# COMP 421 Study guide

Francis Piché

April 2, 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.

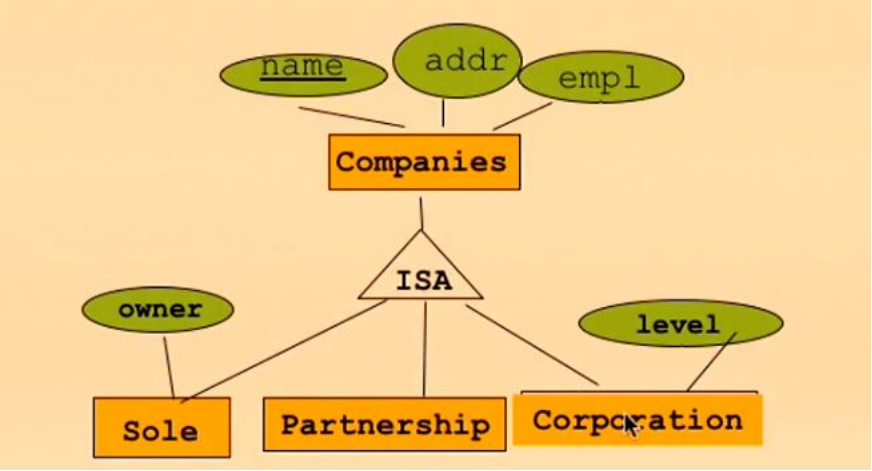
Things like the students, addresses, fees, courses etc are all relevant information. We would also need to think about the types of operations that must be possible by our database. Things like adding students, changing addresses, assessing fees etc.

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).

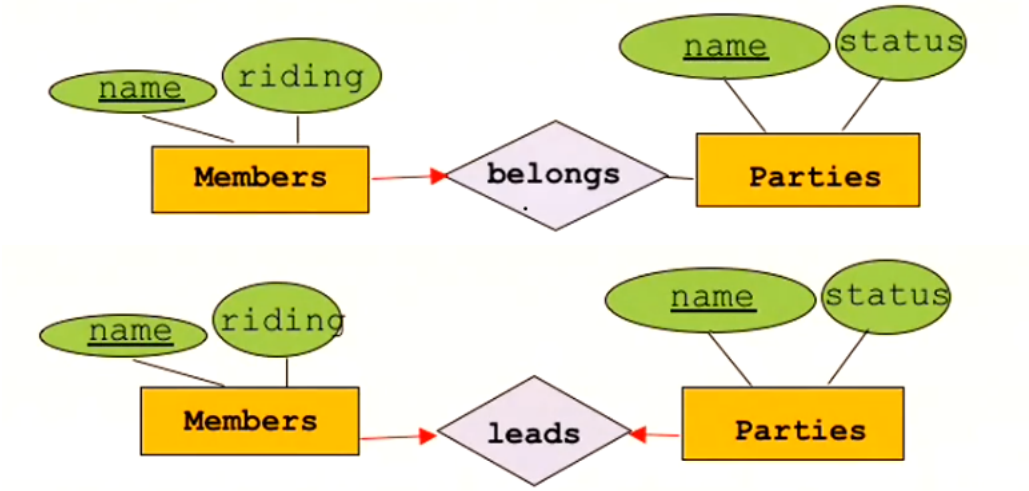
If there are overlaps (a company can be both a partnership and Corporation), we must put a note explaining the situation. We assume that the parent is not covering all possible subtypes. (There may exist a company which is not a Sole, Partnership or Corporation).

## 3.3 Relationships

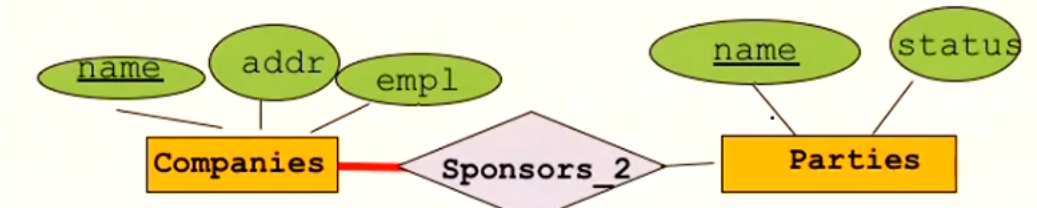
A relationship is an association among two or more entities. This can be one to one, many-to-one, many-to-many. A relationship set is a collection of similar relationships.

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.



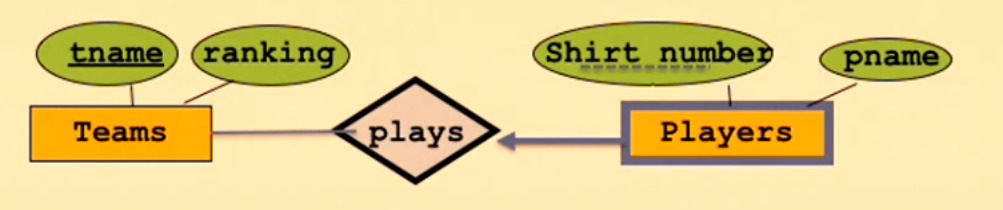

which is shown by a bold line. This can be combined with a key constraint to show a relationship in which exactly one entity set is related.

**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.

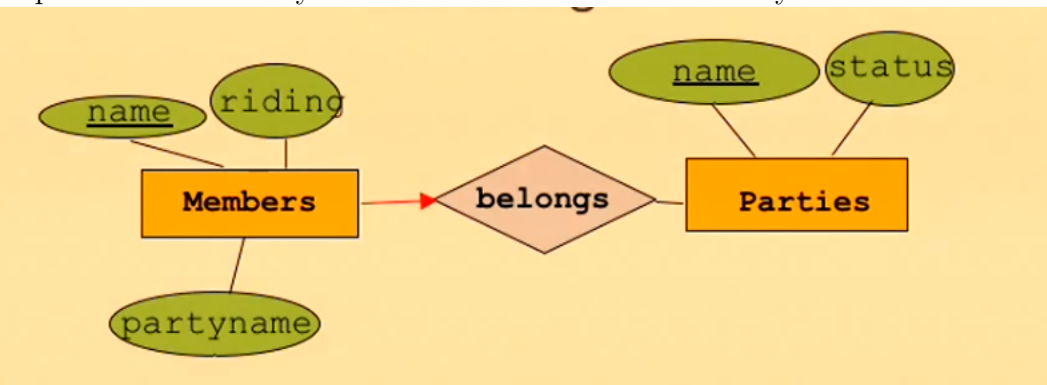



We often create artificial keys to keep track of information. For example student numbers. However, you should never use an artificial key as a partial key. If you're going to make one, make sure it's unique.

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.

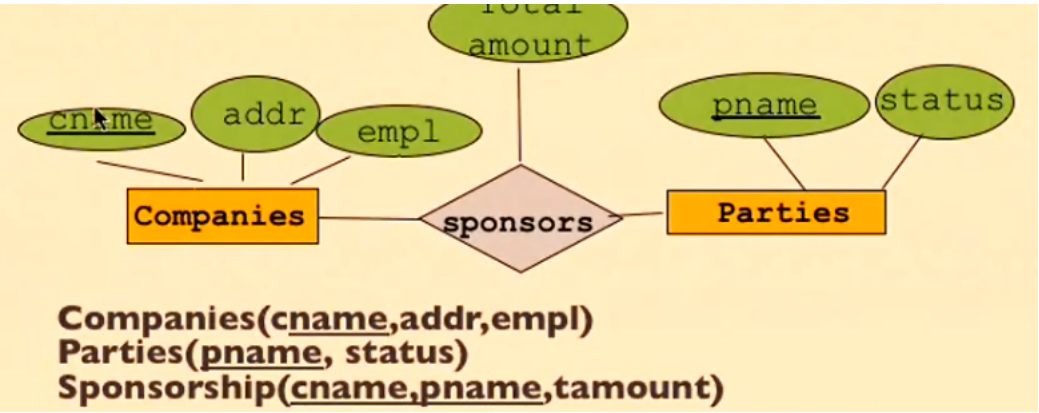
# 5 Translating from ER to DDL

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

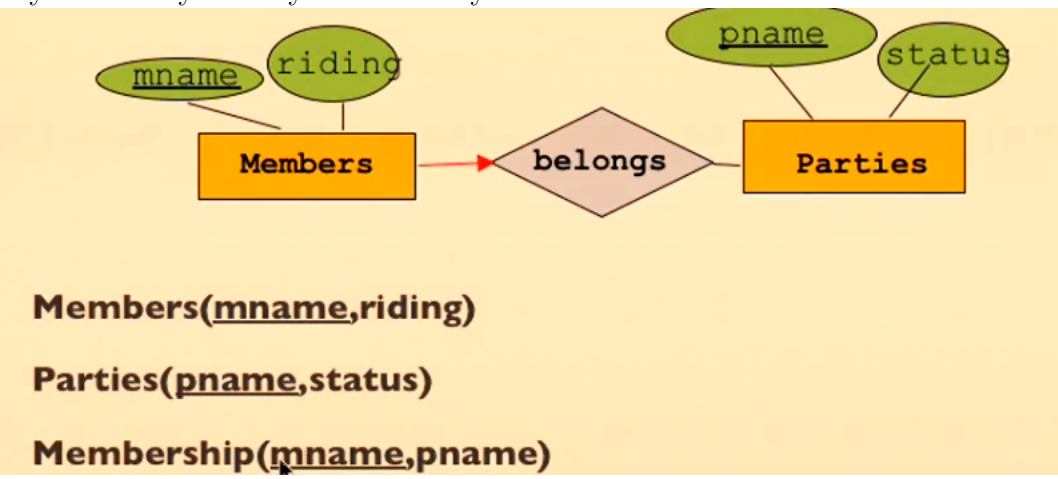
## 5.1 Relationships

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:

- $\sigma$ select some tuples from a relation.
- $\prod$ get just a few attributes from a relation
- $\rho$ 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.

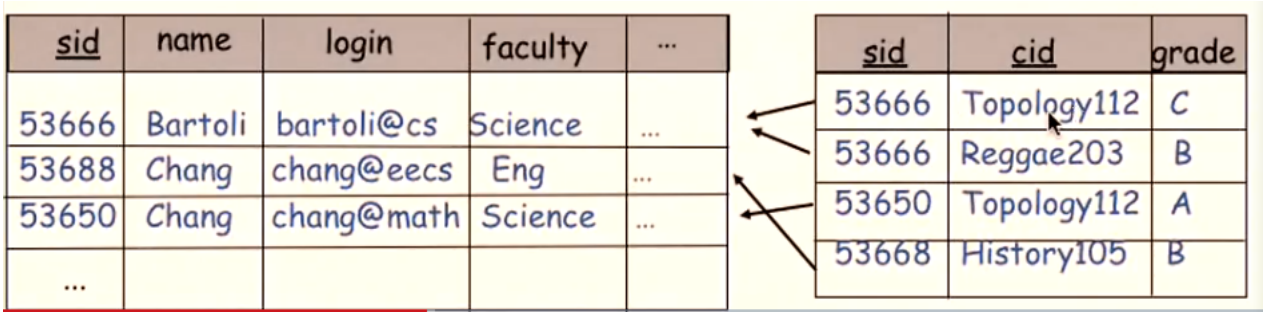
### 11.1 Primary Key Constraints

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:

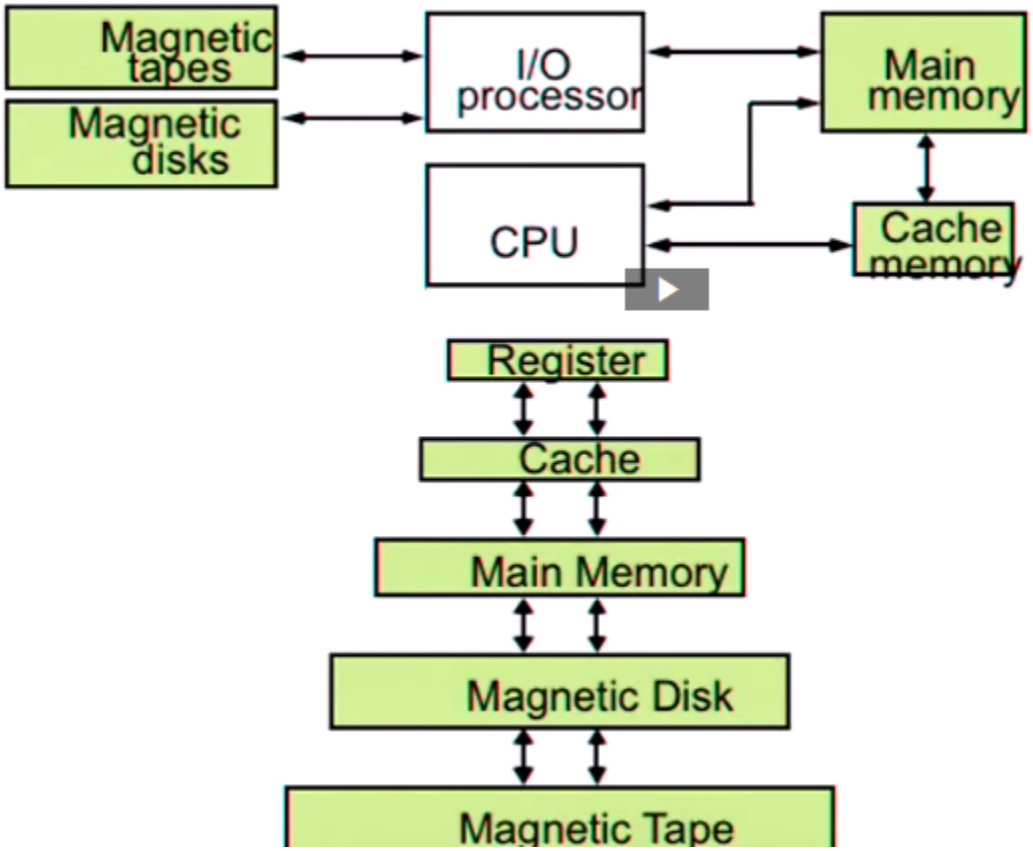
```
SELECT s.sid, s.sname, p.cid, p.rank
FROM skaters s INNER JOIN participates p
   ON s.sid = p.sid;
   ------------------
SELECT s.sid, s.sname, p.cid, p.rank
FROM skaters s JOIN participates p
ON s.sid = p.sid;
    ------------------
SELECT s.sid, s.sname, p.cid, p.ran
FROM skaters s, participates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

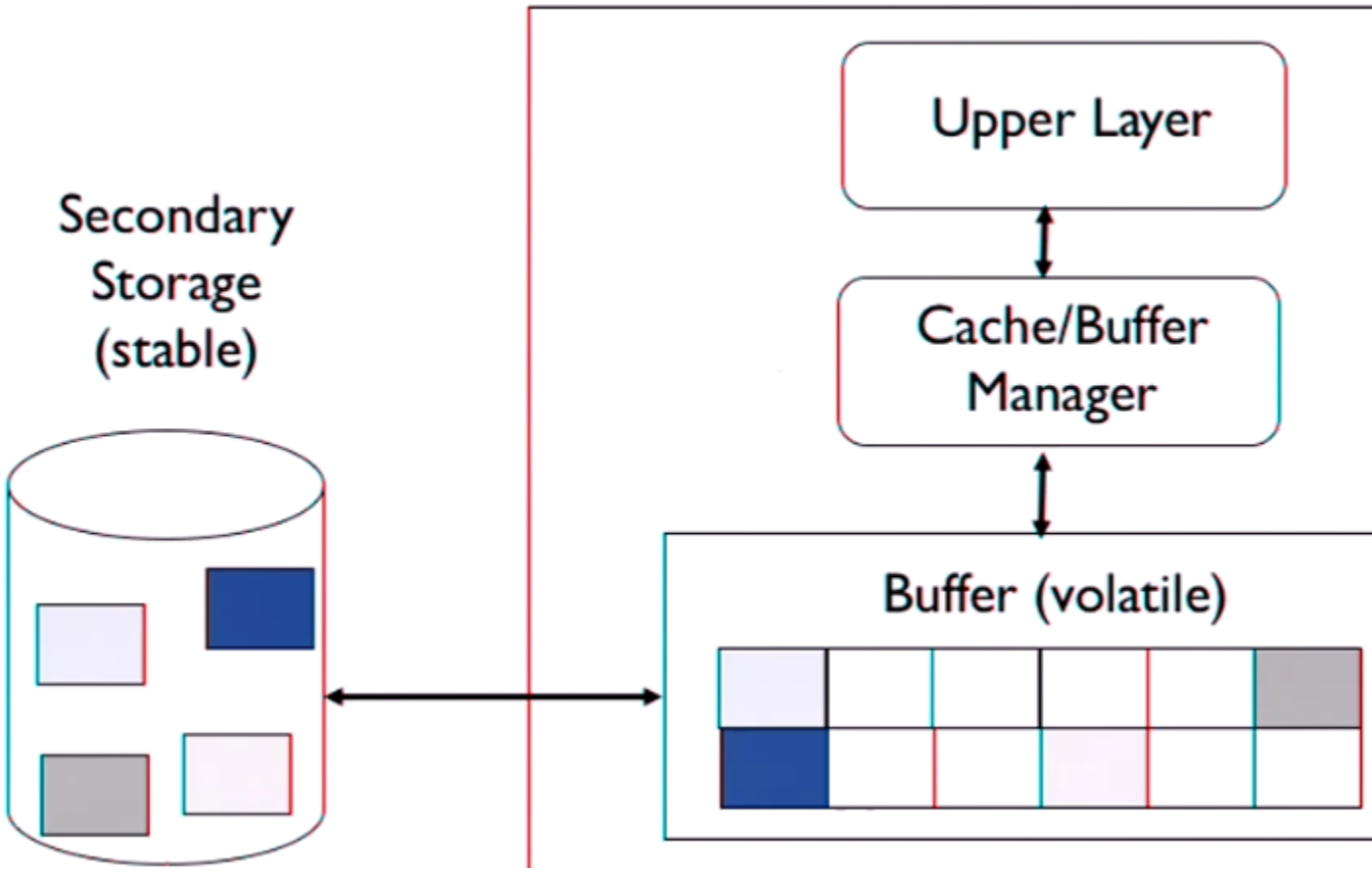
COMP310 study guide for more on that) and a large permanent storage.

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.

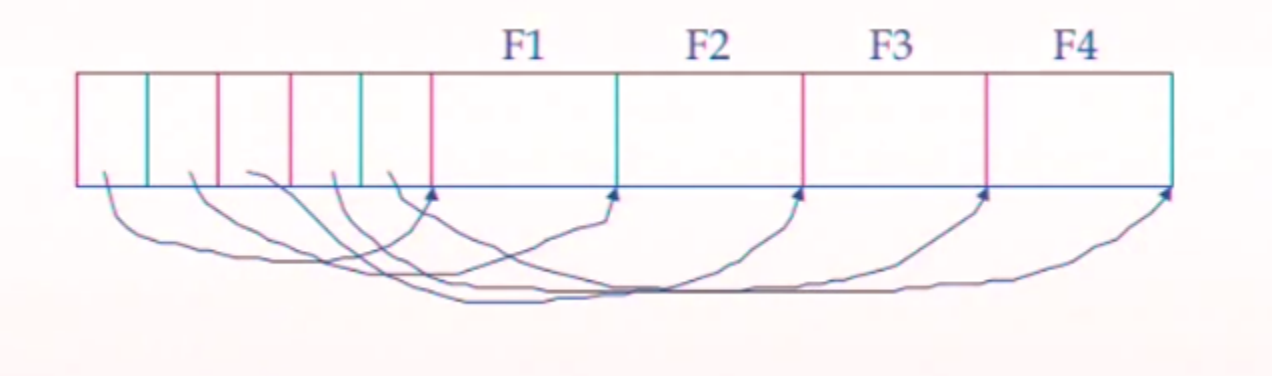
Why not just leave all this to the OS?

- 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

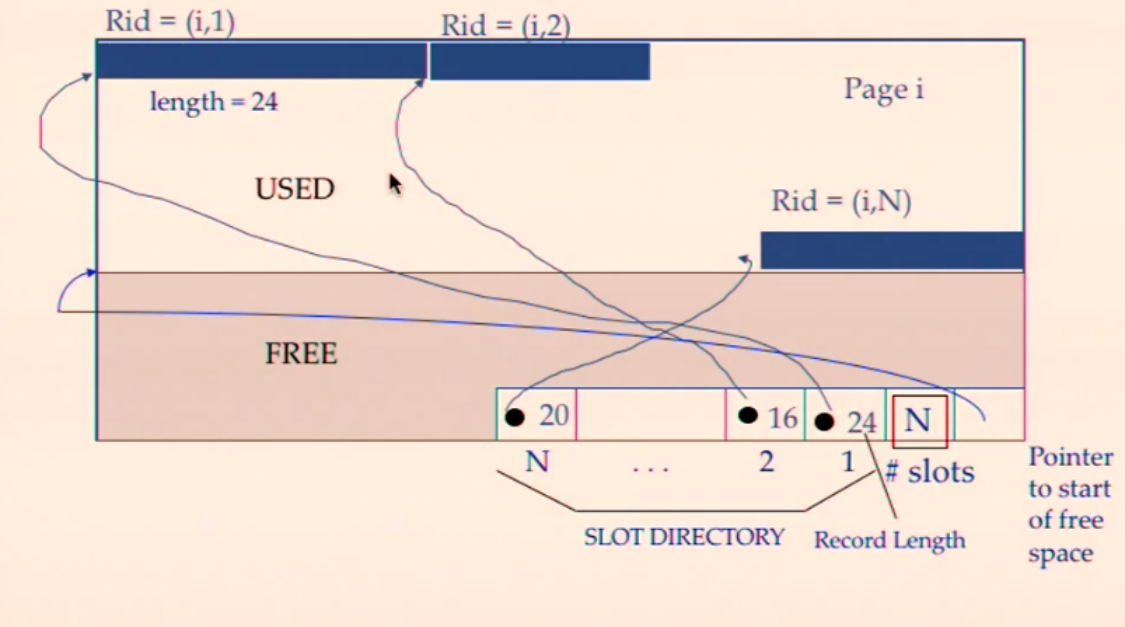

If we had fixed length records and fields, then we could easily figure out the start and end of each attribute we might want to read. This doesn't work well in practice since we waste a lot of space with padding. When we have large records like this, we might take up more pages in the page table, which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

### 16.1.1 Unsorted Filesystems

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.

A range query again requires reading all the pages since we do not know anything about how many entries we are looking for. Once we find an entry, we need to keep going to be sure there aren't more in that range.

Doing an insert we only need to read one page since we can just insert into the first page with free space.

### 16.1.2 Sorted Filesystems

Suppose we have sorted filesystem.

Then `SELECT *` is the same, since we have to grab all pages anyway.

Equality search on some attribute will take $O(log(n))$ since we can now do binary search to find the page. (Since we have a sorted filesystem).

A range search will be $O(N)$ but generally much smaller, since we know that once we are out of the range there will be no more entries in the range.

Insert is not good, since if we have a lot of records, we need to ensure the page has enough space, if not then we need to add to another page, and reorder a lot of data to keep it sorted.

Delete and update are similar to search since we need to find the entry. Update might lead to re-sorting the pages, however.

We get a bonus advantage in that we can easily give outputs that require sorting (ORDER BY).

However, we can only **sort by one attribute**. If we need to find some other attribute, the sorting does not help us.

# 17 Indexes

The solution to the above problem with sorted file systems is the idea of indexes.

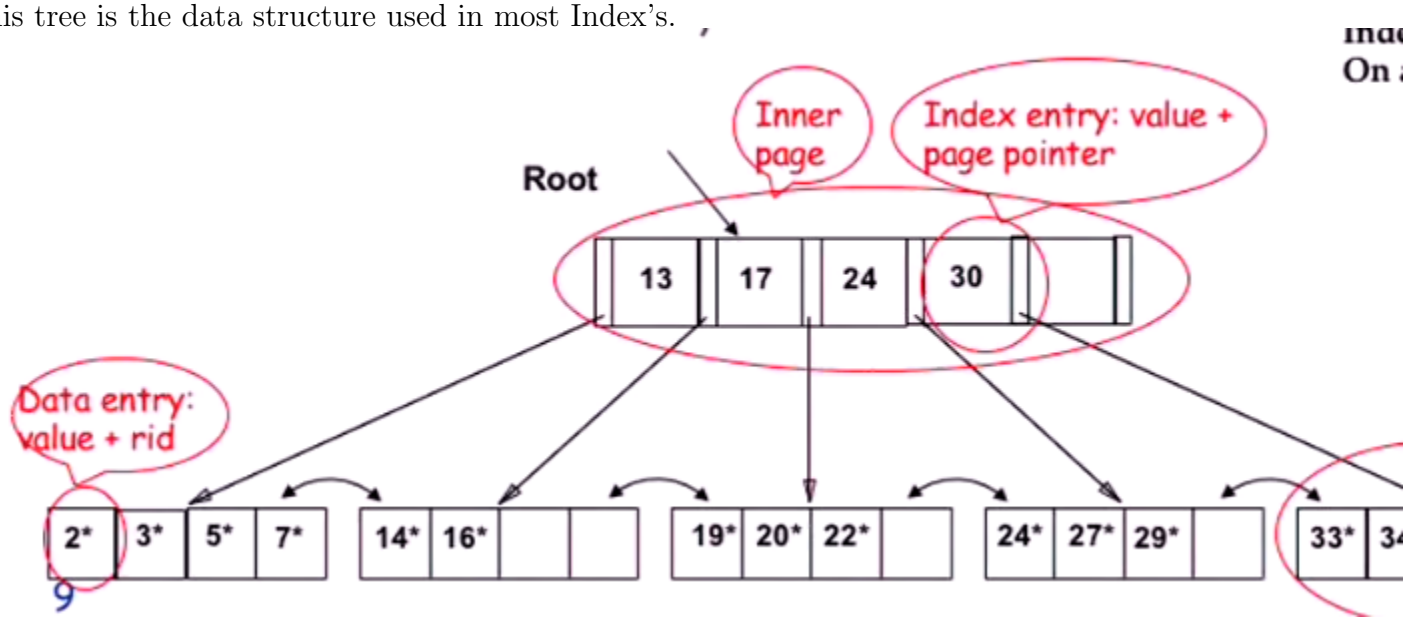
**Indexes** help find attributes faster. A collection of attributes on which an index is built is called the **search key attributes** for the index.

To create an index:

```
CREATE INDEX index ON students(sid);
DROP INDEX index;
```

## 17.1 B+ Tree: The Most Used Index

This tree is the data structure used in most Index's.




There are two types of nodes in this tree. The leaves are pages contain both the values of the entries, and pointers to their record identifiers. Recall that the record identifier is made up of the page id and slot number.
The root and inner nodes have auxiliary index entries. These contain the largest value in the page it points to, and a pointer to that page (page id). These would be smaller in width than the rid's since they do not need to contain a slot number.

So say we are looking for 16. We would read the root node left to right until we see that 16 is less than the value. In this case, 17. Go to that page. Repeat until we get to a leaf. Read left to right until we find the value, then follow the rid pointer to find the entry. Total number of IO is the number of nodes loaded. 1 for the root + k for the depth k of the tree.

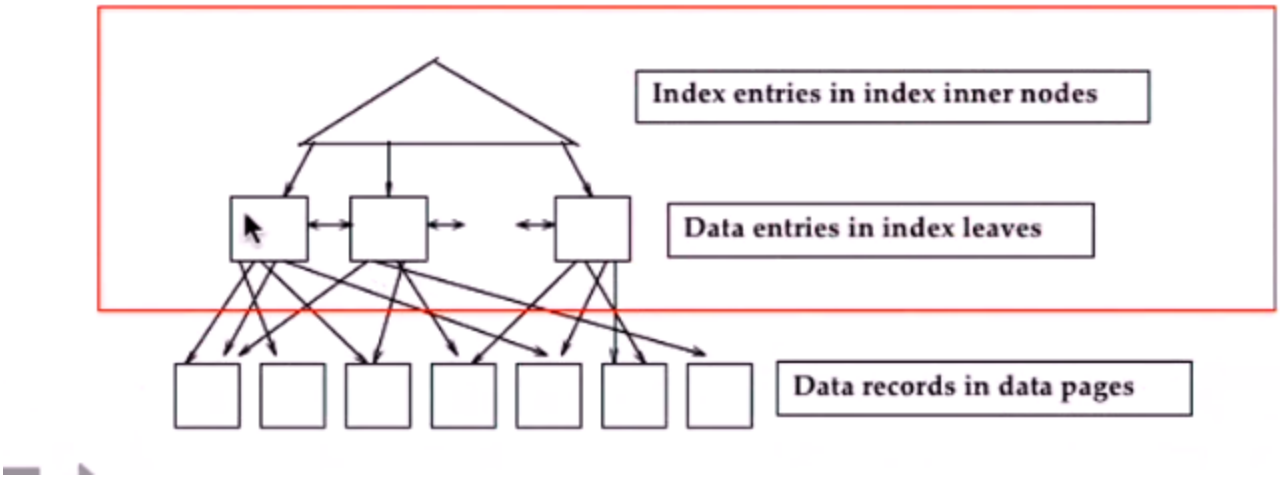
Each leaf node also has a pointer to its siblings to aid with range queries and stuff. If the tree is very tall, we don't want to always start all the way at the root and come down. Sometimes it's more beneficial to cross directly to the next leaf.

Note that all the pages need to be at least 50% full, otherwise its not efficient. The root might be an exception if it doesn't have enough children. The tree is also balanced.

### 17.1.1 Costs

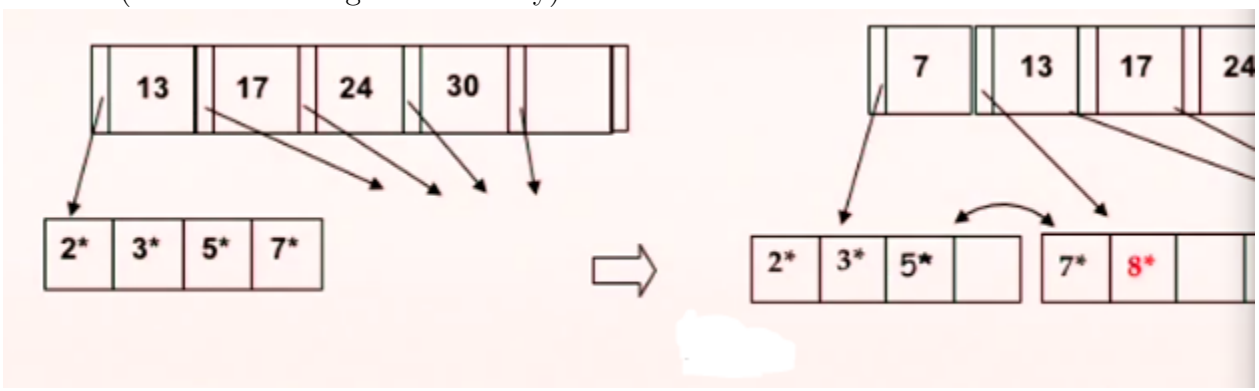
The fanout, $F$ is the number of children for each node. $N$ is the number of leaf pages.

Inserting/deleting has cost $log_F N$ cost.

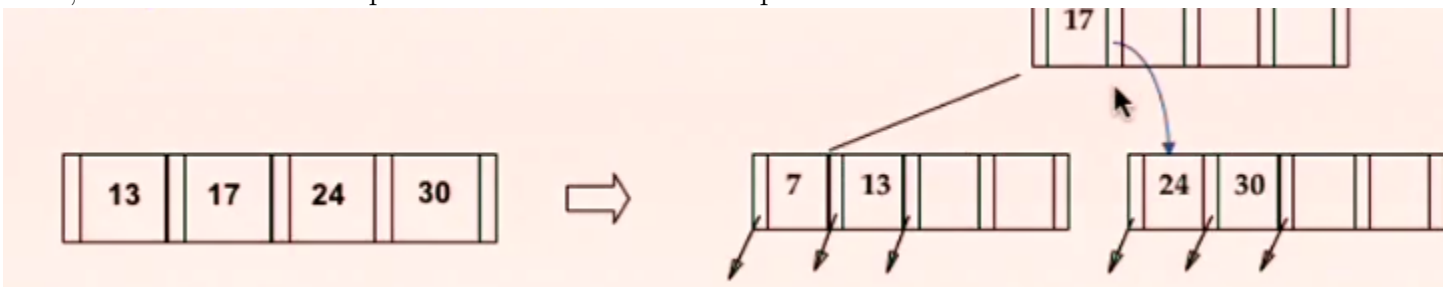



The red section is the index part, the nodes outside are the actual data (requiring their own IO).

To insert, if the leaf node we are trying to insert into is full, we split the leaf into two, placing the second half of the entries into a new leaf node. The new one will now be roughly half full (hence conserving the efficiency).




Next, the index node that pointed to the leaf needs its pointers reset.




Note that due to this splitting, the tree's height grows much slower than the width.

### 17.1.2 Indirect Indexing

What we described above is called indirect indexing because the data itself is not part of the index. (Only the rid pointer is part of the index). There are two ways to do this:

Suppose we were using an index for a non-primary key. Then we may have many repeated leaf nodes. One way of storing this is to just keep storing them over and over.

The smarter way is to keep the value once, and the pointers as a list.

### 17.1.3 Direct Indexing

Here, the data entry is actually in the leaf node. Thus there can only be one direct index per table.

## 17.2 Clustered Indexing

A clustered index is when the data themselves are also sorted on that index. So not only is the index values sorted (occurs in both clustered and unclustered), but the actual data entries are sorted on the same attribute.

So you can create a clustered index only on one attribute in a table.

## 17.3 Multi-Attribute Indexing

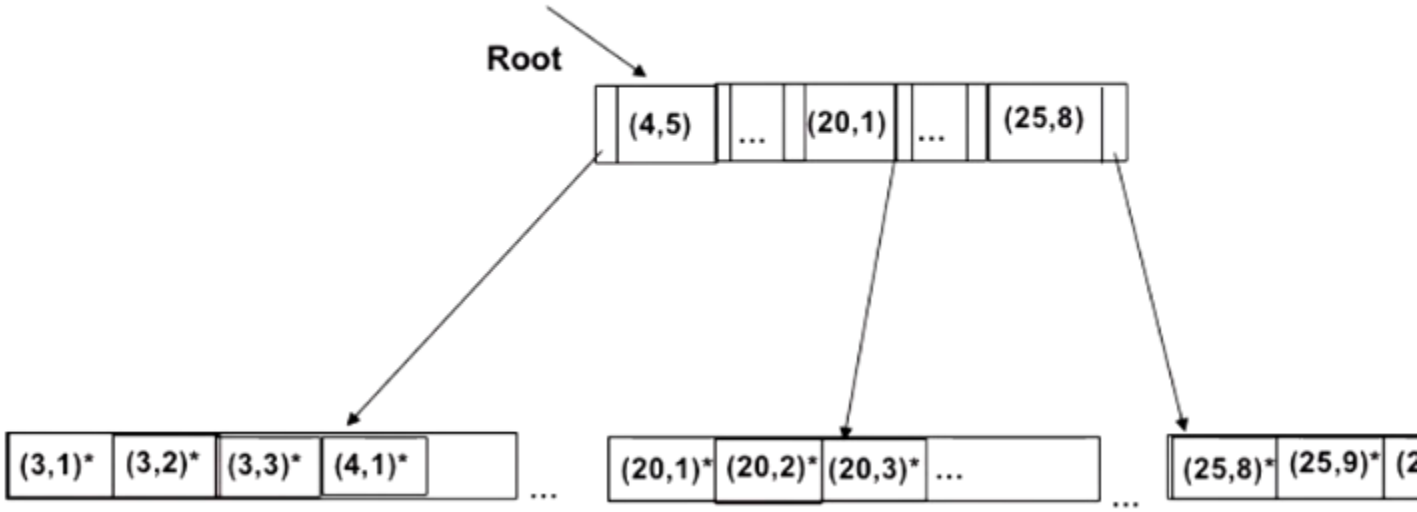
The order in which we create the index is important. When we create an index on two attributes we first sort on the first, then on the second as a tie-breaker.

The leaf pages then look like (x, y)* where x is the first value, y the second and * the pointer to the rid.

These take up a bit more space because we can have: `(x, 1)*, (x, 2)*, (x, 3)*` the same first value multiple times, each one storing a different second value and rid. Then there can be less total entries.

With a multi-attrib index you can do something like `SELECT* FROM Skaters WHERE age = 20 AND rating < 5` very well, since you could find age=20 on the index easily, and then the range query is fast since you can just read to the right since the leaves will be ordered by (20, rating).

We cannot do something like `SELECT * FROM Skaters WHERE rating < 5` where rating is the second index attrib, since they will be scattered all over the leaves of the tree.




# 18 Static Hashing

An alternative to B+ tree indexing is to use a hashtable but with the pages as the unit of storage. (As opposed to an array of pointers).

This is useful for equality queries but not so much for range queries since hashtables are not sorted.

# 19 Query Evaluation

Processing an SQL query starts with a **parser** which translates it into an internal expression. This will check for syntax, existence of relations/attributes, and replace any views by their definitions.

The **query optimizer** will turn the query into some efficient plan.

The **plan executor** executes the execution plan.

## 19.1 Query Decomposition

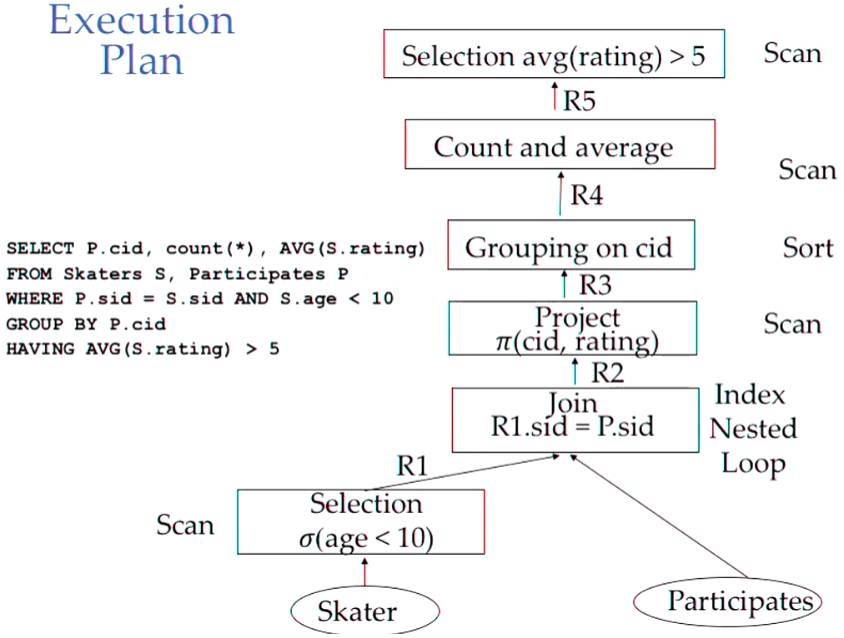
Assume the parser has already run. Then the query is broken down into basic operators. The most efficient order in which to compute these operators needs to be determined. This decision depends on amount of data, resources available etc.

## 19.2 Terminology

- **Access path**: The method used to get a set of tuples from a relation. This includes scanning the whole file, using an index, or partitioning etc.
- **Cost Model**: A metric used to compare different execution paths. Needs to know the query, database stats, resource availability. We will use the number of pages retrieved from disk as our cost model. Assuming that the root and intermediates are already in memory.

## 19.3 Concatenation of Operators

We build an execution tree in which the leaf nodes are the base relations, and inner nodes are the operators. Execution would begin at the bottom of the tree and work up to the root.




Here, each operation is pipelined. Meaning when one operator finishes on an entry (say SELECT) it will immediately send the result to the next operator. This allows for a high level of concurrency.

The SELECT on AGE $>$ 10 is done first, since then there will be less entries to deal with for the other operations. Then the join is done, followed by the projection to get rid of unnecessary columns. Grouping is done by sorting since using in-place quicksort we can do it without using extra memory. This also halts the pipelining since the sort must complete before the count can start. Count and average is done, and finally a selection on the average.

## 19.4 Reduction Factor

The reduction factor is the ratio of number of records which would be outputted on applying a condition to the total number of records. ie:

$$Red(\sigma_{condition}(R)) = \frac{|\sigma_{condition}(R)|}{|R|}$$

### 19.4.1 Example

Suppose we have: `Users(uid: int, uname: string, exp: int, age: int)`
`Groupmembers(uid: int, gid: int, stars: int)`
With:
For the Users table:
40,000 tuples, 80 per page, 500 pages, an index on uid having 170 leaf pages, and an index on uname with 300 leaf pages.
For Groupmembers:
100,000 tuples, 100 per page, and a total of 1000 pages.

The database must keep track of the size of the tables, number of pages, indexes and how big indexes are etc. It also must keep track of the rough domain of data values, and how they are distributed.

Then the reduction factor of selecting on the condition experience = 5, would be 0.25 assuming 10,000 users have experience of 5.

If the database didn't know there were 10,000 users with exp 5, it might know there are 10 different experience levels, and assume a uniform distribution, then the reduction factor would be $1/10 = 0.1$

Suppose we select on age $\leq 16$, and the database knows the minimum and maximum age:

$$\frac{(16 - min(age) + 1)}{max(age) - min(age) + 1}$$

Can we combine these conditions? We have the reduction factor for each, so assuming the distributions are independent, then we can just multiply them together. However, often in real world data there is a dependence, so be careful!

### 19.4.2 Why Reduction Factor Is Used

If the query has multiple options for what to process first, often we would compute the reduction factor, multiply by the number of input tuples to figure out how many tuples are in the output. Then you would choose the query with the lease tuples in the output, since this reduces the amount of work needed on subsequent queries.

## 19.5 Simple Selections

If we have just a basic SELECT * FROM bla WHERE uid = 123
Assuming no index, if we search on some arbitrary attribute, the cost would be the total amount of pages in the relation, whereas if we search on the primary key, then we cut the cost in half, since on average we would find it within $n/2$ reads.

If we had an index, then the cost would depend on how the index is set up, and the number of tuples which match the condition.

If we have a clustered B+ tree, then both the leaves and the data pages are sorted on the same attribute. We find the path from the root to the leftmost leaf. Then, since its a clustered index we can keep reading to the right until the condition no longer matches. So assuming each data page holds 80 tuples, and there are 100 matching tuples, we need to read 2 data pages, plus the leaf index page.

If we had an unclustered b+ tree, we'd have to read the first leaf page qualifying, and again go fetch all the data pages. However, some pages may be loaded twice. This is because, say the 3rd data page is attempted to be loaded, but memory is full. Then say it kicks out the 1st data page. Since the data pages are unordered, the query might still need to access the 1st page again, and then load it back into memory once again.

Suppose roughly 25% of our tuples match. Then its likely that every data page has a tuple somewhere in it. However, each retrieval might cost an I/O since the page might have been purged in between. So then we would read a number of pages equivalent to 25% of the number of tuples. If thats a large number, say 10,000, then we are reading 10,000 + 25% the number of leaf pages! Then a simple scan would have been faster. So using an index here would be a bad idea. Normally you only use an index for high reduction factor queries.

## 19.6 External Sorting

Sorting in databases is useful in a few situations. Maybe the user specifically asked for an ORDER BY. Or maybe we need to remove duplicates, in which case sorting makes it quite easy to do so. Other times, sorting can be beneficial for subsequent join operations.

Suppose we need to sort a large amount of data, such that it does not fit into memory. How can we handle this?

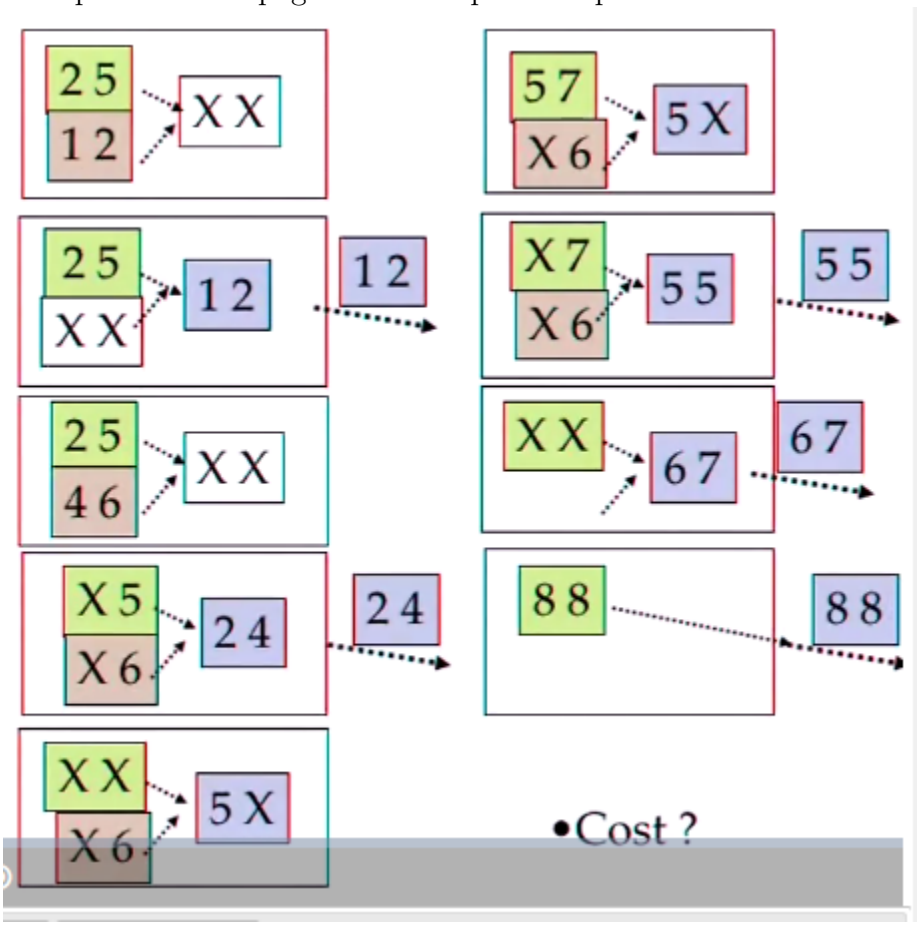
We accomplish this with the idea of *passes*. A pass is a "step" of the sorting.

Suppose we have $B$ buffer frames available, and there are $N$ pages to sort. Pass 0 will bring B pages into the buffer frames, sort them, and write a copy of the original data to the disk. (in a temporary storage). This is called a *run* of pass 0. The second run will bring the next B pages into memory, sort them, and write to temporary storage. This repeats until all N pages have been processed.

We are not done! We have $N/B$ runs completed in the first pass. This means we have $N/B$ chunks of memory existing in the temporary file, each sorted, but overall we have not sorted.

The cost of this pass is simply twice the number of pages, $2N$ since we read each page, and wrote each page back again.

For subsequent passes, we perform a merge-sort. That is, we take one of our buffer frames to be the "output frame". Then, with the remaining B-1 frames, we load the first page from each run of the previous pass. We then determine which is the smallest, write it to the output frame, and then the output frame is written out to the disk. We proceed until we have processed all pages from the previous pass.




The cost of each pass is again 2N since each page is read and written once.

If $B - 1 < N/B$, then we would more passes to finish the sorting.

The algorithm completes when a pass only executes one run.

An important note about the cost: the cost is logarithmic in the number of buffers used. This is because the cost is given by: $2N * (passes)$, where $passes = 1 + \lceil log_{B-1}(\lceil N/B \rceil) \rceil$. So this is a case where the amount of memory we have available actually causes a speedup in our algorithm.

## 19.7 Sorting with other operators

Depending on which other operators are involved in the query, we may want to employ some tricks to reduce the overall cost.

For example, if we are doing an ORDER by and a Projection, we do not need to write the result of the sorting to the disk, we can pass it directly to the projection operator (provided we only need pass 0).

If there is multiple passes required, can only skip the write for the final pass. (since we just give directly to the next operator).

We can further optimize by applying the projection after pass 0, since then we can fit more pages into the buffers, and reduce the number of passes required!

In general, projection is pushed to the lowest level possible since it reduces the size of the processing.

## 19.8 Joins

How do joins actually work?

### 19.8.1 Join Cardinality Estimation

To analyze different join algorithms, we need to be able to estimate how many records will be output by a join.

$|A \bowtie B| =?$
If the join attribute is a primary key for A, then each B tuple will match with exactly one A tuple. so the result here will be $|B|$.

How about something like:

$$|A \bowtie \sigma_{B.attrib>n} B| =?$$

Well the number of records in this output is limited by the selection. Then, since the selection is on B, and since each entry of B has exactly one match in A, we have $= |\sigma_{B.attrib>n} B|$ However, if there is no foreign key to rely on, for example if we did $\sigma_{A.attrib>n}(A)$, then we would have to compute the reduction factor and thus the number of result tuples would be

$Red(\sigma_{A.attrib>n}(A)) * |B|$.

### 19.8.2 Simple Nested Loop Join

This is the simplest (yet most of the time most inefficient) algorithm for executing a join.

```
for a in A:
   for b in B:
      if a.key == b.key:
         output <a, b>
```

Simply loop through all possible combinations and output them if the matching condition passes.

So the IO cost of this would be:

$$Pages(A) + |A| * Pages(B)$$

since we have to load all the pages of A, and for each record of A, we need to load all the pages of B to be able to combine all the records of B.

### 19.8.3 Page Nested Loop Join

The idea behind this algorithm is to block the matching by page, so that we do not have so many I/O's.

```
for p_a in Pages(A):
   for p_b in Pages(B):
      for a in p_a:
         for b in p_b:
            if match:
               output <a, b>
```

The main savings here come from the fact that we are not loading each page of B for each record of A, we instead or loading each page of B for each page of A, which is a large reduction.

$$Pages(A) + Pages(A) * Pages(B)$$

### 19.8.4 Block Nested Loop Join

We can take the idea of the previous algorithm one step further by joining blocks of multiple pages in the outer relation at once.

So for $A \bowtie B$ we would break up $Pages(A)$ into blocks $bp_a$ of the pages. Then for each block we would perform a join on each page of $B$. The idea here is that we further reduce the number of times we need to read each pages from B.

$$Pages(A) + \frac{Pages(A)}{|bp_a|} * Pages(B)$$

### 19.8.5 Note on Join Efficiency and Table Size

One important thing to notice is that the size of the tables matters in the algorithms. The outer table should always be the smaller table since if you revisit the equations up there, just imagine what would happen if $|A| < |B|$ and we swapped the placements of $A$ and $B$ in the equations.

### 19.8.6 Index Nested Loops Join

Here we use the fact that we have an index to help find matching tuples. This index must be on the joining attribute on the inner table.

```
for a in A:
   find matching tuples in B through index:
      add all <a, b> found to result
```

We still need to read all the pages of the outer table as before, but what about the index table? Well for each entry in the outer relation we do an index lookup in the inner table.

$$Pages(A) + |A| * costoflookupinindex(B)$$

The actual cost of the lookup will depend on whether the index is clustered or not, and on some information about the data.

### 19.8.7 Which To Use

Simple nested and page nested kind of suck.

In general, block nested performs best if the outer relation is able to fit into memory, then the cost would just be $OuterPages + InnerPages$, since then we would have just one block.

The Index nested loop performs well when the cost of reading the index is low, and if the cardinality of the outer is low.

So: Choose Index Nested loop if:

$$InnerPages > |Outer| * matchingTuplesInner$$

otherwise we can choose block nested.

There are cases when the index nested join in better. For example if we have a select that reduces the number of tuples to join by a lot, then it may happen that the cardinality of the outer reduces by a lot.

### 19.8.8 Sort Merge Join

In Sort Merge Join, we use a very different approach. If the tables are already sorted, we can do the join in exactly one pass. This is done by grabbing each entry of the outer table, and we will join with entries from the inner table until we see that there is no match. Then, since the table is sorted we can move to the next entry of the outer table, since we know that there will not ever be a match later on.

So the join part costs $OuterPages + InnerPages$. However, we must first sort the tables. Sorting cost depends on several factors such as how much memory is available, size of tables, pages etc, as seen in the previous subsection.

However, if we put the sort and join together, we can avoid the last step of sorting which is writing to the disk. Similar to what we did with sorting on projections selections etc, we can pipe the final pass of the sort directly to the join and avoid the cost of writing the sort, and reading for the join.

### 19.8.9 Hash Join

To do a hash join, we send the outer table into a hash function. This hash function will use one frame buffer for performing the operation, then it will hash the information into B-1 (B is the number of available buffers) buffers. If one of these buffers becomes full we'll have to write to the disk for the overflow.
Next, do the same thing for the inner table. (Kept separate).

This hashing will separate the tables into partitions.

Now, for the join step, we assume that the number of pages in each partition is less than B-2, since we need 1 for the input and 1 for the output. (Leaving B-2 available). Recall that for identical keys, the hash function will always store in the same partition. So for each table, we can look in the same partition bin to find the matching records!

So if you join partition 1 of table 1 and partition 2 of table 2, you'll get no results since there

can never be identical records in different partitions. So we don't need to join partition 1 with any other partition! So we can treat each partition as a table, and use the previous join algorithms to join each partition with its corresponding partition from the other table.

The cost of this operation would be $hashCost(A) + hashCost(B) + joinCost(A, B)$. The hash cost for some table $T$ is: $hashCost(T) = 2Pages(T)$ (note that this might not be the case since we may have many half full output pages resulting in more than Pages(T) being written, and there could be less if the input pages are not full since we can ensure that the hashing only write full pages). The join cost for A, B would be just the cost of reading each table once, so $Pages(A) + Pages(B)$.

Thus the total cost is: $3Pages(A) + 3Pages(B)$.

#### 19.8.10 Comparing MergeJoin to HashJoin

HashJoin wouuld be better if one of the relations is extremely large,since the merge join may require several passes. Wherase the merge join is better in cases where there is a lot of partitions that don't fit into the main memory.

Note that the result of a merge join is sorted, so if there is an ORDER BY in the sql, it may be more efficient to use it.

Also, Hash Join is easier to parallelize. Each pair of partitions can be matched by different processes or threads.

If there is a very large number of records, we can optimize further on the CPU side (not I/O) by hashing again once we've loaded a partition into memory and we are performing the join. This will tell each record of the inner table exactly which page to find the matching records (rather than only which partition).

### 19.9 Other Operations and Optimizations

#### 19.9.1 Projection

How does a projection actually get executed?

This is usually done during some other operation for efficiency. It's not usually required that the projection be done on its own. However, we cannot be too overzealous doing the projection early, since we may require some of the trimmed attributes later.

To see why this is useful, if we need to SELECT DISTINCT how do we chop the duplicates? A good way is to sort and then keep track of the previous value we processed. This

seems expensive, and if the developer put a distinct when it wasn't necessary (for example if there is a projection to the primary key), then there is no reason to do the extra step of sorting.

### 19.9.2 Set Operations

It turns out that intersection and cross product are special cases of join (intersection where we join on all attributes, and cross product where there is no condition).

Union (is distinct) and except are similar to each other and involve join with sorting. So for these we can use a merge-sort join.

Aggregate operations are generally done at the last step. If we do not have grouping, then we would need to scan the whole relation. But if we have grouping, then we would need to sort on the group-by attributes, then scan the whole relation to compute the function. If we instead compute the function *during* the sort, then we can save some I/O time.

# 20 Execution Plan + Pipelining

So how do we put all of this together? We must come up with a tree depicting how the query will be executed in the most optimal way. The leaves of such a tree will be the tables, and the root will be the result output.

Often its possible within the execution plan to send the result of one operator directly to the next. (Rather that writing back to disk).

The general strategy is to push projection and selection as far down the tree as possible, since this will lower the amount of data to be manupilated in higher operations. Don't worry right away about the specific join algorithms etc, until after the plan is done.