COMP 421 Study guide

Francis Piché

February 21, 2019

Contents

Part I
Preliminaries

1 License Information

These notes are curated from Joseph D'silva COMP421 lectures at McGill University. They are for study purposes only. They are not to be used for monetary gain.

Part II
Introduction to Databases

2 Data Storage

In operating systems, file systems are used for persistent storage. This is insufficient because:

- Only provides a basic API
- Information may not be structured.
- Only linear seek possible (slow)
- May have data loss in case of power outages etc
- Can't have concurrent access to files

3 Relational Model and Data Definition Language

Entity Relationship Model is a language that describes the data determined through requirement analysis. This is very similar to UML, but is not the same.

3.1 Requirement Analysis

When designing a database, we must first identify which data needs to be stored, and how it will be used. (Which operations need to be executed on the data).

For example, if we were designing a database for Minerva at McGill, we would need to think about which entities are relevant to be stored. In OOP, these would be the classes.

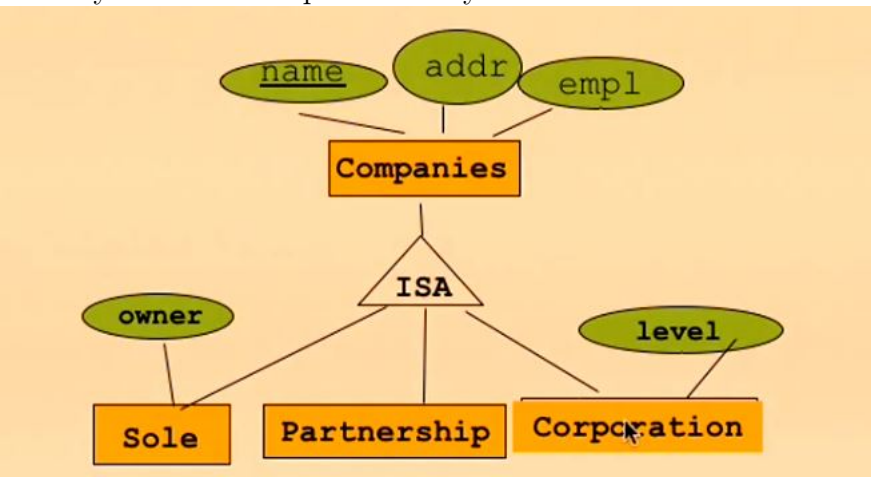
Things like the students, addresses, fees, courses etc are all relevant information. We would also need to think about the types of operations that must be possible by our database. Things like adding students, changing addresses, assessing fees etc.

In an OOP setting, we would break down the problem by identifying classes and relationships between them. For example, we might have Student and Instructor classes, with a parent class Person, since the Student and Instructor share some attributes (name, age...). We also have relationships between classes such as a Student and Transcript. A Transcript does not exist unless assigned to a Student. We need ways to model all of these relationships in a database setting rather than an OOP setting.

3.2 ER

An *entity* is a real world object which contains a set of attributes. A collection of instantiated entities would be an *entity set*. An entity set must have a *key*. This is an attribute that is underlined, and MUST be unique. There can be two attributes underlined, in which case both COMBINED must be unique. Individually they need not be unique.

ISA ("is a") hierarchy is similar to subclasses in OOP. This is represented as a triangle. The key is still in the parent entity set.



There is a subtle point here in that unlike OOP, the attributes of the parent are not part of the child entity set (to avoid duplication, and keep things consistent).

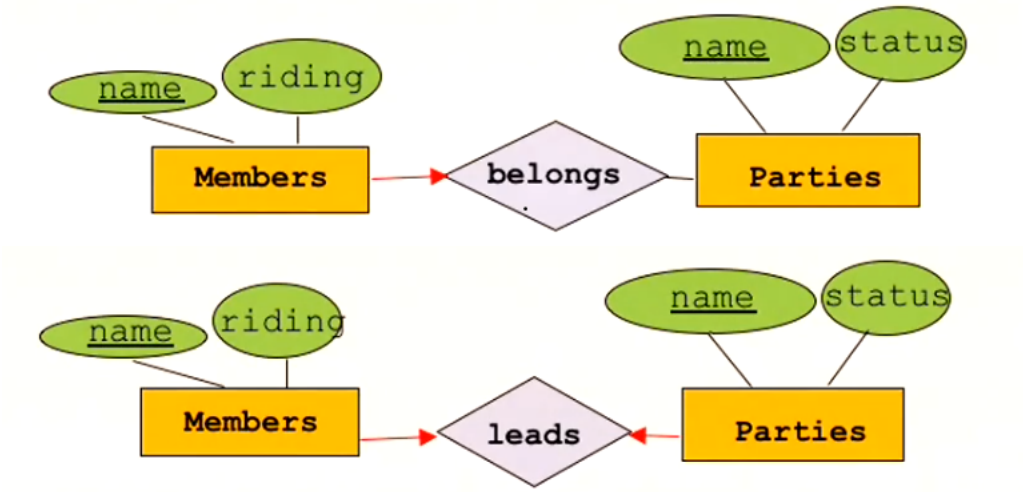
If there are overlaps (a company can be both a partnership and Corporation), we must put a note explaining the situation. We assume that the parent is not covering all possible subtypes. (There may exist a company which is not a Sole, Partnership or Corporation).

3.3 Relationships

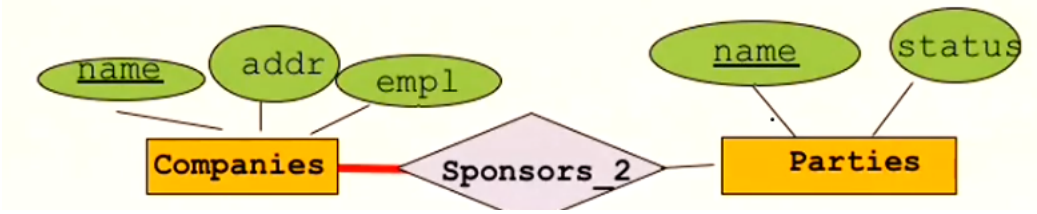
A relationship is an association among two or more entities. This can be one to one, many-to-one, many-to-many. A relationship set is a collection of similar relationships.

There cannot be duplicate association instances. For example, if a Company donates 10$ to a Party, (the association being the sponsorship), then there cannot be another 10$ donation with the same Company and Party. The original must be updated.

There are constraints as well. We can add an arrow, or two, to constrain to a one-to-many or one-to-one (respectively).



Participation constraints are when we must have at least one entity set is participating in the relationship.



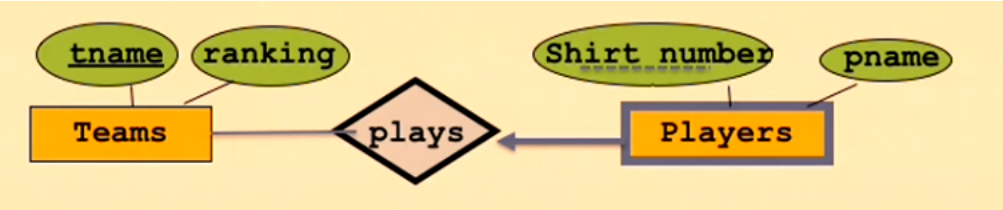
which is shown by a bold line. This can be combined with a key constraint to show a relationship in which exactly one entity set is related.

Ternary relationships are relationships involving 3 entity sets. Keep in mind that a ternary relationship database entries MUST include all 3 entities. If you have more than a ternary relationship (n-ary), it's probably an indication of bad design.

3.4 Weak Entities

Weak Entities are entities which do not have a natural attribute which can be used as a key, but there is another entity set with which we can identify the entities. For example,

suppose we have Teams and Players. The teams have names, Players have names and shirt numbers. Here, the Team name may be unique for a league, but not overall, and a shirt number may be unique within a team, but not for the league. However, we can use the Team name as a primary key, with the Player shirt number as a partial key to identify a player. Note that this implies that a Player cannot exist without belonging to a Team.

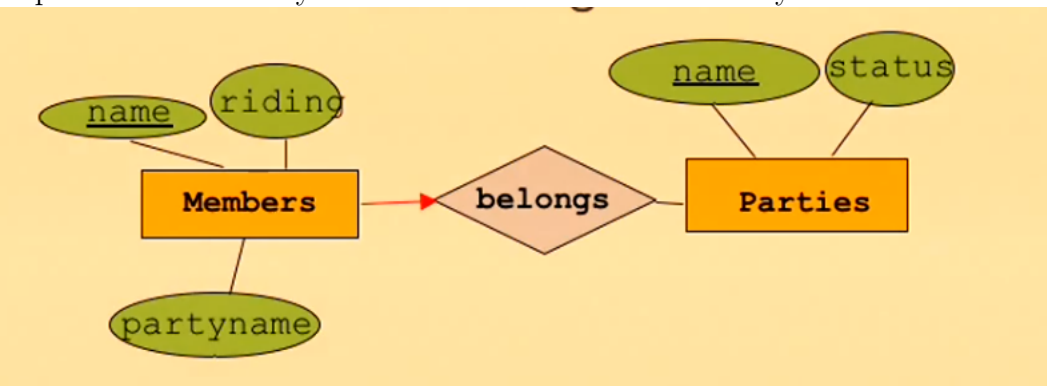


We often create artificial keys to keep track of information. For example student numbers. However, you should never use an artificial key as a partial key. If you're going to make one, make sure it's unique.

We use weak entities when there is no global authority that allows creating unique identifiers.

3.5 Avoiding Redundancies

We should avoid redundancies by not storing the same information more than once. This is important since it may allow for data to become out of sync.



suppose someone changed the partyname but not the name of the associated party. The data would be inconsistent.

3.6 Entity Set or Attribute

When can an entity set simply be an attribute? If we only care about the key of an entity set, and we only have a one to many relationship, we can represent it instead as an attribute. The one-to-many is important since we can't have a list as an attribute value. For example, if we have Movies, with the attribute Category (rather that have a relationship between Movie

and Category), we can't have a list of Categories if the Movie belongs to several Categories. This would require leaving it as a relationship rather than an attribute.

4 Relational Model

This is the most common model, and is closer to actual implementation. Used by iBM DB2, PostgreSQL, MySQL, SQLite etc.

There are other types of models, which we will either ignore or look at later.

A relational database is nothing more than a set of relations. A relation consists of a schema and instance. A schema defines the name of a relation, its attributes, and the domain/type of each attribute. `Students(sid: int, name: string, login: string, faculty: string, major: string)` Ie: relation == table. Not to be confused with relationship in ER modeling (different things!). The instances are actual tuples (rows) in the table.

We cannot have identical instances (duplicate rows), at least one attribute value must be different. We also cannot assume anything about the order of the rows. The column order also doesn't matter.

4.1 DDL and DML

DDL (data definition language) defines the schema of a database. (What the tables are called, their attributes) This only gives structure, no real data.

DML (data manipulation language) manipulates the data. (deals with instances of the relations). For example, insert, update delete, query etc.

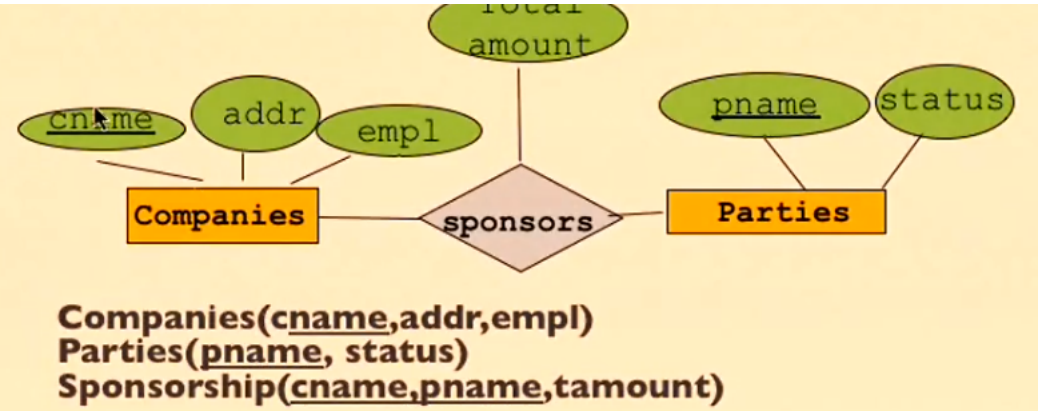
5 Translating from ER to DDL

Entity sets are easy, just use the name of the entity set and its attributes. `EntitySetName(attr1: type, attr2: type ... attrn: type)`

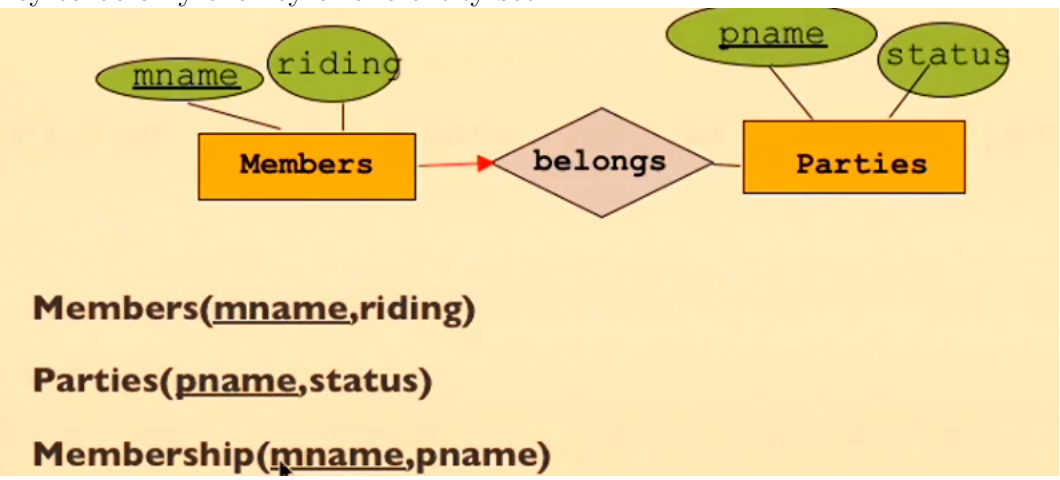
5.1 Relationships

What if there is a relationship? Well first translate the entity sets involved, then the relationship is a relation with the key of each entity set as it's key, and then it's attributes.

5.2 Key Constraints



What if there are key constraints? For a one to many relationship, we can make the primary key to be only the key of one entity set.



So here, we cannot have one member belonging to multiple parties in both the ER and the DDL (since mname must be unique in a table, and the key constraint is in the diagram).

We can actually do better here. By moving the pname attribute up into the Members table as an attribute, we can reduce the total number of tables by 1. (Eliminating the need for the Membership table). However, this is only good if all members will have a party. If the new column we created in Members is sparse (a lot of NULL's) then its a big waste of space, in which case the 3 table approach is better.

5.3 Participation Constraints

Participation restraints are generally enforced by a NOT NULL on the foreign key. If we have a participation constraint on a key constraint, our 3 table solution from before no longer works. We now need to go with the 2 table solution since there is no way to enforce that the Members table has a corresponding entry in the Membership table (separate tables).

If there is no key constraint along with the participation constraint (just the participation constraint on its own) **THERE IS NO WAY TO ENFORCE IT.** It's an important note that **all** many-to-many relationships require 3 tables. And since participation constraints require 2 tables. So we must put a note to indicate that it should be a participation constraint.

5.4 Weak Entity Sets

Translating weak entity sets requires both the partial key of the weak entity set, and the key of the entity set on which the weak depends on.

5.5 ISA Hierarchies

For ISA, we need to copy the key of the parent into all child tables, and then just the attributes of each child.

There is another way which is to copy ALL parent attributes and then add the child-specific attributes. This might lead to duplication, but also make things easier on an application/-efficiency level.

A last alternative is to have only a table for the root (parent) entity set, with attributes for all the attributes contained in all the children, with NULL's if the entry does not have that datatype. This easy to store, but requires a lot of care in the application, since there is nothing stopping NULL values to be updated with real values, causing inconsistency with the model.

6 Relational Algebra

In a relational model, the Entity Relationship Model is only a high-level view. In order to interact with the actual data, we need Relational Algebra.

Query languages allow manipulation ad retrieval of this data. These are not programming languages, but rather they are just used for access to large data sets. SQL uses relational algebra. In order to understand SQL (or any QL) we need to first understand relational

algebra.

Every RA consists of a set of 1 or 2-ary operators that operate on relations. They output a relation, which means you can have nesting of them. For example if x is your operator, and a, b are your relations, you could have $x(a, x(a, b))$ to give some new relation.

6.1 Operators

Unary operators:

- σ select some tuples from a relation.
- $\prod$ get just a few attributes from a relation
- ρ Rename relations or attributes.

Binary operators:

- $\times$ Combine two relations
- $\bowtie$ (Join) Combination of cross product and select
- $\cap$ Intersection
- $\cup$ Union
- $-$ Difference

6.1.1 Projection

Suppose we do $\prod_L(R)$. We will get a subset of the attributes of the input relation, R. L denotes the list of attributes we want. This operation also eliminates duplicates. (Real languages don't actually do this unless specified, since its very expensive to do so).

6.1.2 Selection

$\sigma_C(R)$ would give a subset of all the rows of the relation which match a condition C. There are no duplicates in the output, and the condition must involve some attributes of the relation.

6.1.3 Cross-Product

The cross product: $R_1 \times R_2$ pairs rows of R_1 and R_2. This is honestly best shown by example.

```
A = {a, b, c}
B = {d, e}

A X B = {ad, ae, bd, be, cd ,ce}
```

So each element of A is paired with each attribute of B, forming new entries containing attributes of both A and B.

If the attribute names are the same, we simply rename one of them by pre-pending the name of the table. For example, the attribute *name* in the table *People* would become *People.name*

6.1.4 Join

A join operation $R_1 \bowtie R_2$ is a convenience operator which combines a select and a cross product. That is:

$$R_1 \bowtie R_2 = \sigma_C(R_1 \times R_2)$$

Equi-join is a special case of join where the condition is equality on one or many attributes. We simply do a cross product, and keep all rows where the specified attributes from A are the same as the specified attributes from B. The "equal" attributes are not duplicated in the output. For example, if you're keeping all rows where *name* is equal in $Table1$ and $Table2$, we don't keep two columns $Table1.name$ and $Table2.name$ we would only pick *name*.

Natural join is an equi-join on all attributes which have the same name.

6.1.5 Renaming

Renaming: $\rho(R_{out}(B_1, ..., B_n), R_{in}(A_1, ..., A_n))$ where A_i and B_i are attributes. The output relation is R_{out}. This operation renames all A_i to B_i.

Renaming is useful when you want to do a join on the same relation. For example, you have a table for $Team$ and you want to find all the members of $Team$ of the same height. You could rename $Team$ to $Team1$ and then do an equijoin on the two tables. Although, we need to be careful to make sure that the names do not match so that we don't match the same entry with itself.

Part III
SQL

SQL is used as a standard to define DDL and DML. There are many dialects of it in various DB's.

7 IMPORTANT

I more or less skipped this section (SQL and Application programming) since there isn't much value I can add other than just rewriting the syntax and commands which are already on the slides. If, at a later date I have time to go through and find little nuggets of information, I would add them in this part of the guide.

8 Creating a Table

Very similar to DDL, just:

```
CREATE TABLE students
   (sid INTEGER,
    name VARCHAR(30),
    login VARCHAR(30),
    faculty VARCHAR(30),
    major VARCHAR(30) DEFAULT 'undefined')
```

Note that the types are specified, it is case insensitive (but uppercase on commands is convention) and we can define default values.

9 Simple Queries

9.1 Insert

```
INSERT INTO Students
   VALUES (53666, 'Bartoli', 'bartoli@cs',
          'Science', 'Software Engineering')
```

Note that the order of the values is the same as the order of our CREATE TABLE statement. If we want to insert only a subset of the values, we must specify which attributes are inserting. The rest will be NULL or default.

```
INSERT INTO Students (sid, name, faculty)
VALUES (53668, 'Chang', 'Eng')
```

9.2 Delete

Similar syntax to Insert, but we need to specify a key on which to delete entries.

```
DELETE
FROM Students
WHERE name = 'Chang'
```

This will delete all entries with the name as Chang. Any attribute can be used, or even a combination of attributes.

9.3 Update

```
UPDATE Students
SET major = 'Software Engineering'
WHERE sid = 53688
```

This will update the major of the entry with sid = 53688. We can update multiple columns at once as well the same way we inserted multiple values.

10 Advanced Queries

11 Constraints

We can use NOT NULL when defining the table to ensure that an attribute is non-empty.

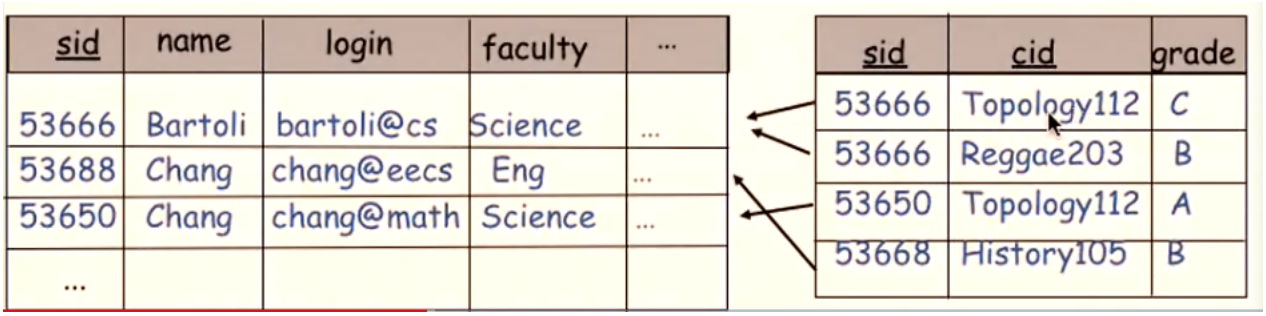
11.1 Primary Key Constraints

A set of fields is a **key candidate** if this set is unique in the table, and it is not true for any subset of the key set. Ie if there is a simpler unique key set, you must use it. If there are more than one elements in the set, then one is chosen to be a **primary key**.

Often, this primary key is required to be NOT NULL.

11.2 Foreign Key

A foreign key is a set of attributes from one relation that is used to refer to a tuple in another relation.



sid	name	login	faculty	...
53666	Bartoli	bartoli@cs	Science	...
53688	Chang	chang@eecs	Eng	...
53650	Chang	chang@math	Science	...
...				

sid	cid	grade
53666	Topology112	C
53666	Reggae203	B
53650	Topology112	A
53668	History105	B

So here, sid refers to the students table, (foreign key) while the cid is the primary key of the Enrollment table. The foreign key must be the primary key of the other table. However, if we delete a record from the student table, we must first delete all dependent records in the Enrollment table.

```
CREATE TABLE Enrolled
(
   sid CHAR(9),
   cid VARCHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid, cid),
   FOREIGN KEY (sid) REFERENCES Students
)
```

11.3 Joins

As in the relational algebra, we can Join in SQL. INNER JOIN is the default type of join, where doing:

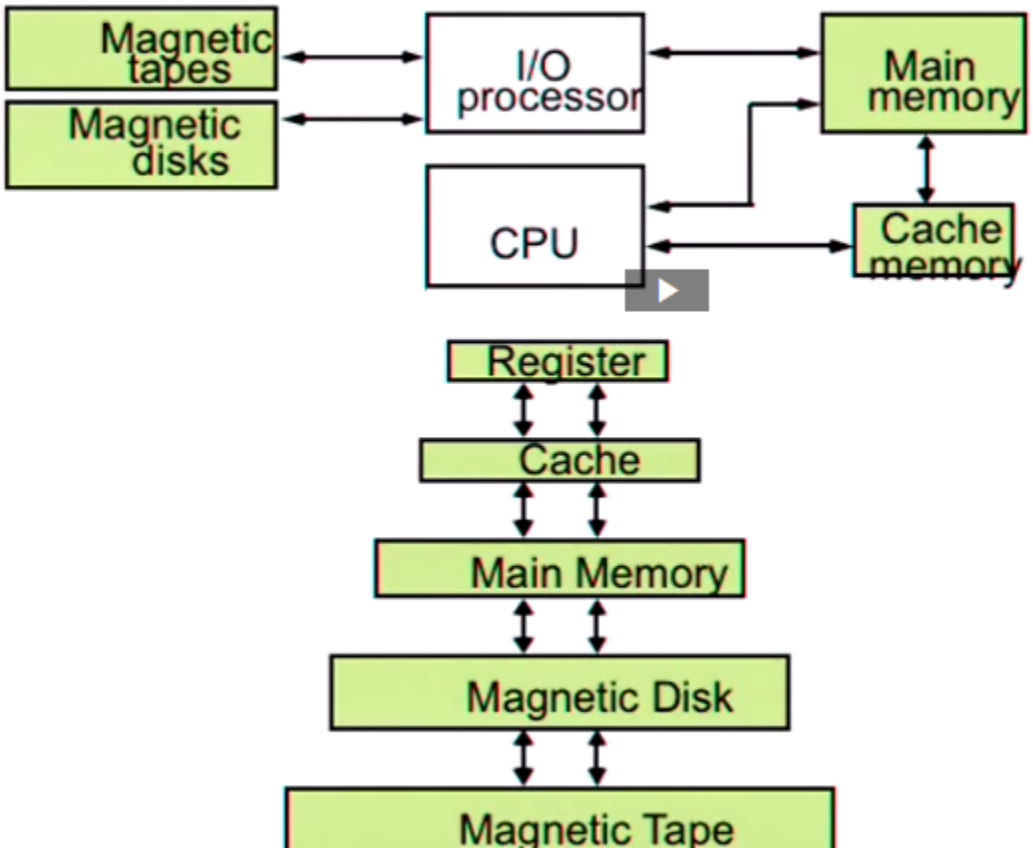
```
SELECT s.sid, s.sname, p.cid, p.rank
FROM skaters s INNER JOIN participates p
   ON s.sid = p.sid;
   ------------------
SELECT s.sid, s.sname, p.cid, p.rank
FROM skaters s JOIN participates p
ON s.sid = p.sid;
    ------------------
SELECT s.sid, s.sname, p.cid, p.ran
FROM skaters s, participates p
WHERE s.sid = p.sid;
```

all give the same result.

Part IV
DB Internals

How does a database actually work? When we execute a query, how does it actually get done and retrieve your information?

12 Memory Hierarchy



In general, the higher in the hierarchy, the faster and more expensive memory becomes (and hence usually smaller because of the cost). The cost tends upward because of how hardware complexity scales as the amount of memory increases.

Also note that the bottom two, magnetic tape and disk allow persistent data.

Another important distinction is the access unit size. We can access individual words at the register level, whereas at the disk and tape level we can only access blocks.
While we don't strictly need to use blocks, its much more efficient to do so because of magnetic media's structure. (Can only read in one place using the reading head).

Normally, we have several volatile buffers living in main memory called frames (see my

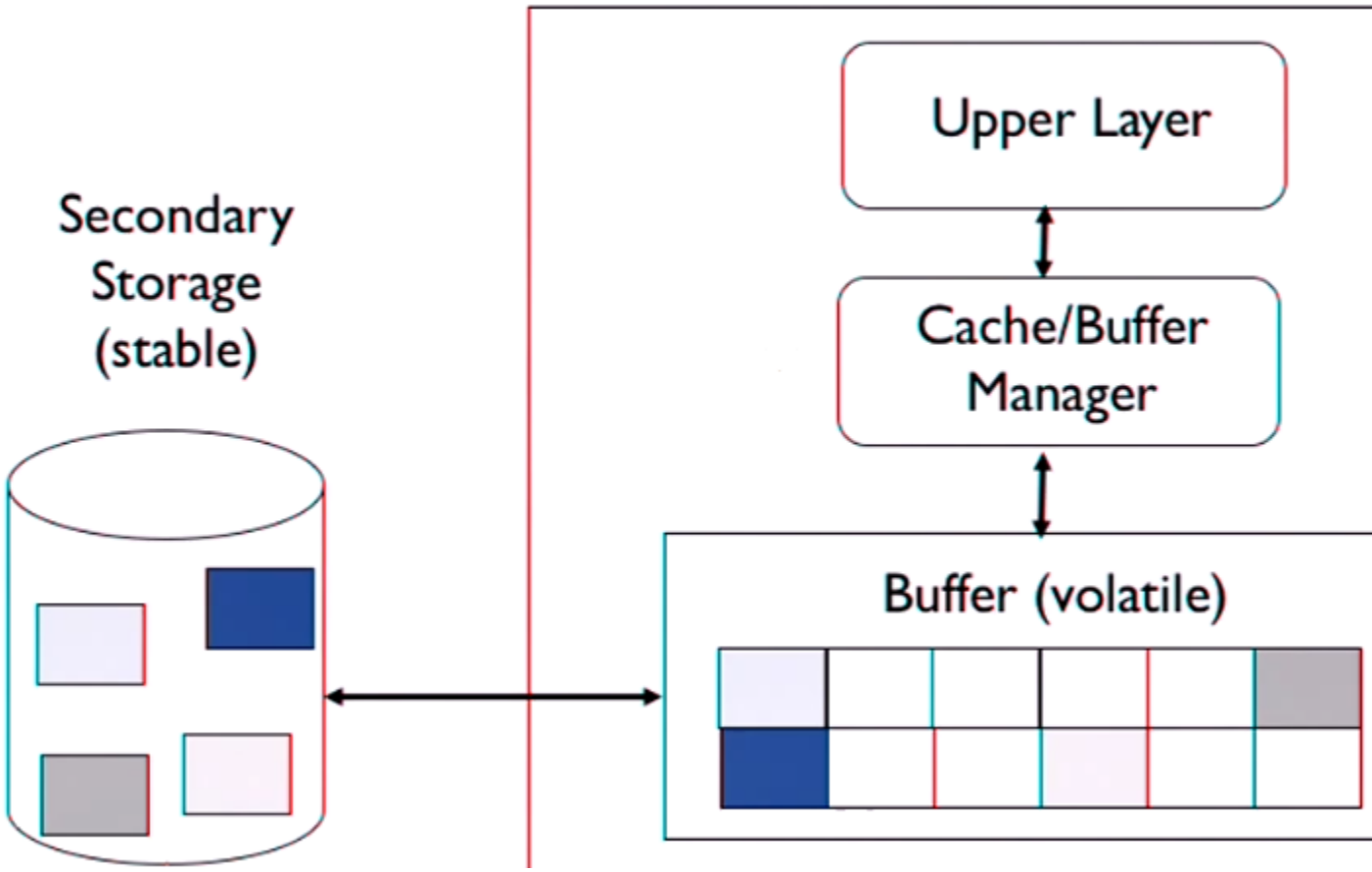
COMP310 study guide for more on that) and a large permanent storage.

Sometimes this permanent storage is solid state drives (fastest), hard drives, or tape (slowest). Some systems use a hybrid of all 3, such as solid state drives for fast data access, hard drives for less commonly accessed data, and tape for backups.

There is also a trend toward using small in-memory databases for ultra fast access. To avoid data loss in the event of an outage, it writes to disk in linear fashion, rather than actually updating values on disk (since the latter is too slow). It is faster since appending rather than seeking reduces the time of the disk head seeking and finding the data. This allows for a retrieval to be run later.

13 Buffer Management

We will assume an architecture that looks something like this:



The buffer manager keeps track of page requests, and which page is in which frame. It then serves back offsets which allow access into the page data. It lives in main memory.

To access a page from disk:

```
if page not in buffer pool:
   if there is an empty frame:
      use it
   else:
      choose a frame to be replaced
      if frame was modified (dirty), write it to the disk
      otherwise, just kick it out
   load page into selected frame
```

Note that when getting rid of a frame we need to make sure it's not being used. To do this, a **page pin** is used. Whenever an application requests a page, a pin is placed on the frame, and whenever the application is done, it removes the pin. Only when the pin count is 0 can a frame be removed from main memory.

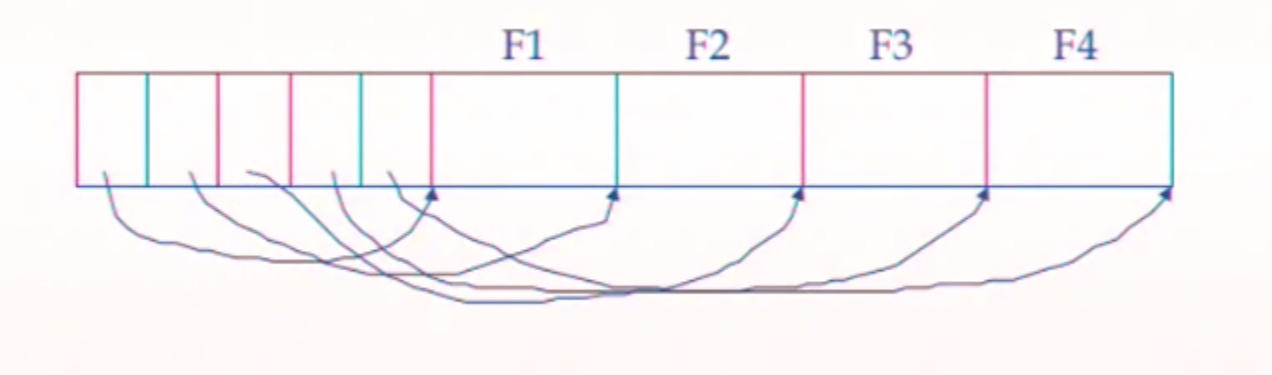
Why not just leave all this to the OS?

- We need our DB to be portable across multiple OS.
- File size limits
- need to pin pages
- need to force pages to the disk
- Adjust the replacement policy
- Pre-fetch pages

14 Record Format

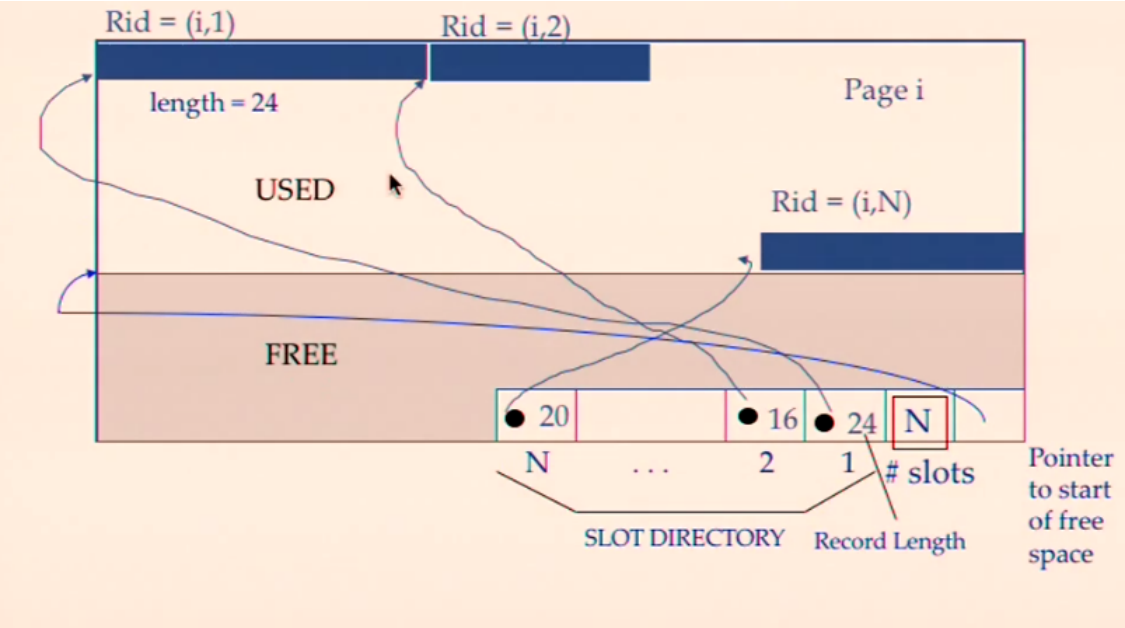
If we had fixed length records and fields, then we could easily figure out the start and end of each attribute we might want to read. This doesn't work well in practice since we waste a lot of space with padding. When we have large records like this, we might take up more pages in the page table, which increases the amount of I/O we need to do which slows things down greatly.

If we instead have a **variable length format** we can fix these issues.



At the beginning of the record, we have pointers to where to find each field. This allows the actual fields to be whatever length we want, thus not losing too much space. A bonus perk from this is that NULLs then take up only one pointer size, rather than a full entry.

If we do this, we need to change tha page format to include a pointer to the first unused slot entry, a counter to the number of used slots, and a slot directory keeping track of where each entry is and how big it is.



One advantage is that the record ID need not change even if the record is moved. Instead, at the old location there will be another record ID showing where to find the entry now.

15 Relations as Files

A file is a collection of pages, and each page holds information about a relation.

We need some way to organize these pages within the file for optimal access.

One way would be to use an **unordered (heap)** file. Any data can be placed in any page. To accomplish this, we need a header page which keeps track of two linked lists. One list of full pages, and another with pages containing free space.

We can also have **sorted files**. Each page will have a pointer to the next, and previous as well as it's own data.

16 Indexing

What is the cost of execution? There are several possible metrics:

- Number of reads/writes
- CPU usage
- Network cost

For this course we only really worry about reads/writes (IO cost) since it is much larger than the others. We will actually only consider read costs since write costs are much more complex.

16.1 Typical operations

```
--Scan over all records
SELECT * FROM Students;
--Point Query: equality condition on the primary key
SELECT * FROM Students WHERE sid = 100;
--Equality Query: equality condition on something other than primary key
SELECT * FROM Students WHERE starty = 2015;
--Range Search
SELECT * FROM Students
WHERE starty > 2012 AND starty <=2014
```

How would this work on an unsorted filesystem? (Heap files). How many pages would you need to read to do a SELECT *? Well, all of them since we are reading everything.

How many pages for a point query? We still need to read all the pages potentially, since we don't know where the record will be located. But we know that the key is unique so we need not continue reading. So on average, we have to read $n/2$ pages.