE START ALL PARAMETER NAMES WITH _

Factorial function in Plpgsql

```
create or replace function
      fac(n integer) returns integer
as $$
declare
      i integer;
      f integer := 1;
begin
      if (n < 1) then
            raise exception 'Invalid fac(%)',n;
      end if;
      i := 1;
      while (i <= n) loop
            f := f * i;
            i := i + 1;
      end loop;
      return f;
end;
$$ language plpgsql;
```

Function that will record interval between two numbers as table type

```
create type IntVal as ( val integer );

create or replace function
      iota(_lo int, _hi int, _step int) returns setof IntVal
as $$
declare
      i integer;
      v IntVal;
begin
      i := _lo;
      while (i <= _hi)
      loop
            v.val := i;
            return next v;
            i := i + _step;
      end loop;
      return;
end;
$$ language plpgsql;
```

Returning table type plpgsql

```
create or replace function
      iota(_lo int, _hi int) returns setof IntVal
as $$
declare
      v IntVal;
begin
      for v in select * from iota(_lo, _hi, 1)
      loop
            return next v;
      end loop;
      return;
end;
$$ language plpgsql;
```

Extending SQL

Sometimes vanilla SQL isn't enough so we can

- Create new data types
- More operations/aggregates for use in queries
- Triggers for constraint checking
- Event-based triggers
- Recursive queries

New data types

SQL provides us a way to create our own n-tuple types. Which can also be defined as enum or values. As seen above, we can create/drop types, and or even declare them as enum

Functions

Note functions have different possible modes

```
create function
  f(arg1 type1, arg2 type2, ...) returns type
as $$ ... function body ... $$
language language [ mode ];
```

Different modes

- Immutable
  - Does not access DB for function, very fast and doesn't need to lock table
- Stable
  - Will not modify database, read-only so read-locks table
- Volatile
  - Can change database, slowest and default

Lecture 8

We have two different query types in SQL that are standard

- Recursive
  - Manage hierarchies, graphs
- Window
  - Spread group-by summaries

Window functions

Group by

Group-by allows us to summarise a set of results into a smaller set that have been grouped by a common attribute that is represented by a single row, e.g.

```
select student, avg(mark)
from   CourseEnrolments
group  by student;
```

Window functions allow us to

- Perform aggregate functions on values of a group e.g. max/min/avg/count
- Append the aggregate value from the function to each tuple in the group

E.g. attach student's average mark to each enrolment

```
select *, avg(mark)
over   (partition by student)
from   CourseEnrolments;
```

We use a partition to split sections of the table for our aggregate functions, suppose the table has more entries than just student id, then we need to make sure we perform our aggregate function OVER the students.

```
 student  |  avg
----------+-------
 46000936 | 64.75
 46001128 | 73.50


select *,avg(mark) over (partition by student) ...

 student  | course | mark | grade | stueval |  avg
----------+--------+------+-------+---------+-------
 46000936 |  11971 |   68 | CR    |       3 | 64.75
 46000936 |  12937 |   63 | PS    |       3 | 64.75
 46000936 |  12045 |   71 | CR    |       4 | 64.75
 46000936 |  11507 |   57 | PS    |       2 | 64.75
 46001128 |  12932 |   73 | CR    |       3 | 73.50
 46001128 |  13498 |   74 | CR    |       5 | 73.50
 46001128 |  11909 |   79 | DN    |       4 | 73.50
 46001128 |  12118 |   68 | CR    |       4 | 73.50
```

In the first one, we ignore the courses since we group everything by student, however DEPENDING on requirements we may want the second version which will append the aggregate computation to the end of our result.

Exercise

Using window functions, write an SQL function to find:

- all of the students in a given course
- whose mark is < 60% of average mark for course

```
create or replace function
  under(integer) returns setof CourseEnrolments
as $$
...
$$ language sql;
```

WITH queries

- We can often break up queries into multiple views and perform joins
- But sometimes we don't want the views to persist
- We use the with keyword to perform this
- It creates temporary views

```
with V as (select a,b,c from ... where ...),
     W as (select d,e from ... where ...)
select V.a as x, V.b as y, W.e as z
from   V join W on (v.c = W.d);
```

Now when we perform this Query, the view V and W only exist in the scope of the query, and are destroyed afterwards. V and W are CTE's (Common Table Expressions). This is the exact same as performing subqueries! With however allows us to perform recursive queries.

Recursive Queries

We can define recursive queries in the following

```
with recursive T(a1, a2, ...) as
(
    non-recursive select
  union
    recursive select involving T
)
select ... from T where ...
```

Base case is non-recursive select, and the recursive select is the inductive case e.g.

Example: generate sum of first 100 integers

```
with recursive nums(n) as (
    select 1
  union
    select n+1 from nums where n < 100
)
select sum(n) from nums;
-- which produces ...
 sum
------
 5050
```

Note sum is an aggregate, our nums view is the numbers 1 – 100 that is recursively defined.

Rules for Recursive Queries

- The subqueries generating T cannot be arbitrary
    - Non recursive select rules (base case)
        - Does not refer to T
        - Generates an initial set of tuples for T
    - Recursive select (inductive case)
        - Must be a query involving T
        - Has to include a where condition
        - Must eventually terminate

How recursive Queries work

- First, we initialise our result to the base case
- Set our current iteration to that result
- Loop through till the next result is not empty, appending the result of the next recursive case on top

```
-- res, work, tmp are all temporary tables
res = result of non-recursive query
work = res
while (work is not empty) {
   -- using work as the value for T ...
   tmp = result of recursive query
   res = res + tmp
   work = tmp
}
return res
```

Aggregates

- Aggregates reduce a collection of values into a single result
    - Count (Tuples)
    - Sum (results)
    - Max (result) etc.

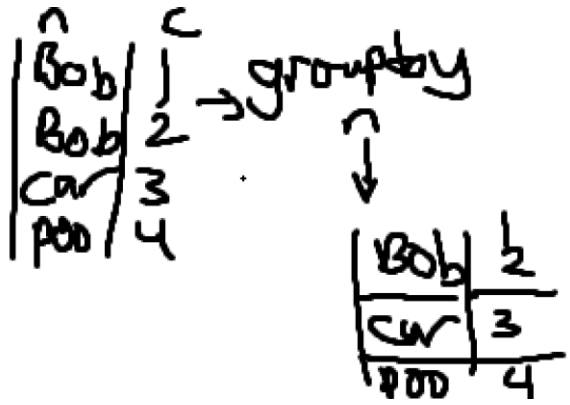
Semantics of Aggregates how they work

```
AggState = initial state
for each item V {
    # update AggState to include V
    AggState = newState(AggState, V)
}
return final(AggState)
```

First, we initialise our aggregate result to the initial result, then for each item in our grouping we will perform our aggregate function to reach new state - dynamic programming approach. At the end we return our final state.

Aggregates are always used with a group by clause or in initial selects, since they perform actions on summaries results

One way of visualising it would be this image below



We group the result into those categories and can now easily perform aggregate functions.

User defined aggregates

SQL standard does not support user-defined aggregates but PostgreSQL has a mechanism to accomplish this. We need to

1. Define a base type of input values
2. Define a state type of intermediate states
3. State mapping function: sfunc (state, value) → newState
4. Optionally we can have
    a. Initial state (defaults to null)
    b. Final state function: ffunc (state) → result

Overall syntax

```
CREATE AGGREGATE AggName(BaseType) (
    sfunc     = NewStateFunction,
    stype     = StateType,
    initcond  = InitialValue,
    finalfunc = FinalResFunction,
    sortop    = OrderingOperator
);
```

Note that the last 3 values are all optional

Creating a custom count aggregate

```
create aggregate myCount(anyelement) (
    stype    = int,     -- the accumulator type
    initcond = 0,       -- initial accumulator value
    sfunc    = oneMore -- increment function
);

create function
    oneMore(sum int, x anyelement) returns int
as $$
begin return sum + 1; end;
$$ language plpgsql;
```

Note that any element will match to all element types however only use when needed.

Aggregate function to add two columns together

```
create type IntPair as (x int, y int);

create function
    AddPair(sum int, p IntPair) returns int
as $$
begin return p.x + p.y + sum; end;
$$ language plpgsql;

create aggregate sum2(IntPair) (
    stype     = int,
    initcond  = 0,
    sfunc     = AddPair
);
```

In this example, the aggregate type is an integer, and the input type is a pair

Stringagg

To concatenate a set of strings together using aggregate functions we have

- Postgres
    - String_agg (field, 'delimiter')
- SQLite
    - Group_concat (field, 'delimiter')

Exercise find the error

```
create or replace function
      append(soFar text, newStr anyelement) returns text
as $$
begin
      if (soFar = '') then
            return newStr::text;
      else
            return soFar||','||newStr::text;
      end if;
end;
$$ language plpgsql;

drop aggregate if exists concat(anyelement);

create aggregate concat(anyelement) (
      initcond  = '',
      stype     = text,
      sfunc     = append
);
```

Note we can use function to finalise the result e.g., drop the final comma or fix up, and call it at the end of our aggregate using finalfunc = "function"

Advanced constraints

For example

- each branch asset = sum of assets in all accounts
- only one manager in each department

these are all constraints relating more than 1 table together

to get around this we have a keyword ASSERTION

- CREATE ASSERTION "name" check(condition)
    - Condition is a query
    - The condition is for integrity of database

Example: #students in any UNSW course must be < 10000

```
create assertion ClassSizeConstraint check (
   not exists (
      select c.id from Courses c, CourseEnrolments e
      where  c.id = e.course
      group  by c.id having count(e.student) > 9999
   )
);
```

Example: assets of branch = sum of its account balances

```
create assertion AssetsCheck check (
   not exists (
      select branchName from Branches b
      where  b.assets <>
             (select sum(a.balance) from Accounts a
              where a.branch = b.location)
   )
);
```

Needs to be checked after *every* change to either **Branch** or **Account**

HOWEVER, this is too slow, on each update/insert we will need to check constraints which is way too expensive so it is not standard in most SQL. Instead we use triggers

Triggers

Triggers are a lightweight mechanism for dealing with assertions, they are an event-based tool for database similar to hardware interrupts. Triggers are very similar to functions; however, they will work as interrupts on your defined events

Example uses for triggers

- Maintaining summary data
- Updating another table on an update for another
- Checking schema-level assertions on updates/insertions
- Performing multi-table updates
- An efficient way to maintain integrity

Steps for making a trigger

1. Set the event which will activate the trigger
2. When event occurs, we go to function which returns trigger

Settings for trigger

- We can set triggers to occur
    - Before/after OR instead of the triggering event
    - Access to old and new values for advanced constraint checking
    - Can limit updates to a particular set of attributes
    - Perform action: for each modified tuple, once for all modified tuples

Syntax for triggers

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [ OR Event2 ... ]
[ FOR EACH ROW ]
ON TableName
[ WHEN ( Condition ) ]
Block of Procedural/SQL Code ;
```

Note

- Trigger events are
    - INSERT
    - DELETE
    - UPDATE
- If we specify the for each row statement
    - The code is executed on each modified row
    - Otherwise the code is executed after ALL tuples are modified before changes are committed to the physical disk, we are storing our database on
- Triggers can be activated before/after their event
    - If active before event we have a new variable to work with
        - "New" contains the proposed value we are adding
        - If we modify NEW, it will cause a different value to be placed in the DB

- If active AFTER the effects of the event are visible
    - NEW contains the current value of the changed tuple
    - OLD contains the previous value of the changed tuple
    - Constraint checking has been done for NEW
- PLEASE NOTE OLD does not exist for insertion and NEW does not exist for deletion

Semantics for Triggers

```
create trigger X before insert on T Code1;
create trigger Y after insert on T Code2;
insert into T values (a,b,c,...);
```

You might be asking, what does this do?

- It will execute whatever is in Code1
- Now our variable NEW has access to (a, b, c, …)
- Code will perform checks on NEW or do whatever it wants
- We can modify NEW which will also modify the tuple being inserted
- DBMS Will then perform constraint checks for the new tuple being inserted
- If fails any of checks, we can cancel that transaction
- After we will execute whatever is in Code2
- Now our variable NEW has access to the final tuple if inserted from Code1
- Now we can do final checking/modifications to ensure constraints are satisfied

Normally we perform our final insert after constraints all satisfied on trigger 2 AFTER insert. This is to ensure safety

Note we can perform our trigger on update/insert by doing this

Create trigger X before update or insert

Trigger and function syntax for Plpgsql

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [OR Event2 ...]
ON TableName
[ WHEN ( Condition ) ]
FOR EACH {ROW|STATEMENT}
EXECUTE PROCEDURE FunctionName(args...);
```

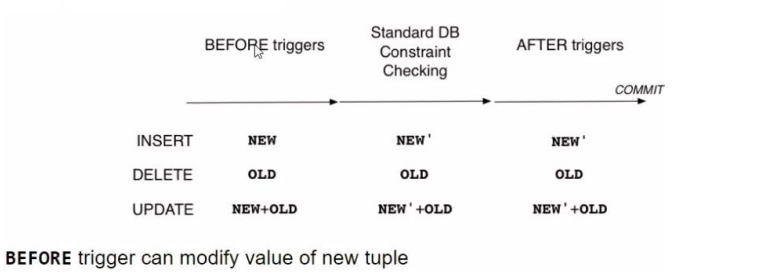
Functions can be any type of functions for triggers, some can even return a trigger to raise exceptions on conditions.

Important note

- If our trigger is Before update/insertion
    - Must return old or must return new has to return one of them
    - This is because if we return old, no change occurs
    - If we return new, we do our update
    - If any exception occurs, we will return from the function do nothing abort!

Summary of Triggers

- Allow us to perform global constraint checks efficiently
- Maintain summary values/perform actions on events
- Invoke functions on interrupts (events)
- Manipulates or uses the OLD/NEW values of changed tuples.



BEFORE trigger can modify value of new tuple

Example, a constraint to make sure person lives inside of a valid state that exists

```
create trigger checkState before insert or update
on Person for each row execute procedure checkState();

create function checkState() returns trigger as $$
begin
   -- normalise the user-supplied value
   new.state = upper(trim(new.state));
   if (new.state !~ '^[A-Z][A-Z]$') then
      raise exception 'Code must be two alpha chars';
   end if;
   -- implement referential integrity check
   select * from States where code=new.state;
   if (not found) then
      raise exception 'Invalid code %',new.state;
   end if;
   return new;
end;
$$ language plpgsql;
```

Note the IF statement could prevent a lot of unnecessary querying so we would like to type check values to avoid queries if invalid.

Note not found is also a keyword but we can also use flag-based checks -1, 1.

Sometimes we need multiple triggers for a single event, e.g. calculating number of students enrolled in course

- Need trigger for when student enrols
- Need trigger for when student swaps
- Need trigger for when student drops

Each need to be a separate trigger.

Examples

Case 1: new employees arrive

```
create trigger TotalSalary1
after insert on Employees
for each row execute procedure totalSalary1();

create function totalSalary1() returns trigger
as $$
begin
    if (new.dept is not null) then
        update Department
        set    totSal = totSal + new.salary
        where  Department.id = new.dept;
    end if;
    return new;
end;
$$ language plpgsql;
```

Case 2: employees change departments/salaries

```
create trigger TotalSalary2
after update on Employee
for each row execute procedure totalSalary2();

create function totalSalary2() returns trigger
as $$
begin
    update Department
    set    totSal = totSal + new.salary
    where  Department.id = new.dept;
    update Department
    set    totSal = totSal - old.salary
    where  Department.id = old.dept;
    return new;
end;
$$ language plpgsql;
```

Need to for this case check what type of update, we need to check if 1. Department change OR if the pay rise changed

```
create trigger TotalSalary3
after delete on Employee
for each row execute procedure totalSalary3();

create function totalSalary3() returns trigger
as $$
begin
    if (old.dept is not null) then
        update Department
        set    totSal = totSal - old.salary
        where  Department.id = old.dept;
    end if;
    return old;
end;
$$ language plpgsql;
```

One assertion, 3 triggers but same effect and more efficient

Self-exercise

Requirement: maintain assets in bank branches

- each branch has assets based on the accounts held there
- whenever an account changes, the assets of the corresponding branch should be updated to reflect this change

Some possible changes:

- a new account is opened
- the amount of money in an account changes
- an account moves from one branch to another
- an account is closed

Implement triggers to maintain **`Branch.assets`**

Lecture 9

We need to make sure for triggers we don't add too much since adding a trigger will degrade transaction speed. Only use if needed they eat performance (still better than assertions) just don't overuse them.

Please note

- Triggers after will add and modify db so if we need to check, first do constraint check with before!

Another self-exercise

Consider a simple airline flights/bookings database:

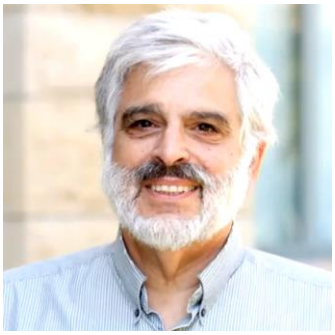
```
Airports(id, code, name, city)
Planes(id, craft, nseats)
Flights(id, fltNum, plane, source, dest
        departs, arrives, price, seatsAvail)
Passengers(id, name, address, phone)
Bookings(pax, flight, paid)
```

Write triggers to ensure that **Flights.seatsAvail** is consistent with number of **Bookings** on that flight.

Assume that we never **UPDATE** a booking (only insert/delete)

Life advice in your job if your new don't judge and flame the schema your working with, learn to work with what I'm given!

Famous DB NERDS OFFTOPIC

- Larry Ellison founder of oracle
  - 
- Peter Chen, invented ER Data Model
  - 
- Don Chamberlin inventor of SQL from IBM
  - 
- Jeff ULLman supervised founders of google and Raymond wong, researcher
  - 

- Ted Codd inventor of relational model and winner of turing award in 1981 worked in IBM
  - 
- Michael Stonebraker, inventor of postgres and Ingred
  - 
- Patricia Selinger, IBM Query optimiser
  - 

- Bruce Lindsay, IBM invented caching and transaction optimisation
  - 
- Jennifer Widom, chair of computer science at Stanford
  - 

- Rasmus Lerdorf, developed PHP **P**eople **H**ate **P**HP, created limit clause in SQL
  - 
- Bayer and McCreight, inventors of B-tree data structure very fast
  - Rudolf Bayer 
  - Eric McCreight 
- Jim Gray pioneer of transaction processing
  - 
- John Shepperd
  - 

PHP – People Hate PHP

Programming with DBs in general.

We have been working at a data level, now we need to know how to transfer and process the data. There are ways to do this in most programming languages.

- We need to devise a way to
  - Connect to the DBMS
  - Map tuples to objects
  - Map requests to queries in PL
  - Iterate through the results of the query, need to return result of query as list or iterable object

- Sometimes languages are closer to a DBMS than other languages
    - Libpq allows C programs to use PG structs in library very easily, closer to database level
    - Using Java JDBC requires conversion of tuples to objects

Generic DBMS program

```
db = connect_to_dbms(DBname,User/Password);
query = build_SQL("SqlStatementTemplate",values);
results = execute_query(db,query);
while (more_tuples_in(results))
{
   tuple = fetch_row_from(results);
   // do something with values in tuple ...
}
```

This pattern is used in a lot of different libraries:

- Java/JDBC
- PHP/PDO
- PERL/DBL
- Python/dbapi2
- TCL etc.

Example php db code.

```
$db = dbConnect("dbname=myDB");
...
$query = "select a,b,c from R where c >= %d";
$result = dbQuery($db, mkSQL($query, $min));
while ($tuple = dbNext($result)) {
   $tmp = $tuple["a"] - $tuple["b"] - $tuple["c"];
   # or ...
   list($a,$b,$c) = $tuple;
   $tmp = $a - $b - $c;
}
```

PHP DB Library

- dbConnect (conn): connection for DB
- dbQuery (db, query): sends query statement for execution
    - mksql allows us to pass arguments, but we can sort of do this anyway by passing them into the query string
- dbNext (res): gets next result from result set
- dbUpdate (db, query): sends query to update/insert/delete

```
$q = "select name,max(mark) from Enrolments ...";
$r = dbQuery($db,$q);
$t = dbNext($r);
# results in $t with value
array(0=>'John', "name"=>'John', 1=>95, "max"=>95)
```

All the code provided however is just a wrapper API for the real code shown below, to perform in raw php we would do this

```
$db_handle = pg_connect("dbname=bpsimple");
$query = "SELECT title, fname, lname FROM customer";
$result = pg_exec($db_handle, $query);
if ($result) {
  echo "The query executed successfully.\n";
  for ($row = 0; $row < pg_numrows($result); $row++) {
    $fullname = pg_result($result, $row, 'title') . " ";
    $fullname .= pg_result($result, $row, 'fname') . " ";
    $fullname .= pg_result($result, $row, 'lname');
    echo "Customer: $fullname\n";
  }
} else {
  echo "The query failed with the following error:\n";
  echo pg_errormessage($db_handle);
}
pg_close($db_handle);
```

We as programmers need to know how much DBMS vs programming. Relative costs of DB access operations:

- establishing a DBMS connection … highest
- initiating an SQL query … high
- accessing tuples … low

We need to find the balance and efficiency.

Example of inefficiency

Example: find mature-age students

```
$query = "select * from Student";
$results = dbQuery($db,$query);
while ($tuple = dbNext($results)) {
    if ($tuple["age"] >= 40) {
        --  process mature-age student
    }
}
```

If 10000 students, and only 500 of them are over 40, we transfer 9500 unnecessary tuples from DB.

Instead we can do this at db level!

We want to minimise number of queries and balance with php code.

Example of cancer

```
$query1 = "select id,name from Student";
$res1 = dbQuery($db,$query1);
while ($tuple1 = dbNext($res1)) {
    $query2 = "select course,mark from Marks".
              " where student = $tuple1['id']";
    $res2 = dbQuery($db,$query2);
    while ($tuple2 = dbNext($res2)) {
        --  process student/course/mark info
    }
}
```

Don't do a frickin query inside a double while loop, would destroy performance will do n+1 query. A better way

```
$query = "select id,name,course,mark".
         " from Student s, Marks m".
         " where s.id = m.student";
$results = dbQuery($db,$query);
while ($tuple = dbNext($results)) {
    --  process student/course/mark info
}
```

Difference between 10 minutes vs 1 minutes.

Lecture 10

If we need something more computational, or more demanding and difficult, we can use programming languages.

PHP – People Hate PHP

Newer versions of PHP are good and have strong OOD but then again, its fricking php.

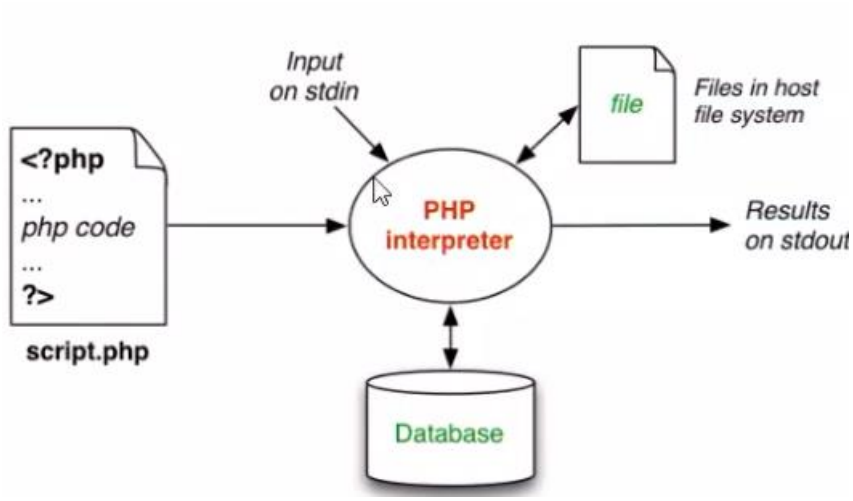
PHP Language

Code is written in these braces

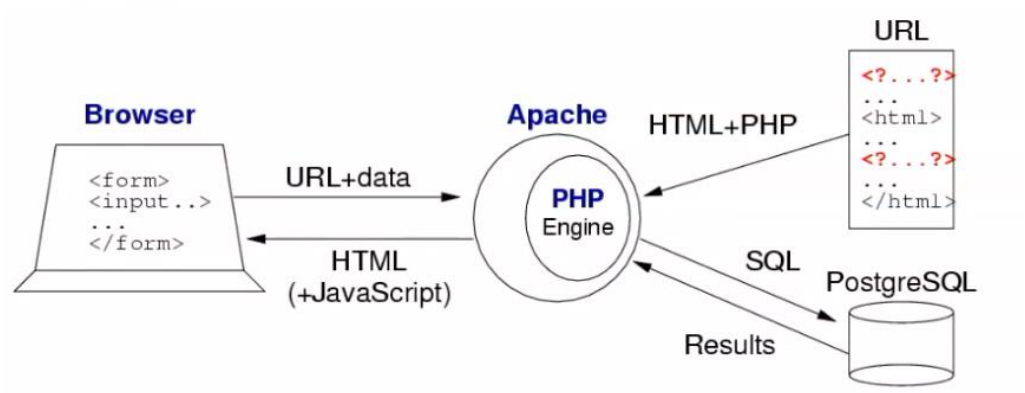
```
#!/usr/bin/php
<?
... PHP code ...
?>
```

$ARGV [] contains command-line parameters STARTING FROM 1 FRICK OFF PHP

PHP MODEL



PHP tends to run server side, such as apache webservers



PHP to HTML

- scans the script top to bottom, also includes required files, similar to how we use import/include
- any text not in <? ?> tag is shown on webpage since it will be on HTML side of code
- any php code in the <? ?> tag is executed and output is sent to webpage
- we send the Header Packet for HTTP client-server
- php header () function lets us modify the HTTP header packet, has to be called before we send the packet though

Example PHP/HTML script

```
<html>
<? require("myDefinitions.php");
    $pageName = $_POST["name"]; $max = $_POST["max"];
?>
<body bgcolor='purple'>
<h1>This is <?=$pageName?></h1>
<? if ($max <= 0) { ?>
    <b>There are no numbers to display</b>
<? }
else {
    for ($i = 0; $i <= $max; $i++)
        echo "$i<br>\n";
}
?>
</body>
</html>
```

Most PHP usage is in web application frameworks

- using MVC design pattern
    - generic abstract pattern to model the relation between models which are the backend representation, a view which is a display frontend representation and a controller which allows interaction between both through the controller itself
    - repeatable pattern that can be applied to most designs that require it.

PHP language has following characteristics:

- similar to C sprinkled with Perl (but more cancer)
- kind of duck typing? Idk why I hate it
- good for string works (but so is Perl)
- associative arrays (best part imo)
- extensive libraries of function
- supports OO design
- comments in // or #

When we execute PHP programs in a webserver

- HTTP request will give us our parameters, or if we run it on our machine it's in $argv []
- We have CGI parameters (web calls) are stored in the arrays, $_GET, $_POST, $_REQUEST

```
http://server/user/list.php?name=John&age=21
```

n the script, the parameters would be accessed as:

```
print "Name is $_REQUEST[name]\n";
print "Age is ".$_REQUEST['age']."\n";
print "Name: $_GET[name] Age: $_GET[age]\n";
```

More of this found in rest api, look for get post put delete request

Note request is a superset of get/put/post, since its how we make requests

Variables in PHP

- No declaration of variables (idk why)
- Variables are created by assigning a value to them
- All variable names are preceded by $, so increment is like $i++ or $++i
- The type of variable is what is last assigned value to variable
- We can check/set variable type with gettype/settype functions
- We can also do casting just like C (int), (String), (Custom Type)
- Default value of unassigned variable is null. (distinguished constant
- If unset variable is used, we get 0, empty string or false depending on context
    - This is cancer because if a result is 0 sometimes it considers unset for error check like movies db assignment

Examples

```
$foo = 3;           # $foo is an int, value 3

$foo = "8";         # $foo is now a string, value "8"

$foo = $foo + 2;    # $foo is now an int, value 10

$foo = "$foo green bottles";
                    # $foo is now "10 green bottles"

$foo = 3.0 * $foo; # $foo is now double, value 30.0

$foo = (int)$foo;   # $foo is now an int, value 30
```

- The lifetime of all variables is the current script
- Variables defined outside any functions have a global scope
    - CANNOT BE ACCESSED UNLESS REQUESTED IN SPECIAL WAY
    -
    ```
    function f() { global $max_num, $colour; .... }
    ```
    - using keyword global
- Super-global giga-god mode arrays, $_GET, $_PUT, $_REQUEST, $_POST, $_DELETE, $_COOKIE etc
    - Access these anywhere

Constant in PHP

Constants are defined using the define () function

- We can only evaluate to scalar base types
    - Int
    - Float
    - String
- Case-sensitive names written without dollar sign
- Are always global available super-giga-god mode global
- Cannot be redefined or undefined once set, will be destroyed after script

Types

- Four scalar types
  - Boolean true/false (case-insensitive)
    - False can also be 0 or “”
    - All non-zero values are seen as true (can lead to issues though)
  - Integer 32-bit int format
  - Float IEEE floating point
  - String

Strings

- Sequence of characters like Perl
- Double quoted strings allow us to plug in variables in directly, e.g.
  - Print “poopoo $variable\n”;
- Single quotes will not allow us to access variables (escapes dollar sign)
- We can also use “heredoc” strings in the following (<<<identifier)

```
print <<<XYZ
This is a "here" document. It can contain
many lines of text, with interpolation.
Such as the value of x is $x
With any old "quotes' the we ``like''
XYZ;

$str = <<<aLongString
This is my "long" string.
Ok, it's not really so long
aLongString;
```

  - USEFUL FOR SHELL/BASH SCRIPTS TOO
- When we use variables, even if they are different type, when we put them in our string it will auto type cast it for us

```
$a = 1;  $b = 3.5;  $c = "Hello";
$str = "a:$a,  b:$b,  c:$c";
// now $str == "a:1, b:3.5, c:Hello"
```

- Double quotes give us access to escape sequences and variables
- Single quotes no escape no variables
- Heredocs will not need escape, variables + escapes work though

We can pass in unescaped ‘ into our “ string and it will not be auto-escaped. This is extremely useful in HTML since HTML tag values are always single quoted, so it allows direct and easy conversion

Note that interpolation does occur in **"This is '$it'"**

I.e. **<? $it = 5; print "This is '$it'"; ?>** displays **This is '5'**

```
print "<input type='text' name='qty' value='$_GET[qty]'>\n";
```

We can also concatenate strings with the ‘.’ Character. Similar to + in java.

Trim removes whitespace from left and right end of string, can be done with regexp too

More operations for strings

More operations on strings:

preg_split() partitions string into array via Perl regexp

```
// $s == "  ab  cde fg"
$a = preg_split('/\s+/',$s);
// $a[0]=="" && $a[1]=="ab"
// && $a[2]=="cde" $a[3]=="fg"
```

join() assembles strings from an array

```
// $a[0]=="" && $a[1]=="ab"
// && $a[2]=="cde" $a[3]=="fg"
$s = join(":",$a);
// $s == ":ab:cde:fg";
```

Plus many others ... see PHP Manual for details.

Arrays best part of PHP imo

- Arrays are sequence of values accessible via index
- Index can be values of any types since they are hashed
    - Scaler + associative arrays, similar to hash table
    -

```
$a[0] = "abc";  $a[1] = 'def';  $a[2] = ghi;

$b['abc'] = 0;  $b[def] = 1;    $b["ghi"] = 2;
```

There are different ways to initialise array with elements

Arrays can be initialised element-at-a-time:

```
$word[0]="a";  $word[1]="the";  $word[2]="this";

$mark["ann"]=100;  $mark["bob"]=50;  $mark["col"]=9;

$vec[]=1;  $vec[]=3;  $vec[]=5;  $vec[]=7;  $vec[] = 9;
// which is equivalent to
$vec[0]=1;  $vec[1]=3;  $vec[2]=5;  $vec[3]=7; $vec[4] = 9;
```

Arrays can be initialised in a single statement:

```
$word = array("a", "the", "this");

$marks = array("ann"=>100, "bob"=>50, "col"=>9);

$vec = array(0 => 1, 1 => 3, 2 => 5, 3 => 7, 4 => 9);
// which is equivalent to
$vec = array(1, 3, 5, 7, 9);
```

Note both approaches are the exact same

Extraction

We can perform multiple value extraction using list keyword

```
$a = array(5, 4, 3, 2, 1);
list($x,$y,$z) = $a;
# $x==5, $y==4, $z==3
```

This will match the first n variables to the ones in the array.

Multi-dimensional array

With great power comes great responsibility

Multi-dimensional arrays work ok (array elements can be any type)

```
$fruits = array ( "fruits"  => array ( "a" => "orange"
                                     , "b" => "banana"
                                     , "c" => "apple"
                                     )
                , "numbers" => array ( 1,2,3,4,5,6 )
                , "holes"   => array (       "first"
                                     , 5 => "second"
                                     ,       "third"
                                     )
                );
```

Array iteration approaches

```
for ($i = 0; $i < count($word); $i++)
   print "word[$i] = $word[$i]\n";

foreach ($words as $w) print "next word = $w\n";

for (reset($marks); $name = key($marks); next($marks))
   print "Mark for $name = $marks[$name]\n";

reset($marks);
while (list($name,$val) = each($marks))
   print "Mark for $name = $val\n";

$elem = current($vec);
while ($elem) {
   print "Next elem is $elem\n";
   $elem = next($vec);
}
```

I like approach 1 and 2 others r annoying and hard to remember

Important for the foreach approach

```
$marks = array("Ann"=>95, "John"=>75, "David"=>60);

foreach ($marks as $name => $mark)
        echo "$name scored $mark%\n";

echo "Whole array: $marks\n";
```

In this we map the values of our array marks by mapping the name to marks!

Note the final echo will just show “Whole array: Array” since $marks is just simply pointer to array structure, doesn’t do the nice java/php print arrays

PHP class Types

Php has standard idea of class e.g.

- $x = new class;
- $x -> method (1, ‘a’);

NULL (case insensitive) indicates variable exists but no variable.

Variable checks

- isset ($x): isset will check if $x has non-null value
- Is_null($v): will check if $v has value null
- Empty($v): will check if $v has value 0, “” or array ()
- Unset($v): will remove the variable $v delete (memory saving)

We can dynamically create variables, powerful but this can also be a bit hard.

```
for ($i = 0; $i < $MAX; $i++) {
   $varname = "myVar$i";
   $value   = ${$varname};
   print "Value of $varname = $value\n";
}
```

In this simple program we can effectively create new variables with generic naming which can be extremely useful, it allows us to iterate over our variables we create

Functions in PHP

- Functions have no type they can return anything, and arguments don't need type they can use anything (I don't like this isn't as strict too abstract)

```
function FuncName($arg1, $arg2, ... )
{
   Statement; ...
   return Expression;
}
```

```
// return array of first n integers
function iota($n)
{
   for ($i = 1; $i <= $n; $i++)
      $list[] = $i;
   return $list;
}
```

- can handle variable-length argument lists (like C's printf)
  - using special functions **func_num_args()**, **func_get_arg()**, and **func_get_args()**

Note we can also use default values for functions similar to coalesce

```
function makeCoffee($type="latte", $size="big") {
    return "Making a $size cup of $type.\n";
}
echo makeCoffee();
echo makeCoffee("cappucino");
echo makeCoffee("espresso","tiny");
```

Debugging

- Print_r will represent value in the print_r
- Var_dump displays more info on value in var_dump
- Error_reporting (Level) controls how much of error to shos
- @func () executes function and supressed error reporting

Finally, PHP PostgreSQL

- **pg_connect()** ... connect to the database
- **pg_query()** ... send SQL statement for processing
- **pg_fetch_array()** ... retrieve the next result tuple
- **pg_num_rows()** ... count # rows in result
- **pg_affected_rows()** ... count # rows changed

Lecture 11

To connect to a DB in php we can use the following

- ```
  $db = pg_connect("dbname=mydb");
  ```
  - If there is no password auth needed

```
$cp = "dbname=hisdb user=fred password=abc";
```

- ```
  $db = pg_connect($cp);
  ```
  - If there is authentication needed

To create a query in php we need to supply the string form of the query, and the database so in the form

- ```
  resource pg_query(resource db, string Stmt)
  ```
- We get returned a resource
  - Set of results from query
  - Or nothing if we did insert/delete/update
- If any problems with query we get returned 0 (illegal cursor)
  - Errors include invalid db/syntax in query
- If we want to find error with query, we can do this
  - ```
    if (!$result)
        print pg_last_error();
    ```

To get the number of rows from query

We use the following

- ```
  int pg_num_rows(resource Result)
  ```

This is similar to using Count in sql

To return the number of modified rows from update/delete/insert

we use the following

- ```
  int pg_affected_rows(resource Result)
  ```

To access a certain row from the result of our query

We use the following

- `array pg_fetch_row(resource Res, int which)`
- If no argument is supplied, it will fetch the next tuple
- Returns an array value that can be treated as a result row
- Fields are accessed by array index from query select
- If no elements, returns 0

Usually this is used in while loop, or for loop

```
$query = "select id,name from Staff";
if ($result = pg_query($db, $query)) {
   $n = pg_num_rows($result);
   for ($i = 0; $i < $n; $i++) {
      $item = pg_fetch_row($result,$i);
      print "Name=$item[1], StaffID=$item[0]\n";
   }
}
```

OR

We can use

pg_fetch_array(resource *Res*, int *which*)

Which is the exact same but indexes by field and number as well! This is better for larger data

```
$query = "select id,name from Staff";
if (!($result = pg_query($db, $query)))
   print "Error: ".pg_last_error();
else {
   $n = pg_num_rows($result);
   for ($i = 0; $i < $n; $i++) {
      $item = pg_fetch_array($result,$i);
      $nm = $item["name"]; $id = $item["id"];
      print "Name=$nm, StaffID=$id\n";
   }
}
```

COMP3311 library

- accessDB(dbname): establish connection to DB
- dbQuery(db,sql): send SQL statement for execution
- dbNext(res): fetch next tuple from result set
- dbOneTuple(db,sql): run SQL to get a single tuple
- dbOneValue(db,sql): run SQL to get a single value
- dbUpdate(db,sql): send SQL insert/delete/update
- mkSQL(fmt,v1,v2,...): build an SQL statement string

Theory section

Relational Design Theory

- We want to make good schemas, but what makes a good schema?
- We want a quantifiable way to tell if a schema is better than another
- We also want a method to transform schemas to make them better

Functional Dependencies

- Constraints between attributes within a relation

A good relational DB design must

- Capture all requirements with minimal amount of stored data
- No redundancy ← cancer big bad

Sometimes redundancy can be good for performance to avoid some joining, but in general it is very bad, we can also do this with a non-redundant design.

Example of redundancy

accountNo	balance	customer	branch	address	assets
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	600	9876543	Round Hill	Horseneck	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000

- By storing everything in one table here we get a lot of duplicate data
- We get repeating information in the same column.

Insertion Anomaly

- When we insert a new record, we need to check that the data is consistent with the existing rows

Insertion anomaly example (insert account A-306 at Round Hill):

accountNo	balance	customer	branch	address	assets
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	600	9876543	Round Hill	Horseneck	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000
A-306	800	1111111	Round Hill	Horseneck	8000800

-
- Inconsistency will create a lot of problems which can be impossible to debug, hair pullers

Update anomaly

- In the above example if we update 1 branch, we have to update ALL rows which have that branch for consistency

Jpdate anomaly example (update Round Hill branch address):

accountNo	balance	customer	branch	address	assets
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	600	9876543	Round Hill	Palo Alto	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000

-
- Updated only one address not all

Deletion anomaly

- If we remove information about the last account at a branch, all of the branch information is gone, since it is tightly coupled to the account in this case
- If we remove an account which has a branch that ONLY has that account, we would effectively be deleting the branch, since the branch technically has no accounts. This is very bad behaviour!

Deletion anomaly example (remove account A-101):

accountNo	balance	customer	branch	address	assets
A-101	500	1313131	Downtown	Brooklyn	9000000
A-102	400	1313131	Perryridge	Horseneck	1700000
A-113	600	9876543	Round Hill	Horseneck	8000000
A-201	900	9876543	Brighton	Brooklyn	7100000
A-215	700	1111111	Mianus	Horseneck	400000
A-222	700	1111111	Redwood	Palo Alto	2100000
A-305	350	1234567	Round Hill	Horseneck	8000000

- Where is the Downtown branch located? What are its assets?
- By deleting that row, we lose all information on Downtown since this is the only tuple that references downtown, even though downtown is a valid branch

We can use triggers/constraints to avoid insertion/update anomaly, however deletion anomaly is dependant on schema design fully.

We can set up new tables like this

- Branch table which has address + asset + name
- Account table which has the balance and branch and account number
- Finally, we have a customer who has an account number + customer number

But this is a very small example, imagine if there were 50 attributes

Design

We want to avoid the problems altogether by

- Having a schema with minimal overlap between tables by breaking them down
- Each table containing a coherent collection of data

To do this we can

- Start with a giga god table U which is the universal relation that has all the attributes
- Decompose the relation U into smaller relations Ri which are subsets of U
- We make sure all our Ri's have minimal overlap with other subsets
- We make sure that there is only just enough links to reconstruct the original table
- Typically, each smaller table contains information about one entity

ER design procedure tends to produce schemas without redundancy, however it doesn't assure us minimal redundancy

- Dependency theory will let us check for breadcrumbs
- The new procedure gives us an automated design that
    - Determines all of the attributes in the domain of our problem
    - Collects them into one giga-wiga table
    - Provides how the information is related
    - Then lets us use **normalisation** to decompose into non-redundant relations

Notations

Relation schemas	upper-case letters, denoting set of all attributes (e.g. *R*, *S*, *P*, *Q*)
Relation instances	lower-case letter corresponding to schema (e.g. *r(R)*, *s(S)*, *p(P)*, *q(Q)*)
Tuples	lower-case letters (e.g. *t*, *t'*, t_1, *u*, *v*)
Attributes	upper-case letters from start of alphabet (e.g. *A*, *B*, *C*, *D*)
Sets of attributes	simple concatenation of attribute names (e.g. *X=ABCD*, *Y=EFG*)
Attributes in tuples	tuple[attrSet] (e.g. *t[ABCD]*, *t[X]*)

Functional Dependency

- A relation instance r will satisfy dependency x → y (x depends on y)
- If for any two tuples in R agree in their set of values for attribute X, then they must also have same value for attributes Y
    - In our example for the account/branch, if two rows had the same branch, that means they had the same assets and same address
- $\text{for any } t, u \in r, \quad t[X] = u[X] \quad \Rightarrow \quad t[Y] = u[Y]$
- this can be said as "Y is functionally dependant on X"

Example

Consider the following instance *r(R)* of the relation schema *R(ABCDE)*:

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_1	c_2	d_2	e_1
a_3	b_2	c_1	d_1	e_1
a_4	b_2	c_2	d_2	e_1
a_5	b_3	c_3	d_1	e_1

What kind of dependencies can we observe among the attributes in r(R)?

From observation we can see that all values of A are unique, but all values of b differ,

- A → B
- A → C
- A → D
- A → E

Since A is unique, we can use it to identify any tuple as a primary key

- A -> BCDE

Since E values are all same, we have E is dependant on all rows, given any value of ABCD, we can find E

- A → E
- B → E
- C → E
- D → E

Since ABCD all can give us E, we cannot however say ABCD → E since we cannot use E to get a value ABCD

Remember the definition of →

We also have BC have unique combination, and CD have unique combinations

- BC → ADE
- BD → ACE
- C → D, but D does not determine C, since there is d1 for c1 and c3 not one to one

However, doing all this manually by inspection is tedious and has room for errors, in practice we want to choose a minimal set of functional dependencies from which ALL other functional dependencies can be derived.

Example

Real estate agents conduct visits to rental properties

- need to record which property, who went, when, results
- each property is assigned a unique code (P#, e.g. PG4)
- each staff member has a staff number (S#, e.g. SG43)
- staff members use company cars to conduct visits
- a visit occurs at a specific time on a given day
- notes are made on the state of the property after each visit

The company stores all of the associated data in a spreadsheet.

Describe any functional dependencies that exist in this data:

P#	When	Address	Notes	S#	Name	CarReg
PG4	03/06 15:15	55 High St	Bathroom leak	SG44	Rob	ABK754
PG1	04/06 11:10	47 High St	All ok	SG44	Rob	ABK754
PG4	03/07 12:30	55 High St	All ok	SG43	Dave	ATS123
PG1	05/07 15:00	47 High St	Broken window	SG44	Rob	ABK754
PG1	05/07 15:00	47 High St	Leaking tap	SG44	Rob	ABK754
PG2	13/07 12:00	12 High St	All ok	SG42	Peter	ATS123
PG1	10/08 09:00	47 High St	Window fixed	SG42	Peter	ATS123
PG3	11/08 14:00	99 High St	All ok	SG41	John	AAA001
PG4	13/08 10:00	55 High St	All ok	SG44	Rob	ABK754
PG3	05/09 11:15	99 High St	Bathroom leak	SG42	Peter	ATS123

State assumptions used in determining the *fd*s.

As we can see, S# will determine name + carREG

S# → name, CarReg

P# → Address

P#, When → s#

Now we want a way to derive functional dependancies

Exercise: Functional Dependencies (2)

What functional dependencies exist in the following table:

A	B	C	D
1	a	6	x
2	b	7	y
3	c	7	z
4	d	6	x
5	a	6	y
6	b	7	z
7	c	7	x
8	d	6	y

How is this case different to the previous one?

A → BCD
B does not determine CD,
B → C
This is our minimal set of functional dependancies. We can also see BD → C

The trick is to see uniqueness for functional dependency!

- Check a value, see if for any repeat value we get a different result
  - Example, check 1, see if for any other 1 it is the same, if this is the case, we have functional dependancies
- we can also extend this for n number of pairs if need be

Generalising functional dependancies

- if we know Y is a subset of X, then we know X → Y, but this is trivial

we have Armstrong's complete general rules of inference for functional dependancies

F1. Reflexivity e.g. $X \rightarrow X$

- a formal statement of *trivial dependencies*; useful for derivations

F2. Augmentation e.g. $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$

- if a dependency holds, then we can freely expand its left hand side

F3. Transitivity e.g. $X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z$

- the "most powerful" inference rule; useful in multi-step derivations

Derived rules

F4. Additivity e.g. $X \rightarrow Y, X \rightarrow Z \Rightarrow X \rightarrow YZ$

- useful for constructing new right hand sides of *fd*s (also called union)

F5. Projectivity e.g. $X \rightarrow YZ \Rightarrow X \rightarrow Y, X \rightarrow Z$

- useful for reducing right hand sides of *fd*s (also called decomposition)

F6. Pseudotransitivity e.g. $X \rightarrow Y, YZ \rightarrow W \Rightarrow XZ \rightarrow W$

- shorthand for a common transitivity derivation

Example uses of rules

Example: determining validity of $AB \rightarrow GH$, given:

$$R = ABCDEFGHIJ$$

$$F = \{ AB \rightarrow E,\ AG \rightarrow J,\ BE \rightarrow I,\ E \rightarrow G,\ GI \rightarrow H \}$$

1.	$AB \rightarrow E$	(given)
2.	$AB \rightarrow AB$	(using F1)
3.	$AB \rightarrow B$	(using F5 on 2)
4.	$AB \rightarrow BE$	(using F4 on 1,3)
5.	$BE \rightarrow I$	(given)
6.	$AB \rightarrow I$	(using F3 on 4,5)
7.	$E \rightarrow G$	(given)
8.	$AB \rightarrow G$	(using F3 on 1,7)
9.	$AB \rightarrow GI$	(using F4 on 6,8)
10.	$GI \rightarrow H$	(given)
11.	$AB \rightarrow H$	(using F3 on 6,8)
12.	$AB \rightarrow GH$	(using F4 on 8,11)

Closures

Given a finite set of functional dependancies F, the largest collection that can be derived is called the closure known as F+.

Closures will allow us to answer

- is a particular dependency X → Y derivable from F
- are two sets of dependencies F and G equivalent

Closures are generally very large though, an example of 3 elements

$R = ABC, \quad F = \{ AB \rightarrow C, \quad C \rightarrow B \}$

$F^+ = \{ A \rightarrow A, \quad AB \rightarrow A, \quad AC \rightarrow A, \quad AB \rightarrow B, \quad BC \rightarrow B, \quad ABC \rightarrow B,$
$C \rightarrow C, \quad AC \rightarrow C, \quad BC \rightarrow C, \quad ABC \rightarrow C, \quad AB \rightarrow AB, \quad \ldots\ldots,$
$AB \rightarrow ABC, \quad AB \rightarrow ABC, \quad C \rightarrow B, \quad C \rightarrow BC, \quad AC \rightarrow B, \quad AC \rightarrow AB \}$

We can reduce this down however

- Given a set X of attributes
- Given a set F of functional dependancies
- The largest set of attributes that can be derived from X using F is the closure of X noted as X+
- |X+| is bounded by the number of attributes.

This will massively simplify our problems, because now we can check if an attribute is in |x+| if we need to check if X → Y is derivable from F

For the question "is $X \rightarrow Y$ derivable from F?" ...

- compute the closure X^+, check whether $Y \subset X^+$

For the question "are F and G equivalent?" ...

- for each dependency in G, check whether derivable from F
- for each dependency in F, check whether derivable from G
- if true for all, then $F \Rightarrow G$ and $G \Rightarrow F$ which implies $F^+ = G^+$

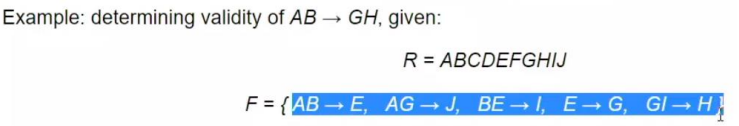
For the question "what are the keys of R implied by F?" ...

- find subsets $K \subset R$ such that $K^+ = R$

Lecture 12

Recap of Lecture 11

Shortcut to check if a rule is in the finite set of minimal FDs



We can find |AB+|, and check if GH is contained in the minimal set of AB,

|AB+| = ABEGIHJ

Since GH is in the set, we can say it's true!

Exercise

Determine possible primary keys for each of the following:

1. $FD = \{A \rightarrow B, C \rightarrow D, E \rightarrow FG\}$
2. $FD = \{A \rightarrow B, B \rightarrow C, C \rightarrow D\}$
3. $FD = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$
4. $FD = \{ABH \rightarrow C, A \rightarrow D, C \rightarrow E, F \rightarrow A, E \rightarrow F, BGH \rightarrow E\}$
5. $FD = \{AB \rightarrow C, C \rightarrow B\}$
6. $FD = \{MN \rightarrow P, NP \rightarrow Q, Q \rightarrow R\}$
7. $FD = \{ABH \rightarrow C, A \rightarrow DE, BGH \rightarrow F, FD \rightarrow ADH, BH \rightarrow GE\}$

1. ACE, since ACE → ABCDEFG (from addition), |ACE+| = ABCDEFG
2. A, since A → ABCD
3. A or B or C

Normalization

- Allows us to transform schemas
- Characterise how much redundancy is in the schema, and try to remove it
- Uses functional dependencies!

There are 6 normal forms for normalization

- First, second, third normal form (codd)
- BCNF 3.5
- Fourth normal form
- Fifth normal form

NF hierarchy: 5NF => 4NF => BCNF => 3NF => 2NF => 1NF

1NF will allow for more redundancy, think of this as subset of

1NF	all attributes have atomic values we assume this as part of relational model, so *every* relation schema is in 1NF
2NF	all non-key attributes fully depend on key (i.e. no partial dependencies) avoids much redundancy

1NF – put everything in one table

2NF – full dependancies, no partial dependencies

BCNF and 3NF – no attributes dependant on non-key attributes, no transitive dependancies avoids most redundancy.

The Process

- Decide which normal form rNF is "acceptable"
- If a relation is not in rNF
    - Partition into sub relations where each is closer to rNF
- Repeat until all relations in schema are in rNF

In industry, BCNF and 3NF are the most important and most used in industry

Boyce-Codd Normal Form (BCNF)

- Eliminates all redundancy due to functional dependancies
- May not preserve original functional dependancies

3NF

- Eliminates most redundancy due to functional dependancies
- Guaranteed to preserve all functional dependancies

Decomposition

The technique to remove redundancy is simple

1. Decompose our relation R into smaller relations S and T
    a. Select overlapping subsets of attributes
    b. Forming new relations based on attribute subsets

Properties: $R = S \cup T, \quad S \cap T \neq \emptyset$ and $r(R) = s(S) \bowtie t(T)$

So, from this we can see

- We can stitch back together our decompositions to our original relations
- The intersection of our decompositions cannot be the empty set! There needs to be a link (foreign key)
- We can get something from r by doing a join on S and T

This process is similar to merge/quicksort where we perform multiple breakdowns till our base cases. But we don't join at end obviously. This can be done algorithmically and there are Normalization algorithms that tell us how to choose S and T

Example for bank loans

branchName	branchCity	assets	custName	loanNo	amount
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-15	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
North Town	Rye	3700000	Hayes	L-16	1300

1NF

If we just do breakdown like this

- Branch (branchname, branchCity, assets)
- Loans (customer, loanno, amount)

That wont work because we lose information on which branch is loan at, no foreign key to branch, even if customer number is foreign key for branch still doesn't work properly

This is clearly not a successful decomposition.

The fact that we ended up with extra tuples was symptomatic of losing some critical "connection" information during the decomposition.

Such a decomposition is called a lossy decomposition.

In a good decomposition, we should be able to reconstruct the original relation exactly:

if R is decomposed into S and T, then $Join(S,T) = R$

Such a decomposition is called lossless join decomposition.

BCNF

A relational schema R is in BCNF with respect to a set of functional dependancies IF

- For all functional dependancies X → Y in F+
  - Either X → Y is trivial (so Y is a subset of X)
  - Or X is a super key
- A DB schema is in BCNF if all relational schemas are in BCNF

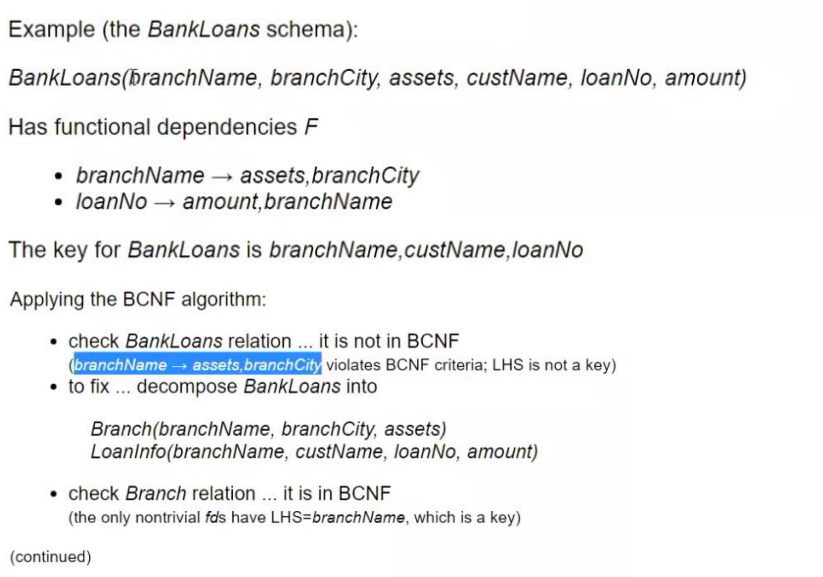
Any two-attribute relation is in BCNF

Any relation with key K, other attributes X, and K → X is in BCNF

If we transform a schema into BCNF we will guarantee

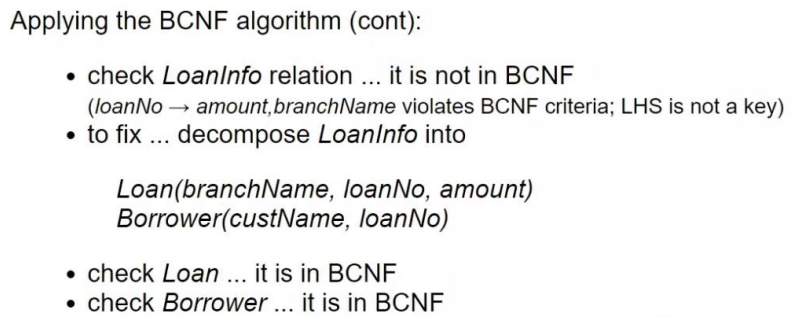
- No update anomalies due to functional dependency-based redundancy
- Lossless join decomposition
- Not guaranteed the new schema will preserve all functional dependancies from the original schema

This may be a problem if the functional dependancies contain significant information about the problem, and this would be a case where 3NF would be needed



The highlighted part basically says that since branchname is not the super key for bank Loans, and it is not trivial, we need to break it down further, branch name is the key for branch!

However now only branch is in BCNF loan is not in BCNF because loan number is not the key for loan table, so we decompose loan again



BCNF ALGORITHM

The following algorithm converts an arbitrary schema to BCNF:

```
Inputs: schema R, set F of fds
Output: set Res of BCNF schemas

Res = {R};
while (any schema S ∈ Res is not in BCNF) {
    choose any fd X → Y on S that violates BCNF
    Res = (Res-S) ∪ (S-Y) ∪ XY
}
```

Start with the original table, then any schema in our set not in BCNF we will then do decomposition.

Decomposition will choose the functional dependency that will violate BCNF (not super key/non-trivial), we will then perform decomposition and add it to our final set of tables. The final line says, remove the original table violating BCNF, then we split it up into two smaller tables that are linked together with a foreign key.

Exercise

Consider the schema R and set of fds F

$R = ABCDEFGH$

$F = \{ ABH \rightarrow C, A \rightarrow DE, BGH \rightarrow F, F \rightarrow ADH, BH \rightarrow GE \}$

Produce a BCNF decomposition of R.

Full solution: https://www.cse.unsw.edu.au/~cs3311/19s1/lectures/06/exercises/bcnf1

Candidate key for schema is BH since BH can derive everything! (ABH works but A can be derived from BH)

As we can see however A → DE doesn't have the key on the LHS so it violates BCNF

We split our big table into two smaller tables

ADE and ABCFGH where A is the link between the two tables and a is the super key

Check ABCFGH in BCNF with our remaining requirements BGH → F, F → ADH, and BH → GE

Check BGH → F, since BGH contains key check next requirement

Check F → ADH, need to decompose since key isn't on LHS and it is not non-trivial (i.e. ADH subset F)

Next decomposition

So now ABCFGH can now be decomposed to

BCFG and AFH

AFH is in BCNF since A is super key for the table AFH

Now we check BCFG and our remaining functional dependancies however there are none left!

So finally, we get the tables

BCFG, ADE, AFH all in BCNF (0 redundancy very nice very nice algorithm!), we lose some functional dependency but 0 redundancy and anyways we can link data back together anyways R03 is worse and easier imo

Exercise

P#	When	Address	Notes	S#	Name	CarReg
PG4	03/06 15:15	55 High St	Bathroom leak	SG44	Rob	ABK754
PG1	04/06 11:10	47 High St	All ok	SG44	Rob	ABK754
PG4	03/07 12:30	55 High St	All ok	SG43	Dave	ATS123
PG1	05/07 15:00	47 High St	Broken window	SG44	Rob	ABK754
PG1	05/07 15:00	47 High St	Leaking tap	SG44	Rob	ABK754
PG2	13/07 12:00	12 High St	All ok	SG42	Peter	ATS123
...						

Solution: https://www.cse.unsw.edu.au/~cs3311/19s1/lectures/06/exercises/bcnf2

This table PWANSMC has the following FDs (P → A, S → MC, PW → S)

Find key for whole thing minimal candidate key, PWN since PWN can give all rows

Check PAWNSMC is in BCNF with key PWN and FDs (P → A, S → MC, PW → S)

Check P → A, need to decompose because the full key is not in LHS

Table PA and PWNSMC and FDs (S → MC, PW → S)

Check S → MC problem since full key not in LHS

Table PA and SMC and SPWN and Fds (PW → S)

Check SPWN is in BCNF with FDs (PW → S)

Check PW → S, problem since full key not in LHS

Table PA SMC PWN PWS

Finally PWN is the candidate key so it itself is in BCNF and we finish

Final PA, SMC, PWS, PWN

3NF

A lot easier, we keep all our functional dependancies, so we make tables out of our functional dependancies. Will keep some redundancy so I hate this

A relation schema R is in 3NF w.r.t. a set F of functional dependencies iff:

for all *fd*s $X \rightarrow Y$ in F^+

- either $X \rightarrow Y$ is trivial (i.e. $Y \subset X$)
- or X is a superkey
- or Y is a single attribute from a key

Weakens BCNF requirements.

Just make a table from ALL FD's lmao,

3NF will not guarantee to avoid update anomaly

Rules for 3NF

Critical step is producing minimal cover F_c for F

A set F of fds is minimal if

- every fd $X \rightarrow Y$ is *simple*
  (Y is a single attribute)
- every fd $X \rightarrow Y$ is *left-reduced*
  (no $Z \subset X$ such that $Z \rightarrow A$ could replace $X \rightarrow A$ in F and preserve F^+)
- every fd $X \rightarrow Y$ is *necessary*
  (no $X \rightarrow Y$ can be removed without changing F^+)

Algorithm: right-reduce, left-reduce, eliminate redundant fds

Exercise

Consider the schema R and set of fds F

$R = ABCDEFGH$

$F = F_C = \{ ABH \rightarrow C, A \rightarrow D, C \rightarrow E, F \rightarrow A, E \rightarrow F, BGH \rightarrow E \}$

Produce a 3NF decomposition of R.

so, our minimal key for this case is BGH or BF

to do 3NF decomposition simple

tables in 3NF are

- ABHC
- AD
- CE
- FA
- EF
- BGHE

Next, we will check at least one table includes full key, BGH finished! And we preserved our functional dependency and reduced redundancy, however if we chose BF, we need one last table BF to capture the key! However, this will add redundancy slightly, it still reduces but keeps all FD

So, if key is not captured, we simply just add it lol this is too easy

The real estate example above is simply like this in 3NF

```
R = PWANSMC

Fc = {P->A, S->M, S->C, PW->S}

key = NPW

decomposition:

        PA
        SM
        SC
        PWS
```

And we add NPW table to link them all together since it is not fully in one of the tables

Important note

BCNF → removes all redundancy, loses some of Functional dependency

3NF → keeps all functional dependency, keeps some redundancy too

To have a good database design we need to make sure we

- Identify attributes, entities and relations from ER design (requirements)
- Map ER to relational schema
- Identify our constraints, so keys and functional dependancies
- Apply BCNF or 3NF algorithms to produced normalised schema

Note that we may need to denormalise shit design

Lecture 13

We want to make our Query processing more efficient and use transactions to make our applications safer. We need to break it down to a core level to understand our query speeds. We will know why we don't want to make more functions, modify tables, create tables etc.

Relational Algebra

- Understanding what operations on the data is required
- Understand how these operations might be implemented in the DBMS
- How much each implementation will cost?

Most expensive operation is moving data from disk to memory, it is more I/O intensive than size intensive, retrieving 10000000 rows is sometimes cheaper than retrieving a lot less rows and doing heavy i/o operations

Querying

- Join is generally faster than sub querying
- If subquery is correlated (using where) is very bad
    - Slowest of the slowest of the slowest
- Avoid producing large views or selects before filtering, always filter early as much as possible
- Avoid applying functions in where/group by
    - This will ruin database optimisation because it can't really run optimiser on your function

Creating indexes on tables O (1) lookups but can have negative side-effects and must be maintained

- Index will speed up filtering based on indexed attributes
- Reduces transaction and update time
- Indexes effective for equality, greater than or less than
- Indexes have update-time and storage overheads
- **Only useful if filtering much more frequently than you is updating**

Whenever we write a query, the DBMS will transform our query to make it as efficient as possible, and this is done by the query optimiser.

The query optimiser will

- Assess possible query execution approaches
- Evaluates likely cost of each approach and picks the cheapest

This is something we have no control over, but we can block certain bad options and limit the query optimisers chance to improve if we make bad queries.

SQLite has a very primitive optimiser, whereas postgres has an extensive optimiser. Writing Query optimiser is very hard, we can only guess and we have to be very fast otherwise we defeat the purpose of optimisation.

Example

Example: query to find sales people earning more than $50K

```
select name from Employee
where  salary > 50000 and
       empid in (select empid from Worksin
                 where  dept = 'Sales')
```

A query optimiser might use the strategy

```
SalesEmps = (select empid from WorksIn where dept='Sales');
foreach e in Employee {
    if (e.empid in SalesEmps && e.salary > 50000)
        add e to result set
}
```

Needs to examine *all* employees, even if not in Sales

Here the query optimiser will process the subquery first. But it is still not optimal yet.

A different expression of the same query:

```
select name
from   Employee join WorksIn using (empid)
where  Employee.salary > 5000 and
       WorksIn.dept = 'Sales'
```

Query optimiser might use the strategy

```
SalesEmps = (select * from WorksIn where dept='Sales')
foreach e in (Employee join SalesEmps) {
    if (e.salary > 50000)
        add e to result set
}
```

Only examines Sales employees, and uses a simpler test

This is much better since it will perform our join before we loop to reduce how many times we need to loop.

The best example of optimisation with shit query

A very poor expression of the query (correlated subquery):

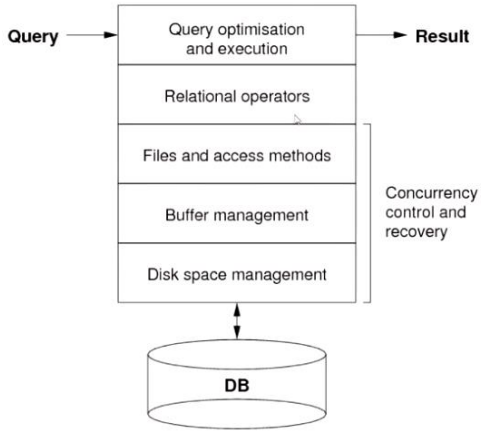
```
select name from Employee e
where  salary > 50000 and
       'Sales' in (select dept from Worksin where empid=e.id)
```

A query optimiser would be forced to use the strategy:

```
foreach e in Employee {
    Depts = (select dept from WorksIn where empid=e.empid)
    if ('Sales' in Depts && e.salary > 50000)
        add e to result set
}
```

Needs to run a query for *every* employee ...

DBMS architecture



Relational Algebra

- Mathematical system for manipulating relations
- Data manipulation language for relational model

It consists of

- Operands
    - Relations or variables representing relations
- Operators
    - Map relations to relations
- Rules for combining operands/operators into expressions
- Rules for evaluating expressions

Core operators

- Selection (selects subset of rows)
- Projection (choose subset of columns)
- Product, join (glue)
    - Combining relations
- Union, intersection, difference
    - Combining relations
- Rename
    - Change name of attributes or relations (symbolism)

We can also have common extensions such as aggregation or projection++ or division.

Notations

Operation	Standard Notation	Our Notation
Selection	$\sigma_{expr}(Rel)$	*Sel[expr](Rel)*
Projection	$\pi_{A,B,C}(Rel)$	*Proj[A,B,C](Rel)*
Join	$Rel_1 \bowtie_{expr} Rel_2$	Rel_1 *Join[expr]* Rel_2
Rename	$\rho_{schema}Rel$	*Rename[schema](Rel)*

Semantics

We can define RA operations using

- Conditional set expression {x | condition for X}
- Tuple notations
    - t[AB] (extracts A and B from tuple t)
    - (x, y, z) tuple with type x, y, z
- Quantifiers, set operations, Boolean operators

We can algorithmically compute each operation to get the result of our query tuple, by tuple

We can also label results from our operations since RA maps relations to relations

```
Temp = R op1 S op2 T
Res  = Temp op3 Z
-- which is equivalent to
Res  = (R op1 S op2 T) op3 Z
```

Rename

- Schema mapping

If expression E returns a relation $R(A_1, A_2, \ldots A_n)$, then

$$Rename[S(B_1, B_2, \ldots B_n)](E)$$

gives a relation called S

- containing the same set of tuples as E
- but with the name of each attribute changed from A_i to B_i

Selection

- Returns a subset of tuples in relation r(R)

$$\sigma_C(r) = Sel[C](r) \doteq \{ t \mid t \in r \wedge C(t) \}$$

C is a boolean expression on attributes in R.

Result size: $|\sigma_C(r)| \leq |r|$

Result schema: same as the schema of r (i.e. R)

Algorithmic view:

```
result = {}
for each tuple t in relation r
    if (C(t)) { result = result ∪ {t} }
```

-

Projection

- Returns a set of tuples containing a subset of attributes in original table
- Select in sql is a projection, since select can be filtered to attributes we want.
- $\pi_X(r) = Proj[X](r) = \{ t[X] \mid t \in r \}$, where $r(R)$

Example projections:

Proj [A,B,D] (r1)

A	B	D
a	1	4
b	2	5
c	4	4
d	8	5
e	1	4
f	2	5

Proj [B,D] (r1)

B	D
1	4
2	5
4	4
8	5

Proj [D] (r1)

D
4
5

-
- Set semantics so no duplicates

Union

Combines two compatible relations into a single relation via set union

- Combines two relations which have same column types
- Both relations have same schema

$$r_1 \cup r_2 = \{ t \mid t \in r_1 \vee t \in r_2 \}, \quad \text{where } r_1(R), r_2(R)$$

$$\text{Result size: } |r_1 \cup r_2| \leq |r_1| + |r_2|$$

Intersection

Same as union but the tuple has to be the same in both r1 and r2

$$r_1 \cap r_2 = \{ t \mid t \in r_1 \wedge t \in r_2 \}, \quad \text{where } r_1(R), r_2(R)$$

$$\text{Result size: } |r_1 \cup r_2| \leq \min(|r_1|, |r_2|)$$

Difference

Finds the set of tuples that exists in one relation but not in a second compatible relation

$$r_1 - r_2 = \{ t \mid t \in r_1 \wedge \neg t \in r_2 \}, \quad \text{where } r_1(R),\ r_2(R)$$

Set difference like A – B, NOT SYMMETRIC 5-3 =/= 3-5

Example difference:

s1 = Sel [B = 1] (r1)

A	B	C	D
a	1	x	4
e	1	y	4

s2 = Sel [C = x] (r1)

A	B	C	D
a	1	x	4
d	8	x	5

s1 - s2

A	B	C	D
e	1	y	4

s2 - s1

A	B	C	D
d	8	x	5

Product

- Combines information from two relations pairwise on tuples
- Takes all of r1 and links it to r2

$$r \times s = \{ (t_1 : t_2) \mid t_1 \in r \wedge t_2 \in s \}, \quad \text{where } r(R),\ s(S)$$

Result size is large: $|r \times s| = |r|.|s|$ Schema: $R \cup S$

Algorithmic view:

```
result = {}
for each tuple t1 in relation r
    for each tuple t2 in relation s
        result = result ∪ {(t1:t2)} }
```

Natural join

Natural join will join two tables based on the common columns if they have matching values!

- Specialised product
    - Contains pairs that match on common attributes
    - One of each pair of common attributes is eliminated

Consider relation schemas $R(ABC..JKLM)$, $S(KLMN..XYZ)$.

The natural join of relations $r(R)$ and $s(S)$ is defined as:

$$r \bowtie s = r \text{ Join } s = \{ (t_1[ABC..J] : t_2[K..XYZ]) \mid t_1 \in r \wedge t_2 \in s \wedge match \}$$

where $match = t_1[K] = t_2[K] \wedge t_1[L] = t_2[L] \wedge t_1[M] = t_2[M]$

Algorithmic view:

```
result = {}
for each tuple t1 in relation r
   for each tuple t2 in relation s
      if (matches(t1,t2))
         result = result ∪ {combine(t1,t2)}
```

This is similar to natural join in SQL.

R join S = proj [R U S] (Sel [matching] (r x s))

If we wish to join relations, where the common attributes have different names, we rename the attributes first.

E.g. *R(ABC)* and *S(DEF)* can be joined by

R Join Rename[S(DCF)](S)

Theta Join

This is different to natural join, where instead we join based on a condition C

$$r \bowtie_C s = \{ (t_1 : t_2) \mid t_1 \in r \wedge t_2 \in s \wedge C(t_1 : t_2) \},$$
where *r(R),s(S)*

Examples: *(r1 Join[B>E] r2) ... (r1 Join[E≤D∧C=G] r2)*

If our condition is true, we will simply get a cross product of r and s

Example theta join:

r1 Join[D < E] r2

A	B	C	D	E	F	G
a	1	x	4	5	b	x
c	4	z	4	5	b	x
e	1	y	4	5	b	x

r1 Join[B > 1 ∧ D < E] r2

A	B	C	D	E	F	G
c	4	z	4	5	b	x

This is very common for SQL since most our joins are conditional

Division (hardest)

$$r / s = r \; Div \; s = \{ t \mid t \in r[R\text{-}S] \wedge satisfy \}$$

where $satisfy = \forall t_s \in S (\exists t_r \in R (t_r[S] = t_s \wedge t_r[R\text{-}S] = t))$

Operationally:

- consider each subset of tuples in *R* that match on *t[R-S]*
- for this subset of tuples, take the *t[S]* values from each
- if this covers all tuples in *S*, then include *t[R-S]* in the result

Example

Example:

R

A	B
4	x
4	y
4	z
5	x
5	y
5	z

R'

A	B
4	x
4	y
4	z
5	x
5	z

S

B
x
y

R / S

A
4
5

R' / S

A
4

In logical words, when we divide R/S we take all the values from S

- X, y
    - Then we check if value 4 has a matching x y
    - It has all of them so we take it
    - Then we check if value 5 has matching x y
    - It has all of them so we take it
    - In example with R', 5 only had x not x y so we don't include it for R'/S

This is useful for handling queries that include the for all notation, example which course has all students in enrolled

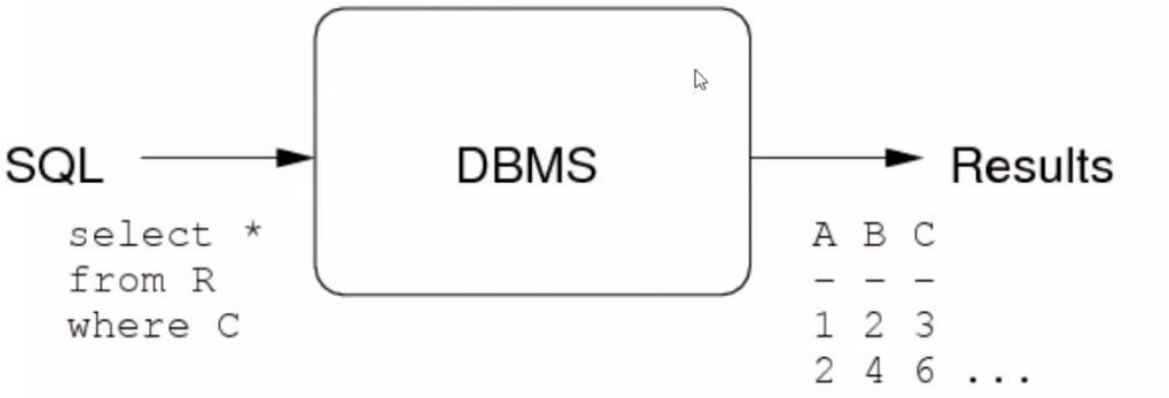
Or which beer is sold in all bars?

RA operations for answering the query:

- beers and bars where they are sold (ignore prices)
    - *r1 = Proj[beer,bar](Sold)*
- bar names
    - *r2 = Rename[r2(bar)](Proj[name](Bars))*
- beers that appear in tuples with every bar
    - *res = r1 Div r2*

Lecture 14

From a high-level view, we see our DBMS as a Blackbox where we supply questions and it returns answers



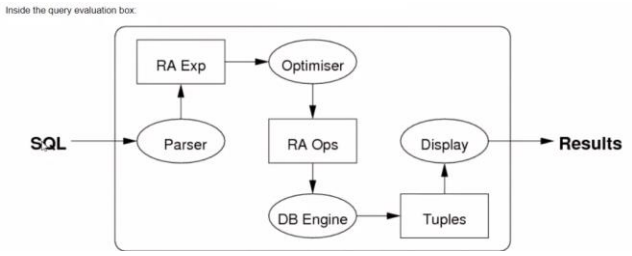
We want to find a way to

- Find/build a set of results that satisfy some set of conditions

We talk about *query* processing ...

- but the considerations also apply to DB update operations
- `INSERT` affects only one tuple (and, maybe, indexes)
- `DELETE` does *find*-then-remove, so is query-like
- `UPDATE` does *find*-then-change, so is query-like
- one difference: update operations act on a single table

So, another way to view our Blackbox is the following diagram



This is the inside of our DBMS "Engine"

So, the order of operations is

1. Parse our code into Relation Algebra expressions
2. Run Optimiser to return the set of Relational Algebra operators to perform
3. Run it through our DB engine which will return a set of tuples
4. Display those set of tuples

Some people see a database engine as a relational algebra virtual machine where for

selection (σ)	projection (π)	join ($\bowtie$, $\times$)
union ($\cup$)	intersection ($\cap$)	difference (-)
sort	insert	delete

we have multiple data structures/algorithms for performing this. Our Query optimiser picks best methods, data structures, algorithms to solve the queries.

Useful trick for checking time for update/insert/delete

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

                                                              QUERY PLAN
---------------------------------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.6
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual t
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0
               Index Cond: (unique1 < 100)
 Planning time: 0.079 ms
 Execution time: 14.727 ms

ROLLBACK;
```

Query Processing

The best method for evaluating a query is by using evaluation time.

Cost is measured in terms of pages read/written

- data is stored in fixed-size blocks (e.g. 4KB)
- data transferred disk↔memory in whole blocks
- cost of disk↔memory transfer is highest cost in system
- processing in memory is very fast compared to I/O

Mapping SQL to algebra

In general

- Select → projection (proj)
- Where condition → selection (sel) or join

example

```
-- schema: R(a,b) S(c,d)
select a as x
from   R join S on (b=c)
where  d = 100
-- mapped to
Tmp = Sel[d=100] (R join[b=c] S)
Res = Rename[a->x](Proj[a] Ttmp)
```

Note this is not the only way, we can choose when to select/join or perform operations

Example

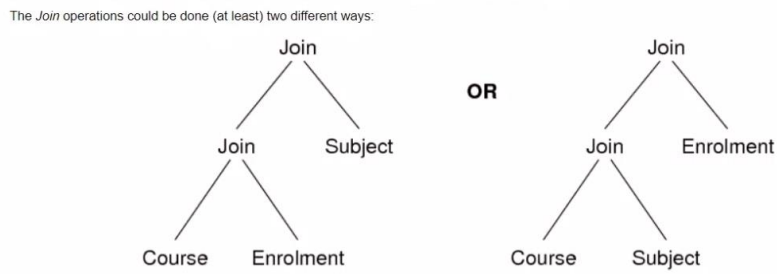
```
select   distinct s.code
from     Course c, Subject s, Enrolment e
where    c.id = e.course and c.subject = s.id
group by s.id
having   count(*) > 100;
```

Result = Project'[s.code](GroupSelect[size>100](GroupBy[id] (JoinRes)))

where

JoinRes = Subject Join[s.id=c.subject] (Course Join[c.id=e.course] Enrolment)

We can map out the join however to a tree to see which is better. The optimiser will work this out.



I would pick second approach as more filtering is done before second join. Join order is important always go small → large. Like answering assignment questions easy → hard.

Optimisation

- The query optimiser will start with an algebra expression then
    - Generate all equivalent expressions (set)
    - Generate execution plans for each one
    - Estimate cost of each plan and choose cheapest
- The cost of a query has factors including
    - Size of relations
    - Access mechanism (index/hashing/sort/join)
    - Size/number of memory buffers
- To analyse cost we need to estimate
    - Size of results
    - Storage access based on this size

Relational Algebra Operations

- Selection
    - Sequential scan (worst case O (N))
    - Index-based (B-trees O (logN)
    - Hash-bashed (O (1) best case, only works for equality though)
- Join
    - Nested-loop join (O (n.m), however we can buffer (materialise to reduce to O (N+M))
    - Sort-merge join (O(NlogN+MlogM))
    - Hash-join (best case O(N+M.N/B) with B memory buffers

Lecture 15

Transactions, concurrency, Recovery

DBMS provide access to valuable information resources in environments that are

- Shared between multiple users (concurrency)
- Unstable (potential for hardware/software crash)

Each users should see system as

- Unshared (their work is not affected by others)
- Stable (data survives in face of system failures)

Our ultimate goal is data integrity is maintained at all times

Transaction processing

- Techniques for maintain logical units of work which may require multiple DB operations

Concurrency control

- Techniques for ensuring multiple concurrent transactions do not interfere with each other

Recovery mechanisms

- Techniques to restore information to a consistent state even after major shutdown/failures on hardware/software end

Transactions

- An atomic unit of work in an application
- May require multiple database changes

Transactions happen in multi-user unreliable environments.

To maintain integrity of data transactions must be

- Atomic - either fully completed or totaly rolled-back
- Consistent - map DB between consistent states
- Isolated - transactions do not interfere with each other
- Durable - persistent, restorable after system failures

ACID Example

Example Transaction

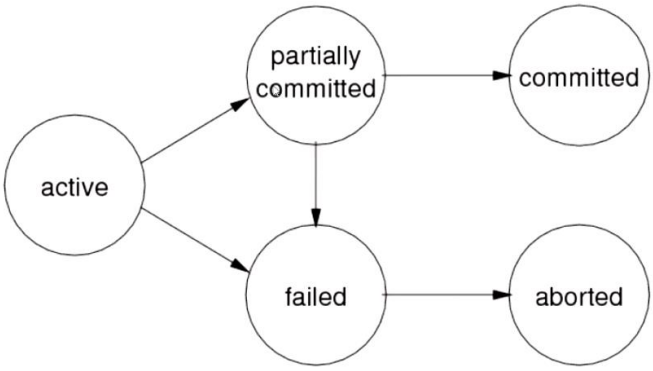
Bank funds transfer

- move *N* dollars from account *X* to account *Y*
- `Accounts(id,name,balance,heldAt, ...)`
- `Branches(id,name,address,assets, ...)`
- maintain `Branches.assets` as sum of balances via triggers
- transfer implemented by function which
  - has three parameters: amount, source acct, dest acct
  - checks validity of supplied accounts
  - checks sufficient available funds
  - returns a unique transaction ID on success

Raise exception will rollback – abort transactions. We can create the transfer as a function and place exceptions if we need to abort.

Modelling Transactions as State Diagrams

ith database unchanged



- At any point we can fail
- At end we go from active → {committed, aborted}

Modelling Transactions

Transaction effects include

- Read data from disk to memory, R(x)
- Write data from memory to disk, W(x)
- Abort, A
- Commit, C

To translate to SQL

- Select produces READ operations on database
- Insert products write operations
- Update/Delete produces both Read/write operations

During Transactions, typically we work in inconsistent states, however BEFORE and AFTER the transaction we will be in a consistent state.

Analysing Transactions in Abstract manner

We can abstract queries to just read/write operations

For example

- T1: R(x)W(x)R(y)W(y)
- T2: R(X)R(Y)W(x)W(y)

And we can use a schedule which will

- Provide a specific execution of one or more transactions
- Typically, perform tasks concurrently with interleaved operations (no collisions make sure we don't read/write to same data at same time)
    - Arbitrary interleaving of operations causes anomalies and will lead to a final state which is not consistent.

Serial Schedules

- If we have two transactions T1 and T2 we need to see

Do we run T1 then T2, or run T2 then T1

```
T1: R(X) W(X) R(Y) W(Y)
T2:                    R(X) W(X)
```

or

```
T1:           R(X) W(X) R(Y) W(Y)
T2: R(X) W(X)
```

Serial execution will guarantee a consistent final state if

- The initial state of the database is consistent
- T1 and T2 are consistency preserving.

However, tooooo slow and wasteful of resources

Concurrent scheduling

Concurrent schedules interleave transaction operations, some are okay some are not

Some concurrent schedules are ok, e.g.

```
T1: R(X) W(X)      R(Y)      W(Y)
T2:           R(X)      W(X)
```

Other concurrent schedules cause anomalies, e.g.

```
T1: R(X)      W(X)      R(Y) W(Y)
T2:      R(X)      W(X)
```

- In the second one, we write x twice which causes an anomaly
- We need to make sure each time we write we follow with a write in same transaction before another transaction accesses data.
- When we write DB applications, we need to be careful with how we let applications run in parallel

We want to make sure only valid scheduling

Serializability

Serializable Schedule

- Concurrent schedule for T1 … Tn with final state S
- S is also a final state of one of the possible serial schedules for T1 … Tn

So, if we run our concurrent schedule and have final state S, if any one of our transactions also reach final state S then the schedule is serializable

Two common formulations of serializability

- Conflict serializability (read/write operations occur in correct order)
- View serializability (read operations see the correct versions of data)

Conflict Serializability

Two transactions have potential for conflict if

- They perform operations on same data item
- At least one of those operations is a write operation

In these cases, the order we perform our operations matters to our results. If we were to have no conflict the order becomes irrelevant to our result.

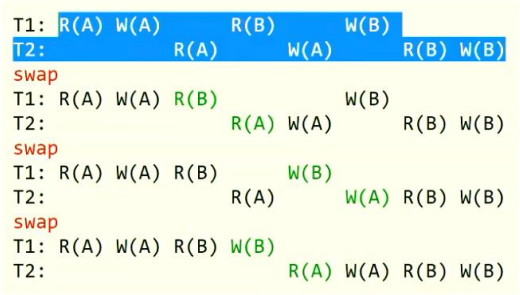
If we can transform a schedule by

- Swapping order of non-conflicting operations so that the result is a serial schedule
- Then the schedule is conflict serializable.

Example

Conflict Serializability (cont)

Example: transform a concurrent schedule to serial schedule



As can be seen we started with a concurrent schedule and we were able to transform it into two serial schedules by simply shifting across. We want to see if we can sift to a serial schedule. Note we cannot swap two write operations to same item this is not allowed. We need to accurately be able to say if something is conflict serializable fast!

To show something can or cannot be conflict-serialized we

1. Build a precedence-graph
2. Nodes represent transactions
3. Edges represent order of action on shared data
4. T1 → T2 means T1 acts on X before T2
5. Cycles indicate not conflict serializable

Concurrency Control

- Serializability tests are useful in theory but do not provide a mechanism for organising schedules
- A check is nice but it doesn't help us achieve our goal

Serializability tests

- Can only be done after the event
- O(n!) combinations

We need methods that

- Can be applied to each transaction individually
- Guarantee overall schedule is serializable

Acid transactions

- lock-based

  Synchronise transaction execution via locks on some portion of the database.

- version-based

  Allow multiple consistent versions of the data to exist, and allow each transaction exclusive access to one version.

- timestamp-based

  Organise transaction execution in advance by assigning timestamps to operations.

- validation-based (optimistic concurrency control)

  Exploit typical execution-sequence properties of transactions to determine safety dynamically.

Lock-based concurrency control

- Before reading X we get shared read lock on X
- Before writing X we get exclusive write lock on X
- An attempt to get a shared lock on X is another transaction has exclusive lock on X
- An attempt to get exclusive lock on X if another transaction has any kind of lock on X

These axioms alone do not guarantee serializability and can introduce deadlock and starvation

Locking will reduce concurrency.

Granularity of locking can impact performance:

+ lock a small item ⇒ more of database accessible

+ lock a small item ⇒ quick update ⇒ quick lock release

- lock small items ⇒ more locks ⇒ more lock management

Granularity levels: field, row (tuple), table, whole database

Many DBMSs support multiple lock-granularities.

Multi-version Concurrency Control

One approach to reduce requirements for locks is

- Provide multiple consistent versions of the database
- Give each transaction access to an “appropriate” version

Key difference between MVCC and standard locking concurrency control is

- Writing never blocks reading
- Reading never blocks writing

SQL concurrency control

Transactions in SQL are specified by

- **BEGIN** ... start a transaction
- **COMMIT** ... successfully complete a transaction
- **ROLLBACK** ... undo changes made by transaction + abort

In PostgreSQL, other actions that cause rollback:

- **raise exception** during execution of a function
- returning null from a **before** trigger

SQL allows for

- Table-level locking
- Row-level locking

LOCK-TABLE will acquire the lock on whole table

Other SQL commands also acquire locks

- ALTER TABLE
- UPDATE/DELETE

All locks are released at end of transaction, no need for “unlock”

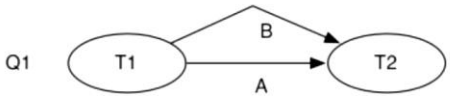
In general, we don’t want to lock table especially in transaction intensive applications.

Exercise

```
Schedule Q1 T1: R(A) W(A)      R(B)      W(B)
            T2:           R(A)      W(A)      R(B) W(B)
```

```
Schedule Q2 T1: R(A)      W(A)      R(B)      W(B)
            T2:      R(A)      W(A)      R(B)      W(B)
```

```
Schedule Q3 T3:      R(B)                W(B)
            T4: R(A)      W(A) R(B)                W(B)
            T5:                      R(A)      W(A)
```




```
------------
Schedule Q1:
------------

T1: R(A) W(A)      R(B)      W(B)
T2:           R(A)      W(A)      R(B) W(B)

T1:W(A),T2:R(A) conflict gives  T1 --> T2
T1:W(B),T2:R(B) conflict gives  T1 --> T2

Graph has no cycles => serializable
 ------------
 Schedule Q2:
 ------------

 T1: R(A)      W(A)      R(B)      W(B)
 T2:      R(A)      W(A)      R(B)      W(B)

 T2:R(A),T1:W(A) conflict gives T2 --> T1
 T1:W(A),T2:W(A) conflict gives T1 --> T2
------------
Schedule Q3:
------------

T3:      R(B)                W(B)
T4: R(A)      W(A) R(B)                W(B)
T5:                      R(A)      W(A)

T4:R(B),T3:W(B) conflict gives  T4 --> T3
T4:W(A),T5:R(A) conflict gives  T4 --> T5
T3:W(B),T4:W(B) conflict gives  T3 --> T4
```

A dirty read occurs when a transaction reads an object that has been modified by another not-yet-committed transaction

Example

- T1 reads A

- T2 reads A
- T2 writes A

Now T1 has a different version of A

Lecture 16

d. *conflict-serializable schedule*

A schedule is conflict-serializable if it is conflict-equivalent to some serial schedule. Two schedules are conflict-equivalent if they involve the same set of actions and they order every pair of conflicting actions in the same way.

e. *view-serializable schedule*

A schedule is view-serializable if it is view-equivalent to some serial schedule. Two schedules are view-equivalent if they satisfy:

- the initial value of any object is read by the same transaction in both schedules, and
- the final value of any object is written by the same transaction in both schedules, and
- any shared object is written-then-read by the same pair of transactions in both schedules.

Two-phase locking

Lock protocol but we add extra protocols to ensure that all results produce serializable schedule

- Acquire a shared lock on object before reading
- Acquire an exclusive lock on an object before writing
- Not acquire any new locks once it has released a lock
    - We will get a serializable schedule through this

Performance Tuning

Sometimes we need to write code outside DBMS for

- GUI or any user interface
- Interaction with other system
- To perform compute-intensive work vs data-intensive
    - Algorithms that cannot be done in DBMS

More conventional programming languages offer these.

Before this we want to pass as much data into our program from DBMS as possible as DBMS is good at data work and much faster than most PL. To understand this however, we need to understand how query optimisation works.

The query optimiser starts with a relational algebra expression

- Generates a set of equivalent expressions
- Generate execution plans for each with costs attached, will then pick cheapest

The cost is determined by

- Size of relations
- Access mechanisms
- Size/number of memory buffers

To properly analyse cost we need to estimate the size and how many i/o calls there are from DBMS. This can be seen when we do explain query in DBMS.

Advice for faster queries

- Filter early
    - When doing joins filter as early as possible!
- Don't block the optimiser by materialising or writing functions into queries etc
- Select what you need not just all
    - Don't just select * blindly. Only select what we need
    - Filter and keep only what we need be smart

We want to devise data structures to achieve good performance. We need to tune our applications for maximum transactions with fast response time. To do this we need to analyse

- Which queries and transactions will be used
- How frequently does each query/transaction occur?
- Are there time constraints on queries/transactions?
- Are there uniqueness constraints on attributes?
    - can define index to speed up insertion uniqueness
- how frequently do we update?
    - index slow down updates because it updates table AND index

Denormalization

Normalization minimises storage redundancy by breaking up data into logical chunks requiring low maintenance to preserve consistency. However, queries might need to put data back together and this can require multiple join operation, and if expensive joins are frequent, then performance can suffer. To solve this

- store some data redundantly
    - queries needing expensive joins are now cheap
    - trade-off is extra maintenance to maintain consistency
    - worthwhile however if joins are frequent and updates are rare

Example of denormalization

```
-- can now replace a query like:
select s.code||t.year||t.sess, e.grade, e.mark
from   Course c, CourseEnrolment e, Subject s, Term t
where  e.course = c.id and c.subject = s.id and c.term = t.id
-- by a query like:
select c.courseName, e.grade, e.mark
from   Course c, CourseEnrolment e
where  e.course = c.id
```

Here instead of doing an expensive string append operator frequently, we can simply store the name together in one attribute

- trade-off can be worthwhile if course insertions/updates are infrequent
- cost is to create trigger before insertion to construct name, and maybe on update/delete to maintain name.

Indexing

Indexes provides fast content-based access to tuples.

```
CREATE INDEX name ON table ( attr1, attr2, ... )
```

We can simply just put the index on the table, or apply index to the attributes we want to index. Before using an index however we need to ask

- is an attribute used in frequent/expensive queries?
- can a query be answered by index alone?
- is the table containing attribute frequently updated?
- should we use B-tree or Hash index?
    - Use Hash for unique attribute equality testing
        - E.g. select * from Employee where **id = 123**
    - Use B-tree for attributes in range test
        - E.g. select * from Employee where **Age < 50**

We can specify index by saying

- Create index name on table (attr1, attr2) using Btree/Hash

Tuning

Sometimes we can rephrase query to massively boost speed

- Helps optimiser make use of indexes
- Avoids unnecessary/expensive operations

Examples of queries which prevent optimiser from using indexes

```
select name from Employee where salary/365 > 100
       -- fix by re-phrasing condition to (salary > 36500)
select name from Employee where name like '%ith%'
select name from Employee where birthday is null
       -- above two are difficult to "fix"
select name from Employee
where  dept in (select id from Dept where ...)
       -- fix by using Employee join Dept on (e.dept=d.id)
```

Instead of ilike if we can try to use name = if possible

Other tricks for query tuning

- Select distinct requires sorting
    - Is sorting necessary (O(nlogn) time)
    - Put it as late as possible or try to do it when there is little data to distinct
- If multiple join conditions are available try to choose join condition on attributes that are indexed
    -
```
select ... Employee join Customer on (s.name = p.name)
vs
select ... Employee join Customer on (s.ssn = p.ssn)
```
    - Avoid joining on strings if can be

- Sometimes or will prevent the index from being used, replace the or condition by a union of non-or clauses
    - ```
      select name from Employee where Dept=1 or Dept=2
      vs
      (select name from Employee where Dept=1)
      union
      (select name from Employee where Dept=2)
      ```

Lecture 17

Catalogs

An RDBMS maintains a collection of relation instances. To do this it needs information about

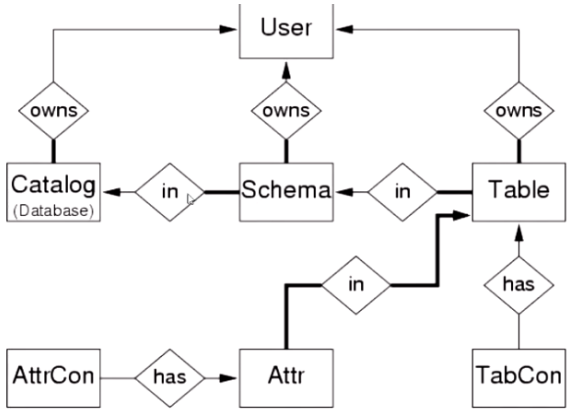
- Name, owner, primary key of each relation
- Name, data type, constraints for each attribute
- Authorisation for operations on each relation

This information is stored in the system catalog in the database itself. This is called the meta-data.

DBMS uses a hierarchy of namespaces to manage names

- Database (catalog level)
    - Top level namespace containing schemas
        - Default schema is public unless you modify this
    - Users connect to/work in a database
- Schemas
    - Second level namespace; contains tables, views etc.
    - Users typically work within a current schema
- Table
    - Lowest-level namespace; contains attributes
    - Select queries set a context for attribute names

We can draw an ER diagram for system catalog



(A small fragment of the meta-data tables in a typical RDBMS)

In 2003 SQL set a standard for metadata known as INFORMATION_SCHEMA containing

This is similar to assignment1, how everyone used the same naming for views and attributes, this allowed for 1 autotest to mark everything in 1 burst. This is similar to the INFORMATION_SCHEMA. What companies did was create a view which contained their meta-data standard information rather than rewrite everything from the ground up.
      ```

When we run psql a1

- It will load into a1 database with the default schema being public
- If we want to set to a different schema, we can do
    - Set schema 'Schema'
- We can access objects in other schemas using Schema.Object
- PSQL provides SQL standard INFORMATION_SCHEMA aswell
- This also allows us to set different permissions public, private etc.

We can see all tables in the database by doing

```
create or replace view myTables
as
select table_name
from   information_schema.tables
where  table_schema = 'public' and table_type = 'BASE TABLE'
order  by table_name;
```

- We can also check the columns table to see how many attributes in each view or anything really a database offers through metadata

We only query the system catalog if

- Auto testing
- A system admin, management
- Database tuner

Can almost automate to find bad queries or bad performance items.

Another example to see db schema

```
create or replace view myTableColumns
as
select t.table_name, c.column_name
from   information_schema.tables t
     join information_schema.columns c
     on (t.table_name = c.table_name)
where  t.table_schema = 'public' and t.table_type = 'BASE
TABLE'
order  by t.table_name, c.ordinal_position;

create or replace view mySchema("table","attributes")
as
select table_name, concat(column_name)
from   myTableColumns
group  by table_name
order  by table_name;
```

Example for size of each table

```
create or replace function
    dbpop() returns setof PopulationRecord
as $$
declare
    r record;
    nr integer;
    res PopulationRecord;
    countQry text;
begin
    for r in select tablename
             from   pg_tables
             where  schemaname = 'public'
             order  by tablename
    loop
--    select count(*) into nr from r.tablename DOES NOT WORK
       countQry := 'select count(*) from '||r.tablename;
       execute countQry into nr;
       res."table" := r.tablename;
       res.ntuples := nr;
          return next res;
    end loop;
end;
```

Security, Privileges, Authorisation

Access to DBMS involves two aspects

- Execute permissions for a DBMS client
- Username/password registered in DBMS
    - Stored in DBMS all of it

Establishing connection to the database:

- User supplies db/username/password to client
- Client passes them to server which will authenticate
- If valid user is logged in to db with their role

Psql will figure out who is the user by who ran the client process.

To access a database via web involves

- Running a script on a web server
- Using web servers access rights on the DBMS

The access is specified in pg_hba.conf file.

We can create user types with permissions by the following

```
CREATE USER Name IDENTIFIED BY 'Password'
ALTER USER Name IDENTIFIED BY 'NewPassword'
ALTER USER Name WITH Capabilities
ALTER USER Name SET ConfigParameter = ...
```

We can also create roles for users which have certain permissions

```
CREATE GROUP Name
ALTER GROUP Name ADD USER User1, User2, ...
ALTER GROUP Name DROP USER User1, User2, ...
```

Examples of groups/roles:

- AcademicStaff ... has privileges to read/modify marks
- OfficeStaff ... has privilege to read all marks
- Student ... has privilege to read own marks only

DB access control in PSQL

In older versions of Postgres

- Users and groups were distinct objects
- Users added via create users
- Groups added via create group
- Groups were built up using alter group … add user

Nowadays users and groups are unified by their roles. Older version is retained only for backward compatibility.

To create user in command line we use

- Createuser “name”

To create user from SQL

```
CREATE ROLE UserName Options
-- where Options include ...
PASSWORD 'Password'
CREATEDB | NOCREATEDB
CREATEUSER | NOCREATEUSER
IN GROUP GroupName
VALID UNTIL 'TimeStamp'
```

Groups are created as Roles

```
CREATE ROLE GroupName
--or--
CREATE ROLE GroupName WITH USER User1, User2, ...
```

And we can modify the role by

```
GRANT GroupName TO User1, User2, ...
REVOKE GroupName FROM User1, User2, ...
GRANT Privileges ... TO GroupName
REVOKE Privileges ... FROM GroupName
```

SQL Access Control

SQL only deals with

- Privileges on database objects
- Allocating these privileges to certain roles

The user who creates an object is automatically assigned

- Ownership of the object
- A privilege to modify/remove the object
- Along with every privilege for the object also including the ones below.

The owner of an object can assign privileges on that object to other users.

Accomplished via the command:

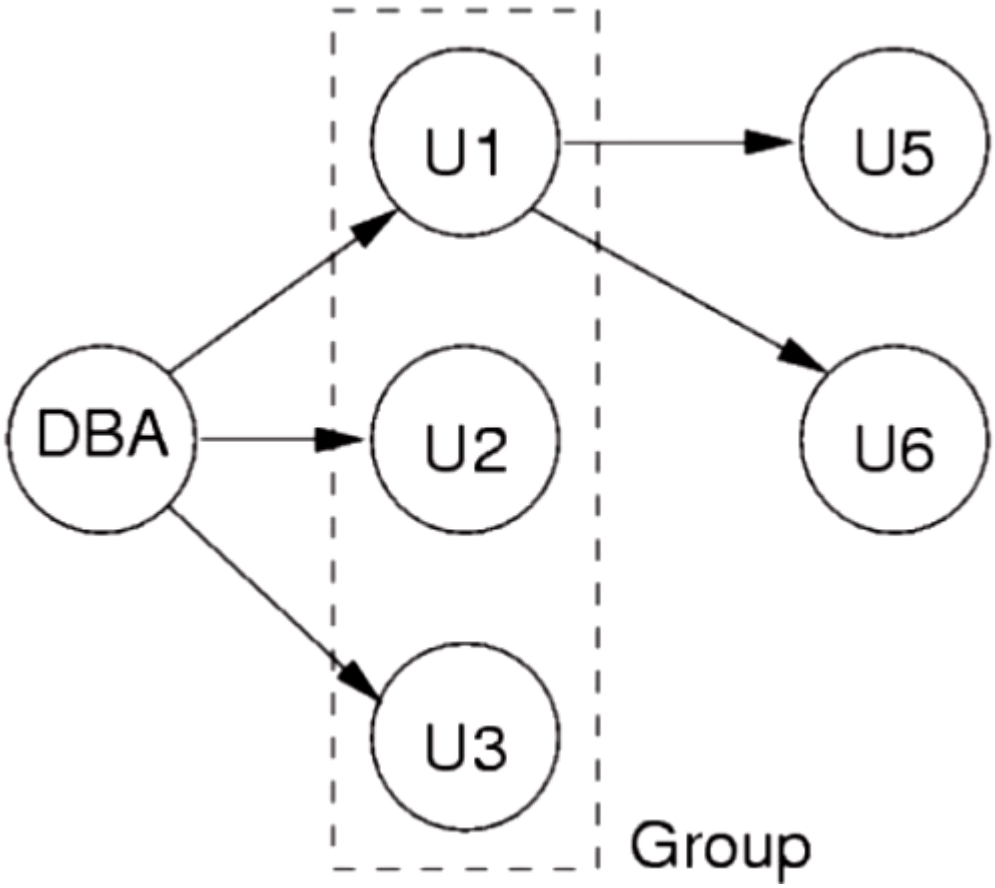
```
GRANT Privileges ON Object
TO ( ListOfRoles | PUBLIC )
[ WITH GRANT OPTION ]
```

Privileges can be ALL (giving everything but ALTER and DROP)

Grant option sort of allows us to be admin and pass down privileges

Real World applications

In a real-world scenario, we can be part of multiple roles and sometimes there are conflicts and this can all be resolved using an authorisation graph. This is managed at a system level, for example DENY has a higher priority over ALLOW so if conflict arises this is how it is dealt with. This can cause many issues however



We can also revoke privileges from objects in the following

```
REVOKE Privileges ON Object
FROM ListOf (Users|Roles) | PUBLIC
CASCADE | RESTRICT
```

Cascade also withdraws from users the object has granted the privileges to. Restrict will fail if the user has passed the privileges to others.

Privileges

- Select privilege
    - User can read all rows and columns of table/view
    - This includes columns added later via ALTER TABLE
- INSERT (optional colName)
    - User can insert rows into table
    - If ColName is specified we can only set value of that column.
        - Note this requires other values

- Update
    - User can modify values stored in database
- Update (column)
    - User can update specified column
- Delete
    - User can delete rows from table
- References (column name)
    - User can use column as foreign key
- Execute
    - User can execute the specified function
- Trigger
    - User is allowed to create triggers on table

Final Lecture

Goal of databases

- Deal with large amount of data (terabytes, petabytes etc.)
- Very-high-level languages (deal with big data in uniform ways)
- Query optimisation (evaluation too slow means useless)

Three phases in DBMS history:

- 1960's: pre-history, first attempts at generalised data access
- 1970's-2000's: relational databases, structure, transactions
- 2000's: Big Data, less structure, relaxed transactions

Off topic history

1960's

- Simple structured records
    - Atomic valued records
- Hierarchical data model
    - All data organised via tree
    - Access by following paths in hierarchy
    - Could only represent 1:n relationships
- Network data model
    - Relationship between records via links
    - Query via traversal of record graphs
    - Developed in late 60's and stayed till 80's

1970's RDBMS rise

- Codd developed relational model
    - Set of tuples, relational algebra
- IBM developed a DBMS based on model (System R (Relational))
    - High level query language
    - Implementation of efficient relational operations
    - Implementation of transaction/recovery
    - System R eventually developed into DB2
- Larry Ellison (java god) commercialised the idea into Oracle

1980's development of RDBMS

- Distribution/replication of data
- Improving implementation of relational operators

1990's Standardization and extension

- SQL standard developed
- SQL extended with PL/SQL, recursion and so on
- Improving implementation of relational operators

2000's New data

- Temporal aspects of data
- Stream data (continuous queries)
    - Starts and only stops when specified to e.g. print top 10 stock prices, since constantly changes we need to continuously print
- High dimensional data (e.g. image feature spaces)

RDBMS have been dominant DB technology for 40 years

- Assume that
    - Data resides on disk device (latency, block-transfer)
        - Virtual memory, page tables etc.
    - Data located on a single machine (can be distributed)
    - Data fits on large array of disks on a local access network
    - Data can be represented by atomic-valued tuples
    - Queries involve precise matching (equality, greater/equal etc.)

But nowadays we also have

- Growth in use of SSD for data storages
    - Much faster and optimisation is different for hardware change
- Need to store very large amounts of data

Limitations/pitfalls of RDBMS

- NULL is ambiguous: unknown, not applicable, not supplied
- Limited support for constraints/integrity and rules
- No support for uncertainty, data represents the state-of-the-world
    - Probability can not be put into queries? Apparently bad from raymond
- Data model is too simple
    - No support for complex objects
- Query model is too rigid, no approximate matching
- Continually changing data sources is not handled well
- Data must be "moulded" to fit a single rigid schema
    - Especially hard when we need schema/data from companies that are likely to not give e.g. banks
- Database systems must be manually tuned
- Do not scale well to some data such as
    - Google
    - Amazon
    - Telco etc.

To overcome some of these limitations we can

- Extend the relational model
    - Add new data types and query optimisation for new applications
    - Deal with uncertainty/inaccuracy/approximation in data
- Replace the relational model
    - Object-oriented DBMS OO programming with persistent objects (OP) can apply design patterns
    - XML DBMS, all data stored as XML documents, new query model
    - Application-effective data model, e.g. using key value model like a giant hashtable, google uses BigTable which is almost like a ultra-giga-wumbo hashtable
- Self-tuning DMS

Big Data

Some modern applications have massive data sets (e.g. Google)

- Far too large to store on a single machine/RDBMS
- Query demands far too high even if could store in DBMS

Approach to dealing with such data

- Distribute data over large collection of nodes (redundancy)
- Provide computational mechanisms for distributing computation
- Allows distribution of load

Often this data does not need full relational selection

- Data represented via key, value pairs (giant hashtable)
- Unique key values can be used for addressing data
- Values are large objects such as web pages, emails etc.

Map/reduce approach

- Suitable for widely distributed, very-large data
- Allows parallel computation on such data to be specified
- Distribute parts of computation across network
- Compute in parallel
- Merge multiple results for delivery to requester

Some Big data advocates say no future for SQL/RDBMS

- Depends on application (e.g. hard integrity vs eventual consistency)

Information retrieval

- DBMS generally do precise matching with exception of ILIKE and regex
- Information retrieval systems do approximate matching
    - Search engine for queries
- E.g. documents that have certain words (google)
- Also introduces notion of “quality” of matching
    - A heuristic to say T1 is a better match than T2 (tuples)
- Quality also implies ranking of results
- Much activity in incorporating IR ideas into DBMS
- We want to support database exploration better

Multi-media Data

Data which does not fit the tabular model

- image
- video
- music
- text
- or a combination

Research problems

- how do we specify queries on this data
    - image1 similair to image2?
- how to display results?
- To solve this we try to
    - Extend idea of matching and indexing for queries
    - Sophisticated methods for capturing features of data
- E.g. find other songs like this
    - Might analyse byte pattern of song.

Uncertainty

- Multimedia/IR introduces approximate matching
- E.g. if we have crime db how do we store
    - “car was red or maybe orange”
    - “75% sure john did crime”
    - How do we store these in DB
- Stanford research lead by Jennifer Widom to solve this

Stream Management

Makes a addition to regular relational model

- Stream of data
    - Infinite sequence of tuples arriving one at a time until requested to halt
- Useful in applications like twitter/news feeds/telecoms/monitoring web usage
- RDBMS will run different queries on fixed data

two approach are

- Window
    - Relation formed from a stream via rules

- Stream data type
    - Build new stream specific operations

Semi-structured Data

- Uses graphs rather than tables as data structure tool
- Applications include
    - Complex data representation via “flexible” objects for instance XML
- Since data presented in graph, this will change query model
    - XQuery language, high-level blackbox like SQL but different operations
- Implementing graphs in RDBMS is most times inefficient
- Research problem: query processing for XML data

Dispersed Databases

Uses in IOT

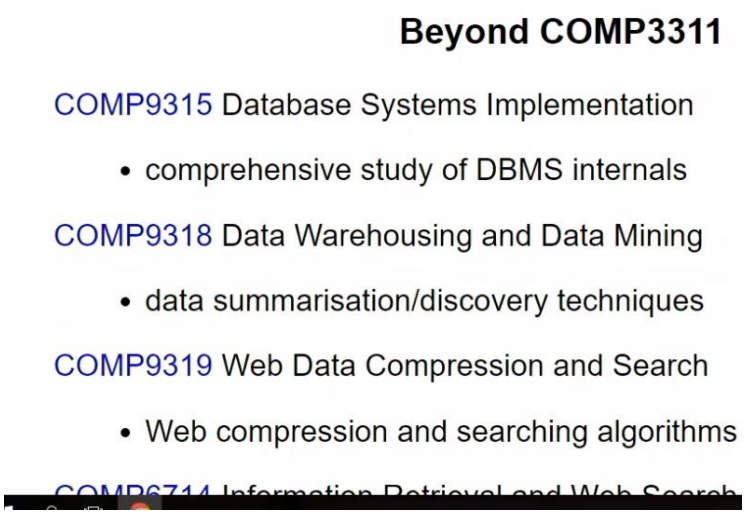
- Large amount of small processing nodes
- Data is distributed between nodes
- Network of database among nodes

Very challenging problems include

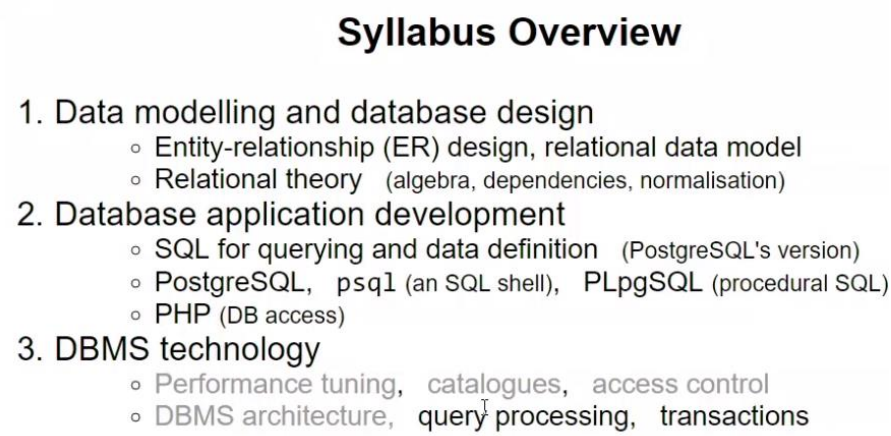
- Query/search strategies (how to organise query processing)
- Distribution of data (trade off between centralisation and diffusion)

Less extreme versions are in place

- Grid and cloud networks
- DBMS management for mobile devices.

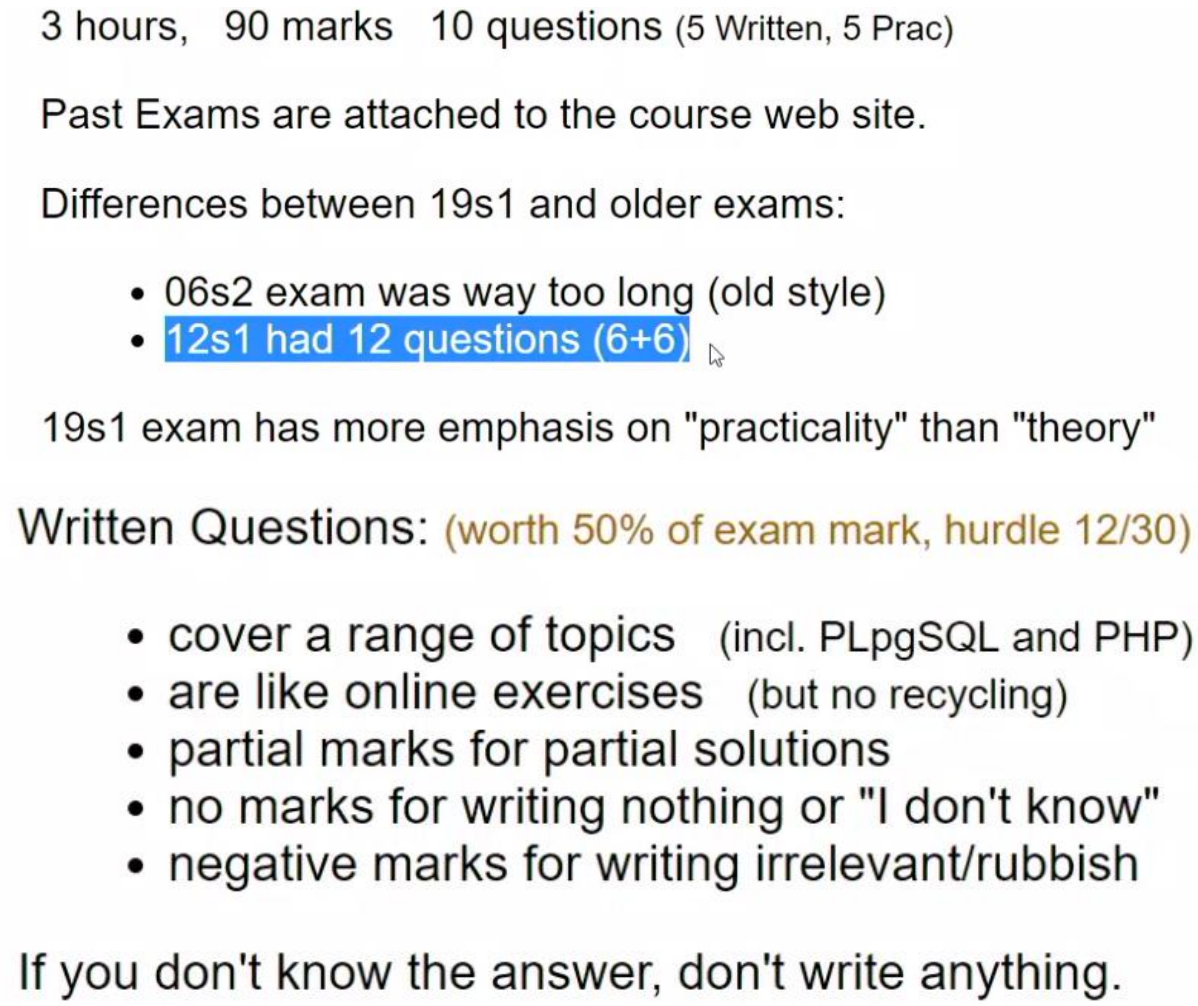


Exam preparation



The grey shit won't be examined but it's good to know!

- Exam has more emphasis on practicality over theory



Prac Questions: (worth 50% of exam mark, hurdle 12/30)

- are like SQL questions from previous exams
- vaguely like Assigt SQL, but DB much simpler/smaller
- write SQL queries/views (using **SQLite**)
- no PLpgSQL or PHP in prac exam
- some (not many) marks for incomplete solutions

Exam working environment:

- using standard CSE exam environment
- the usual collection of editors (e.g. pico, gedit, vim, emacs)
- `xfig` and `dia` for drawing ER diagrams, if any
- `sqlite3` for doing the SQL prac questions
- from within the web browser, you will have
  - exam paper (instructions and all questions)
  - SQLite3, PostgreSQL, PHP docs (copied from course web site)
- all answers will be typed/drawn into files
- submit answers via `give` (special exam version)

Sources for revision material:

- COMP3311 Course Notes, Theory Exercises
- Fundamentals of Database Systems, Elmasri/Navathe
- Database System Concepts, Silberschatz/Korth/Sudarshan
- Database Management Systems, Ramakrishnan/Gehrke
- Database Systems: Complete Book, Garcia-M/Ullman/Widom
- Database Systems: App-oriented, Kifer/Bernstein/Lewis
- SQLite/PostgreSQL/PHP Documentation (learn to navigate)