CO130 - Databases

Prelude

The content discussed here is part of CO130 - Databases (Computing MEng); taught by Thomas Heinis, in Imperial College London during the academic year 2018/19. The notes are written for my personal use, and have no guarantee of being correct (although I hope it is, for my own sake).

Material Order

These notes are primarily based off the slides on CATe. This is the order in which they are uploaded (and I'd assume the order in which they are taught).

1. *Introduction.pdf*
2. *ER Modelling.pdf*
3. *ER to RM.pdf*
4. *Relational Algebra.pdf*
5. *Tutorial ER.pdf*
6. *Tutorial Translation.pdf*
7. *Relational Algebra.pdf*
8. *Functional Dependencies.pdf*
9. *Tutorial Relational Algebra.pdf*
10. *Normalisation.pdf*
11. *SQL.pdf*
12. *Data Definition.pdf*
13. *Data Manipulation.pdf*
14. *Advanced SQL.pdf*
15. *Functional Dependency Tutorial.pdf*
16. *Transactions.pdf*
17. *SQL Tutorial.pdf*
18. *Storage.pdf*
19. *Indexing.pdf*
20. *NoSQL.pdf*
21. *MapReduce.pdf*

Introduction

We use databases as it's more organised; hence it's easier to model and manage. It's more efficient, as it's fast to search, and update, and integration allows us to minimise data duplication. Concurrent (and therefore multi-user) access allows multiple people to access the database at the same time (will require some techniques).

Transactions are sequences of database actions that execute in a coherent, and reliable way - the classical properties are **ACID**. Consider the two transactions `T1`: $A = A - 100; B = B + 100$, and `T2`: $B = B - 100; A = A + 100$, we can observe ACID properties as follows;

- **atomicity** if one part of a transaction fails, the entire transaction fails

  on completion of `T1`, either $A' = A - 100$, and $B' = B + 100$, or $A' = A$, and $B' = B$, where the former is a successful transaction, and the latter is in the case of a failure.

- **consistency** transactions don't leave the database in an inconsistent state

  the sum of the balances must remain the same, such that $A' + B' = A + B$; we can also have more constraints such as keeping balances positive, or limiting the amount a transfer can do at once

- **isolation** transactions run as if no other transactions are running (may need to wait)

  given the two concurrent transactions `T1`, and `T2`, one has to be completed before the other can start

- **durability** results of successful transactions aren't lost on system failure

  the new values, A', and B' must persist if the transaction completes, even if the system fails (disk failure etc.)

A **Database Management System (DMBS)** creates new databases via a **Data Definition Language (DDL)**, which specifies the structure (**schema**). It also queries, and manipulates through a **Data Manipulation Language (DML)**. Examples of this include *PostgreSQL*, *MySQL*, *SQLite*, and can also fall under *NoSQL*, however SQL remains as the most widespread technology (as of writing this).

It lets us define, query, and manipulate databases with a high-level declarative language (**Structured Query Language**). It's standardised by the ISO, but each DBMS implements its own variation of the standards, which may be costly if it's complex.

Relational Model

Consider the following model, represented as a table;

heading	`title:string`	`year:int`	`length:int`	`genre:string`
body	Gone with the Wind	1939	231	Drama
	Star Wars	1977	124	Science Fiction
	Luis' World	1992	95	Comedy

We have the columns be the attribute, with the top row being the heading, and the rest being the body. The attributes are in the format `name:type`. The rows (of the body) are referred to as tuples. A relation is the heading as well as the body (the entire table). The heading is an **unordered set** of attributes, and an attribute is the name as well as the type (typically indivisible types). The body is an unordered set of tuples, and a tuple is the set of attribute values. The schema is for the entire relation is the name of the relation, and the heading, in this case, we'd have; `movies(title:string, year:int, length:int, genre:string)`, and a database is a collection of relations. A schema for a database is the schemas for all relations.

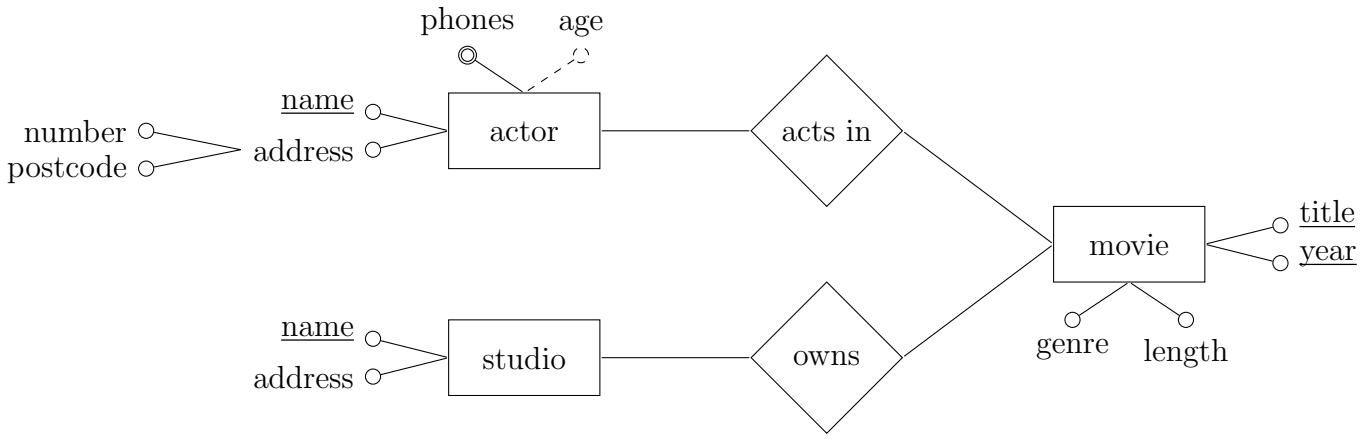
In mathematics, with a set of sets; $S_1, S_2, ..., S_n$, a relation R is a set of tuples $T_1, T_2, ..., T_n$, where $T_k \in S_k$, therefore $R \subset S_1 \times S_2 \times ... \times S_n$. As the idea of relations stems from set theory, it's important to note that the order in which we represent the attributes, and tuples in unimportant. In the Relational model we have **attributed** tuples, rather than **ordered**; R is the set of tuples $(A_1 : S_1 = T_1, ..., A_n : S_n = T_n)$, with $T_k \in S_k$. It's important to note that relations aren't 2-dimensional tables, even though it's more convenient to draw it on paper. We should instead consider them as a set of n-dimensional values, such that we have (`title:string`=StarWars, `year:int`=1997, `length:int`=127, `genre:string`=ScienceFiction), as a 4-dimensional movie value.

Entity Relationship Modelling

When a new database is being developed, it's important to try and model the real-world situation, instead of trying to refine it into an implementation, such as a relational model. In Entity-Relationship modelling, we try to create a diagram which represents the information needed for the database (the Entity Relationship Diagram). As there is no universally accepting notation for ER diagrams, we will use the following notation;

type	description	shape
entity sets	a set of distinguishable entries that share the same set of properties, can be physical (a room etc.), an event (flight, sale, etc.) - they normally correspond to nouns	rectangle
relationship sets	captures how **two or more** entity sets are related (e.g. owns, tutors), we can also have more than one relationship set between entity sets, and they can also have a relationship set on the same entity - they sometimes correspond to verbs	diamonds
attributes	properties of an entity; relationship sets can also have attributes, and primary keys are underlined	small circles

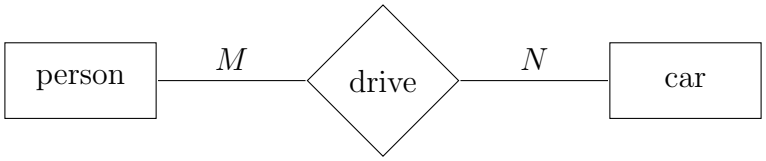
The movie example is represented below. Note that I've also extended it to contain different types of complex attributes. You can see that the address field is subdivided into number, and postcode, we have a multi-valued attribute in phones, and a derived attribute in age (can be calculated from date of birth)



Cardinality Constraints

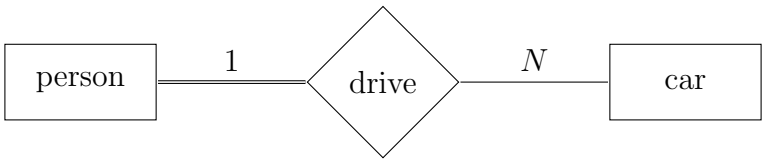
A relationship between two entity sets can be seen as one of the following (using the car example, where a person can drive N cars, and a car can be driven by M people);

- one-to-one $M = 1, N = 1$
- one-to-many $M = 1$
- many-to-many no restrictions

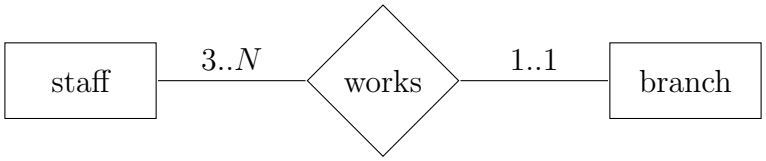


Participation

Using the same example, we can force participation, which means that all entities must participate in a relationship, with a double line. The example below, with the cars, suggests that everyone drives at least 1 car, and car can have at most 1 driver (note that it doesn't mean that every car has a driver).



Sometimes, instead of using double lines, E-R notation may allow explicit bounds. The below diagram suggests that staff work in exactly 1 branch, and that each branch must have at least 3 members of staff;



Fan, and Chasm Traps

See the diagrams in *ER Modelling.pdf* for examples.

We need to ensure that we do not allow ambiguous paths between entities. For example, if we say `staff` works for `faculty` which operates `department`, if some member of staff works for Engineering, which operates both Computing, and EE, we can't follow a path from the staff member to their department. This can be solved by having the staff work for the department, which is operated by some faculty.

If we can't follow a path between two entities, we may need to add another relation between them to specify the relationship.

Weak Entities

Entities which cannot be uniquely identified by their own attributes are called **weak** entities, in contrast to **strong** entities which have primary keys. A weak entity has to be defined by a string entity. Weak entities are drawn as a double rectangle, and the relationship to the strong entity is drawn as a double diamond. The key for a weak entity is formed by combining the primary key of the strong entity with attributes of the weak entity (denoted by a dashed underline). For example, a room in a building is a **weak** entity, with a room number, and can be uniquely identified with the name of the building combined with the room number.

ER to RM

Keys are used to join information when we do queries. The primary key is the unique identifier of a tuple, and a foreign key is the primary key of **another** table.

Once we have an Entity Relationship Model, we can then map it to a relational schema. The schemas can then be refined with functional dependencies, and implemented with relational languages. A high level data model isn't the only concern for database design (considered a simpler aspect).

If we consider a strong entity set, with simple attributes, we can easily create a table from it (see the example, `movie`, drawn above).

```
-- movie(title, year, length, genre)

CREATE TABLE movie (
    title VARCHAR(120),
    year INT,
    length INT,
    genre CHAR(20),

    PRIMARY KEY (title, year)
)
```

Composite attributes are also fairly easy to represent, since we just flatten it, therefore store each of the sub-attributes as their own field - consider the actor example (note that if the primary key is composite, we mark all the sub-attributes as primary keys). This example also shows how multi-valued attributes are stored - we create their own relation, and link back to the entity set with a foreign key constraint;

```
-- actor(name, number, postcode)
-- actor_phones(actorName, phoneID, other attributes)

CREATE TABLE actor (
    name VARCHAR(60),
    number INT,
    postcode VARCHAR(10),
    PRIMARY KEY (name)
)

CREATE TABLE actor_phones (
    actorName VARCHAR(60),
    phoneID VARCHAR(10),
    ...
    PRIMARY KEY (actorName, phoneID),
    FOREIGN KEY (actorName) REFERENCES actor.name
)
```

The relational model doesn't allow us to specify derived attributes, and we're therefore expected to calculate them with queries.

We can also consider how many-to-many relationships can be represented (using the car example again). The first example is when a car can be driven by many people, and a person can drive many cars;

```
-- person(ID, other attributes)
-- car(regno, other attributes)
-- drive(personID, regno, other attributes)

CREATE TABLE drive (
    personID VARCHAR(10),
    regno VARCHAR(12),
    ...
    PRIMARY KEY (personID, regno),
    FOREIGN KEY (personID) REFERENCES person.ID ON DELETE CASCADE,
    FOREIGN KEY (regno) REFERENCES car.regno ON DELETE CASCADE
)
```

However, it's easier to represent a one-to-many relationship, as we can simply include the primary key of the "one" relation as a foreign key in the many (where a car can be driven by 1 person, and a person can drive many cars). The representation for a one-to-one is similar, but the foreign key can be in either one, which is up to the designer;

```
-- person(ID, other attributes)
-- car(regno, personID, other attributes)

CREATE TABLE car (
    regno VARCHAR(12),
    personID VARCHAR(10),
    ...
    PRIMARY KEY (regno),
    FOREIGN KEY (personID) REFERENCES person.ID
)
```

In a weak entity, we include the primary key of the strong relation as a foreign key, but also use it as part of the composite primary key for the weak entity. A delete cascade is also used, and finally the we use `not null` to ensure total participation;

```
-- building(name, other attributes)
-- room(no, buildingname, other attributes)

CREATE TABLE room (
    no VARCHAR(120),
    buildingname VARCHAR(50) NOT NULL,
    ...
    PRIMARY KEY (no, buildingname),
    FOREIGN KEY (buildingname) REFERENCES building.name ON DELETE CASCADE
)
```

A multiway relationship can be mapped as several binary relationships, or we can generalise it to have the primary key consist of the relationship be composed of the primary keys of the many entity sets, and have the primary keys of the one entities be attributes. For roles, we map each role as a foreign key attribute in the entity set;

```
-- movie(ID, other attributes)
-- sequelof(originalID, sequelID, other attributes)

CREATE TABLE sequelof (
    originalID INT,
    sequelID int,
    ...
    PRIMARY KEY (orignalID, sequelID),
    FOREIGN KEY originalID REFERENCES movie.ID,
    FOREIGN KEY sequelID REFERENCES movie.ID
)
```

Extended Models

Newer models extend the classic model by supporting features from **object-oriented design**. We can look at how E-R models handle specialisation, or generalisation with **is-a hierarchies**. For example, we can specify that a `cartoon` **is-a** `movie`, which means that it inherits all the attributes of movie, but can have more attributes. and relations. Once this extends on multiple levels, we begin to form a hierarchy.

Relational Algebra

We can use **relational algebra** to construct new relations from existing ones, the operators include selection, projection, intersection, union, difference, and product. Consider the examples below, `a`, `b`, and `c` are all integers. Intersection, union, and difference will be omitted, since those are fairly self-explanatory.

R						$\pi_{\texttt{a,c,e}}(R)$			S			$\sigma_{\texttt{a>3\|b<5}}(S)$		
a	b	c	d	e	f	a	c	e	a	b	c	a	b	c
1	2	3	4	5	6	1	3	5	3	2	3	3	2	3
1	1	1	1	1	1	1	1	1	2	1	7	8	2	2
2	2	2	2	2	2	2	2	2	8	2	2			
1	2	3	4	5	8				1	2	9			

The syntax in use here for projection is $\pi_{\texttt{attributes}}(T)$, and likewise for selection it's $\sigma_{\texttt{condition}}(T)$. The former returns a relation with only the listed attributes, and the only takes rows which satisfy the given condition. Neither will return duplicate rows. Here is an example of a Cartesian product;

R		S			$R \times S$				
a	b	c	d	e	a	b	c	d	e
1	2	1	2	3	1	2	1	2	3
3	4	4	5	6	1	2	4	5	6
		7	8	9	1	2	7	8	9
					3	4	1	2	3
					3	4	4	5	6
					3	4	7	8	9

We can take a natural join, where the resulting relation contains all the tuples that have matching attributes in R, and S. This is less common, and we'd typically use something closer to $R \bowtie_{R.\mathtt{b}=S.\mathtt{b}} S$. Note that in this case, they are the same, since that's the only attribute that overlaps;

R		S			$R \bowtie S$			
a	b	b	c	d	a	b	c	d
1	2	1	2	3	1	2	5	6
1	3	2	5	6	3	4	8	9
3	4	4	8	9	3	4	9	9
		4	9	9				

Attributes can also be renamed with the notation $\rho_{\mathtt{new/old},...}(R)$ notation, which is fairly self-explanatory.

Functional Dependencies

Relational database schemas should be normalised, which helps reduce redundancy, and avoids update, and deletion anomalies. The basis for normalisation is the concept of **functional dependency**, and **keys**. Consider the following example, which can experience update anomalies;

producer	city	product no.	price	quantity
3	London	52	65	4
22	Birmingham	10	15	5
22	Birmingham	12	4	11
3	London	44	43	32
3	London	43	3	27

If an existing producer changes address, and tuples aren't updated, then it leads to incoherence (as the data would now be invalid). If a new tuple / row is inserted, and the producer already exists, but with an older address, we will once again have incoherence. If a producer doesn't have any open orders, say rows 1, and 5 didn't exist, then data about producer 3 would be lost. The price of the product doesn't depend on the producer, and the location of the producer doesn't have anything to do with the product. There is also redundant data, since the address is duplicated. Ideally, the data should be stored like this;

Open Orders		
producer no.	product no.	quantity
3	52	4
22	10	5
22	12	11
3	44	32
3	43	27

Producer	
producer no.	city
3	London
22	Birmingham

Product	
product no.	price
52	65
10	15
12	4
44	43
43	3

A functional dependency is a constraint that if two tuples of a relation R agree on a set of attributes $A_1, A_2, ..., A_n$, then they must also agree on the set of attributes $B_1, B_2, ..., B_m$. This can be written as $A_1, A_2, ..., A_n \rightarrow B_1, B_2, ..., B_m$, and therefore the B set is functionally dependant on the A set, or the A set functionally determines the B set. Therefore, for any set of values A, then there is only one set of values B. The normal forms a designer can use are defined in terms of their functional dependencies. This can be summarised by the following examples;

functional dependency	for all tuple pairs, x, y, if	then assert
$A \to K$	$x.A = y.A$	$x.K = y.K$
$A, B \to K$	$x.A = y.A$ $x.B = y.B$	$x.K = y.K$
$A \to K, L$	$x.A = y.A$	$x.K = y.K$ $x.L = y.L$
$A, B \to K, L$	$x.A = y.A$ $x.B = y.B$	$x.K = y.K$ $x.L = y.L$

Note that if we have two functional dependencies, $A, B \to K$, and $A, B \to L$, it's trivial to combine them as $A, B \to K, L$ (or to split them). If we have the functional dependency $A, B, C, D, E \to A, C, X, Y, Z$, we can remove all the attributes on the right hand side, that are on the left hand side, such that the aforementioned FD reduces to $A, B, C, D, E \to X, Y, Z$. A **trivial** FD is one where all the attributes on the RHS are on the LHS.

Keys

If we have some set of attributes $A_1, A_2, ..., A_n$, functionally determine all the remaining attributes of the relation, then the set A is a **superkey**. This also implies that two tuples cannot have the same superkey values, due to the uniqueness property. Let there be a set of all attributes in the relation, K, and the superkey set S. Therefore we can state $S \to \{x \in K \mid x \notin S\}$. A single relation may have more than one superkey, and superkeys can contain attributes that aren't strictly required. Generally, we are interested in superkeys where there isn't a subset of the superkey (hence the smallest superkey, by the irreducibility property), and this minimal superkey is referred to as the **candidate key**. If we have more than one candidate key, we can choose one of them to act as the primary key. A candidate key is also often referred to as just a **key**.

Closure

The set of all attributes functionally determined by a set L, under a set of dependencies F, the closure of L, denoted as L^+. If L^+ contains all the attributes of R, then it follows that L is a superkey of R. If RHS $\subseteq$ LHS$^+$, then LHS $\to$ RHS holds.

In order to compute the closure of a set of attributes, L, under a set of FDs given in the form LHS $\to$ RHS, we keep adding all the LHSs which exist as a subset of L, and add the RHS to L. We repeat this until we can no longer add anymore RHSs. The final value is referred to as L^+. Given the starting value of $L = \{A, B\}$, and the following FD set;

(1) $A, B \to C$

(2) $B, C \to A, D$

(3) $D \to E$

(4) $C, F \to B$

Starting with (1), both A, and B are in L, so we can add C. Now that C is in L, and B was already there, we can add D (no need to add A) from (2). Since we have D in L, we can add E. There is nothing else we can do, since F is not in L. Hence the value of $L^+ = \{A, B, C, D, E\}$.

Armstrong's Axioms

This is a sound (doesn't generate incorrect FDs), and compete (allows us to derive all valid FDs) axiomisation of FDs. Given attributes $A, B, C, ...$, and sets of attributes α, β, γ, we have the following axioms;

- reflexivity (trivial FDs) — $\alpha \to \beta$ always holds if $\beta \subseteq \alpha$
- augmentation — if $\alpha \to \beta$, then it follows $\alpha\gamma \to \beta\gamma$

- transitivity — if $\alpha \to \beta$, and $\beta \to \gamma$, then $\alpha \to \gamma$
- (derived) union — if $\alpha \to \beta$, and $\alpha \to \gamma$, then $\alpha \to \beta\gamma$
- (derived) decomposition — if $\alpha \to \beta\gamma$, then $\alpha \to \beta$, and $\alpha \to \gamma$
- (derived) pseudotransitivity — if $\alpha \to \beta$, and $\delta\beta \to \gamma$, then $\delta\alpha \to \gamma$

Closure, and Covers

Similar to attributes, we can find the closure of a FD set, which is the set of all FDs that can be inferred, denoted as F^+. This can be approached by applying Armstrong's axioms. We start by initialising F^+ to the set F. By applying reflexivity, and augmentation, we can add new FDs to F^+. Transitivity can then be applied to suitable FDs, and adding the derived ones to F^+. This is repeated until no more changes occur.

We can say two FD sets F_1, and F_2 are equivalent if they each imply the other - if any relation instance satisfying F_1 also satisfies F_2, they are said to be covers of each other. A cover is canonical / minimal / irreducible if each LHS is unique, and no FD can be deleted from the cover, and still maintain an equivalent FD set. Additionally, no attributes can be deleted from any FD set, and still have an equivalent set. Therefore, the canonical cover has no redundant dependencies, nor attributes.

The method for testing whether some attribute, X, is extraneous depends on whether it's on the LHS, or RHS. For an attribute X on the LHS, we can say X is extraneous if RHS $\subseteq \{\text{LHS} - X\}^+$ under the FD set - note that here we are referring to closure on attributes, not FDs. On the other hand (no pun intended, I'm so depressed), an attribute X on the RHS is extraneous if $X \in \text{LHS}^+$ under the FD set, where X is removed from the RHS of the FD.

To compute a canonical cover F for an FD set, we can use the following algorithm;

```
let F := FD set
do
    replace dependencies of α → β and α → γ, with α → βγ
    remove all extraneous attributes one at a time
until F doesn't change
```

Normalisation

As previously mentioned, in the previous section, normalisation reduces data duplication, and anomalies. One of the approaches taken to develop a normalised schema is to start with a few big relations, and then decompose them into a suitable normal form. The ones we'll look at are **Boyce-Codd Normal Form (BCNF)**, and **Third Normal Form (3NF)**. Consider the following example, which is just a "big" relation;

title	year	length	genre	studio	actor
Star Wars	1977	124	Science Fiction	Fox	Carrie Fisher
Star Wars	1977	124	Science Fiction	Fox	Mark Hamill
Star Wars	1977	124	Science Fiction	Fox	Harrison Ford
Gone with the Wind	1939	231	Drama	MGM	Vivien Leigh
Luis' World	1992	95	Comedy	Paramount	Dana Carvey
Luis' World	1992	95	Comedy	Paramount	Mile Myers

You'll notice straight away that there is a huge amount of redundant, repeated data, which is present in multiple rows. Updating the `length` of Star Wars would require 3 updates (for the relation to remain consistent), which is error prone. Inserting a new actor for a movie would require duplicating a fair bit of data, and would require checks to maintain consistency. Finally, deleting Vivien Leigh would delete the entire row, which deletes information about the movie.

Decomposition

Given a relation $R(A_1, A_2, ..., A_n)$, we can decompose it into two projected relations S, and T, such that $\text{attr}(R) = \text{attr}(S) \cup \text{attr}(T)$. This also means that $S = \pi_{\text{attr}(S)}(R)$, and similar for T.

When we decompose a relation, it's crucial for us to ensure we can recover the original relation by joining the decomposed relations, and also preserve the FDs of the original relation. Note that the latter isn't always possible with BCNF.

Given R decomposed into S, and T, the decomposition is **lossless** if at least one of the following FDs hold in the closure of the FD set of R;

- $\text{attr}(S) \cap \text{attr}(T) \rightarrow \text{attr}(S)$
- $\text{attr}(S) \cap \text{attr}(T) \rightarrow \text{attr}(T)$
- this means the common attributes of S, and T form a superkey of either of the decomposed relations

If we're able to check the FDs of R without joining the decomposed sets, then the decomposition is **dependency preserving**.

Consider the following example, on $R(A, B, C)$, with the FD set $\{A \rightarrow B, B \rightarrow C\}$;

decomposition	lossless?	dependency preserving?
$S(A, B)$ $T(B, C)$	$B \rightarrow A, B$ doesn't hold $B \rightarrow B, C$ holds $\therefore$ lossless	yes, $A \rightarrow B$ is checked with S, and $B \rightarrow C$ with T
$S(A, B)$ $T(A, C)$	$A \rightarrow A, B$ holds $\therefore$ lossless $A \rightarrow A, C$ holds also	no, $B \rightarrow C$ cannot be checked without joining

Boyce-Codd Normal Form

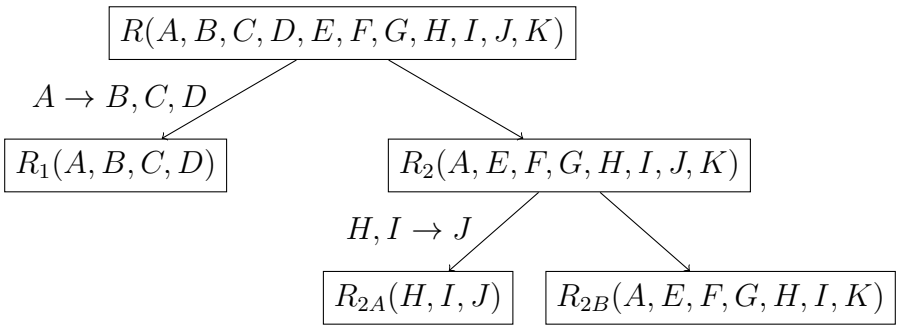
A relation R is in BCNF, if and only if, for all non-trivial FDs (including derived FDs) of the relation, the LHS of every FD is a superkey (i.e. contains a key)

For example, if we have the FD `title`, `year` $\rightarrow$ `length`, `genre`, `studio`, it's not in BCNF, as `title`, `year` doesn't functionally determine `actor`.

In order to convert something into BCNF, we first initialise a set of relations with just R. While the set contains relations that violate the rules of BCNF, take one of them, let it be V. Let us take some non-trivial FD for V, of the form LHS $\rightarrow$ RHS; fulfilling the conditions $\text{LHS} \cap \text{RHS} = \varnothing$ (therefore we may need to check a derived FD for decomposed relations), and LHS $\rightarrow \text{attr}(V)$ does not hold for the FD^+ of R (therefore, LHS is not a superkey of R). Now remove V from the set of relations. Add a new relation $R_k(\text{LHS} \cup \text{RHS})$, and another relation $R_l(\text{attr}(V) - \text{RHS})$ to the set of decompositions. This process is then recursively applied to the child relations. Consider the following example;

$R(A, B, C, D, E, F, G, H, I, J, K)$, and the following FDs;

- $A \rightarrow B, C, D$
- $H, I \rightarrow J$
- $A, E, F, G \rightarrow H, I, K$



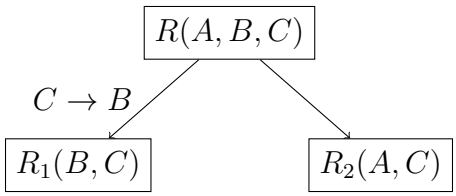
Honestly, just practice this.

Recovering Data

When we decompose into BCNF, it's crucial that we're able to recover the original relation by joining the decomposed ones. As long as the FDs hold for the original relation, it will always be possible. However; if the FDs don't hold, we'd end up with false data; as the join would generate tuples that don't exist in the original relation.

Dependency Preservation

As previously mentioned, we know that BCNF doesn't always preserve dependencies, which can be shown here. Let there be a relation $R(A, B, C)$, and the FD set; $\{\{A, B\} \to C, C \to B\}$. We wouldn't be able to check $A, B \to C$ without the RDBMS joining the relations.



Third Normal Form

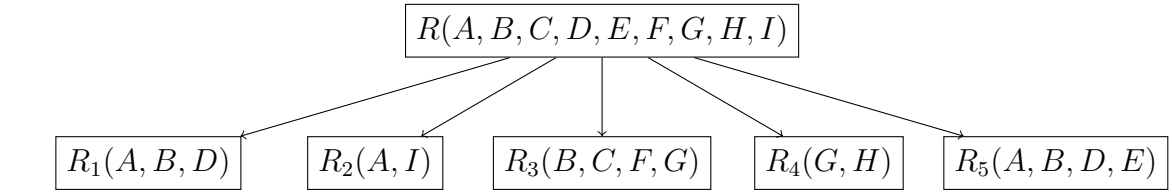
A relation R is in 3NF, if and only if, the LHS of every nontrivial FD is a superkey (same as BCNF test), or if every attribute on the RHS of a FD is prime (if it is a member of any key of the relation)

When a BCNF decomposition doesn't preserve the original FDs, we can either "live with" the violating dependencies, or use a weaker normal form which preserves FDs, but has redundancy. With **3NF**, there always exists a dependency preserving, lossless decomposition.

In order to decompose a relation R, and a set of FDs F for R, we can generate a set of decomposed relations D. We first need to find C, a canonical cover for F, such that it's a minimal FD set for R. Initialise D to be $\varnothing$. For every canonical FD, in the form LHS $\to$ RHS, we add a new relation (LHS$\cup$RHS) to D. Now, for every relation in D, if it is a subset of another relation in D, we remove it. Finally, if none of the relations in D include a key for R, we add a new relation to D, which contains a key.

Consider the following relation $R(A, B, C, D, E, F, G, H, I)$, a set of keys $\{\{A, B\}, \{A, D\}, \{B, D\}\}$ with the following FD set (and its canonical cover on the right);

• $A, B \to D, E$	$A, B \to D$
• $A \to I$	$A \to I$
• $B \to C, F, G, H$	$B \to C, F, G$
• $G \to H$	$G \to H$
• $A, D \to B, C, E, F, G, H$	$A, D \to B$
• $B, D \to A, I, E$	$B, D \to A, E$



dropped, as $R_1 \subseteq R_5$

SQL

Structured Query Language is the most common language for relational databases, with use in more than 99% of applications. It supports schema creation, modification, data insertion, retrieval, updates, deletion, and a lot more. Translating from the relational algebra we previously done, to SQL, can be done with the table below;

relational algebra	SQL	comments
$R \cup S$	R `UNION` S	
$R \cap S$	R `INTERSECT` S	
$R - S$	R `EXCEPT` S	
$\pi_{\texttt{attributes}}(R)$	`SELECT attributes FROM` R	
$\sigma_{\texttt{condition}}(R)$	`FROM` R `WHERE condition`	
$R \times S$	R, S or R `CROSS JOIN` S	
$R \bowtie S$	R `NATURAL JOIN` S	
$R \bowtie_{\texttt{condition}} S$	R `JOIN` S `ON condition`	
relation	table	persistent relations stored on disk
relational expression	views	based on other relations, not normally stored
tuple	row	also called a record
attribute	column	also called a field
domain	type	includes `char`, `int`, `float`, `date`, `time`, etc.

However, SQL differs from the theory we've studied so far, as it is based on multi-sets, and therefore can contain duplicate rows / tuples. It's still best practice to avoid them. Not all attributes need to be filled in, and therefore can be left `null`, once again this should be avoided; because of this, booleans are three-valued (true, false, and unknown). Most implementations have a wide range of types, and therefore numbers have multiple types - arithmetic operators are usually available. Strings can either be a fixed length, and padded with spaces, or varying length. The concatenation operator used in SQL is `||`, and we can use pattern matching, with `_` denoting any character, and `%` matching zero, or more characters. We can store bits, bytes, and large binary objects (blobs) to hold images, movies, and other files.

With booleans, we have additional comparison operators such as `BETWEEN`, `NOT BETWEEN`, `IN`, and `NOT IN`, all of which should be fairly self-explanatory. The truth values for 3VL (three valued logic) can be computed with the following mapping;

- 1 - true
- $\frac{1}{2}$ - unknown
- 0 - false
- x `AND` $y = \min(x, y)$
- x `OR` $y = \max(x, y)$
- `NOT` $x = 1 - x$

It's also important to remember that any arithmetic which uses a `null`, will result in `null`. Any comparisons that use `null` will result in unknown.

Queries

In order to save space, all comments will be done inline, in the SQL code listing below;

```
1 -- movie(title, year, length, genre, studio, producer)
2 -- casting(title, year, name)
3
```

```
-- note that the results of a selection is still a relation, and therefore we can use it as a subquery in other expressions
SELECT title, length/60 AS hours -- projected attributes (and a rename)
FROM movie -- the relation
WHERE studio='fox' AND year > 1990 -- the condition
ORDER BY year DESC, title ASC -- sorts first by year
-- note that we're using attributes which aren't in the projection, the order of operations is FROM, WHERE, ORDER, SELECT
-- note that we can also prefix the fields, to make it more readable
```

In order to join two relations, we can use a `JOIN`;

```
SELECT title, year, name
FROM movie JOIN casting ON (movie.producer=casting.name)
-- we can do a theta join with an ON, and a condition

SELECT title
FROM movie JOIN casting USING (title, year)
-- note that this is the same as ON (movie.title=casting.title AND movie.year=casting.year)

SELECT c1.name, c2.name
FROM casting AS c1 JOIN casting c2 -- note how we can omit AS for readability
ON c1.address = c2.address AND
   c1.name < c2.name
-- we can do a self join, but we have to name the relation, these are called correlation names
```

We can also consider the different types of joins, either natural (by matching attributes), or theta (by condition);

- inner join — returns tuples when there is at least one match in both sides
- left outer join (`LOJ`) — similar to inner join, but will include all tuples from the left relation
- right outer join (`ROJ`) — similar to inner, but includes all tuples from the right relation
- full outer join (`FOJ`) — similar to inner, but includes all unmatched tuples

L			R			L `LOJ` R				L `ROJ` R				L `FOJ` R			
a	b	c	a	b	d	a	b	c	d	a	b	c	d	a	b	c	d
1	2	3	2	3	A	1	2	3	A	1	2	3	A	1	2	3	A
4	5	6	2	3	B	1	2	3	B	1	2	3	B	1	2	3	B
7	8	9	6	8	C	4	5	6	N	N	6	8	C	4	5	6	N
						7	8	9	N					7	8	9	N
														N	6	8	C

```
SELECT DISTINCT title, year, name -- selects only unique tuples
FROM movie LEFT OUTER JOIN casting ON
    movie.producer=casting.name AND movie.year=casting.year
```

We also have a set of aggregation functions, which can be used to calculate a single value, for example;

```
SELECT department
       COUNT(*) as professors,
       SUM(salary) as totalsalary,
       AVG(salary) as averagesalary,
       MIN(age) as youngest,
```

```
6         MAX(age) as oldest
7  FROM employee
8  WHERE position='Professor'
9  GROUP BY department
10 HAVING COUNT(*) >= 10
11 ORDER BY totalsalary DESC
```

This creates a tuple for each department; the results are then sorted in descending order by total salary. Any non-aggregates used in the projection, or the `HAVING` filter **must** be included in the `GROUP BY` list. Only departments with at least 10 professors are listed.

Subqueries

A powerful feature of `SELECT`s is that we can also use them as subqueries in expressions, by wrapping them in brackets. The types supported by SQL are;

- scalar subquery

  a subquery that produces a single value, normally uses an aggregate function

```
1  SELECT title,
2      (SELECT count(name)
3       FROM casting
4       WHERE casting.title=movie.title) AS numactors
5  FROM movie
6  -- note how the outer query is related to the subquery; this is an example of
       a correlated subquery that has to be evaluated for each outer tuple
7
8  SELECT title, count(name) as numactors
9  FROM movie JOIN casting USING (title)
10 GROUP BY title
11 -- this is equivalent, and is clearer
```

- set subquery

  creates a set of distinct values (a single column), typically used for set membership with `(NOT) IN`, or set comparisons with `ALL` or `SOME`

```
1  SELECT title
2  FROM movie
3  WHERE studio IN (SELECT name
4                   FROM studio
5                   WHERE address LIKE 'C%')
6  SELECT name
7  FROM casting
8  WHERE (title, year) NOT IN (SELECT title, year
9                              FROM movie
10                             WHERE genre='sf')
11 -- note how we're able to match on tuple values
12 SELECT title
13 FROM movie m1
14 WHERE year < SOME(SELECT year
15                   FROM movie m2
16                   WHERE m2.title=m1.title)
17 -- the keywords ALL, and SOME can be used with any comparator
18
19 SELECT name
```

```
FROM employee
WHERE salary <> ALL(SELECT salary
                    FROM employee
                    WHERE position='Professor'
```

- relation subquery

  produces a relation, which is typically used as an operand, used with operators `(NOT) EXISTS` to check if it's empty, or operators `(NOT) UNIQUE` to test for duplicate tuples

```
SELECT title
FROM movie m1
WHERE NOT EXISTS(SELECT *
                 FROM movie m2
                 WHERE m2.title=m1.title AND m1.year<>m2.year)
```

SQL Data Definition

SQL's DDL is concerned with schema creation, as well as the specification of any constraints. This course mainly focuses on base relations (which are stored tables), and lightly touches on derived relations (views). The constraints we deal with in SQL include type, primary / foreign key constraints, uniqueness, checking a value exists (not null), check constraints, as well as assertions.

Creating a Relation

Given some relation `movie(title, year, length, genre, ISAN)`, the following SQL instruction will create the table;

```
CREATE TABLE movie (
    title VARCHAR(120),
    year INT DEFAULT 2011,
    length INT NOT NULL DEFAULT 0,
    genre CHAR(20) NOT NULL,
    ISAN CHAR(24) NOT NULL,
    PRIMARY KEY (title, year),
    UNIQUE (ISAN)
)
-- this table can be deleted with 'DROP TABLE movie'
-- please sanitise your inputs (https://xkcd.com/327/)
```

We can then modify the schema, with the `ALTER TABLE` command, for example (leads to the modified schema `movie(title, year, genre, studio, ISAN)`);

```
ALTER TABLE movie ADD studio CHAR(16) DEFAULT '';
ALTER TABLE movie DROP length;
```

Note that we've also assigned a value that is unique, hence no other tuple can have the same `ISAN` value (if a command were to attempt to violate this constraint, such as via `UPDATE`, or `INSERT`, it would fail). Adding a constraint to prevent the assignment of nulls is also shown above.

Primary Key

Each relation should be given a primary (candidate) key, which determines the other attributes, and uniquely identifies each tuple. This is also used to enforce foreign key constraints. The constraints on a primary key are as follows; the `NULL` is not permitted in a primary key, and the whole primary key (in the case of composite keys) has to be unique to the tuple. For example, the following would fail, since we haven't given a value to year (hence it's `NULL`);

```
INSERT INTO movie (title, genre) VALUES ('Up', 'cartoon');
```

Check Constraints

In order to prevent the repetition of code, I'll simply leave `...` where the original code (from the first declaration of the table) is. While giving types to the attributes, and ensuring that they aren't nulls allow us to limit the values we store, we need `CHECK` constraints to define predicates that must be satisfied on the insertion, or update of a tuple. While the first two are probably the most common types of check (where it's a simple expression on a single attribute), we can have arbitrary expressions that may involve other queries;

```
CREATE TABLE MOVIE (
    ...
    CHECK (year BETWEEN 1900 AND 2020),
    CHECK (genre IN ('sf', 'comedy', 'drama', 'western')),
    CONSTRAINT validISAN CHECK (ISAN in (SELECT no FROM ISANcatalog))
)

CREATE ASSERTION nopoorbosses CHECK (
    NOT EXISTS (
        SELECT s.name
        FROM studio s JOIN movieboss m ON (s.boss=m.name)
        WHERE m.networth < 314159265 -- some large number
    )
)
```

You'll note that we can also name our constraints, which not only allows us to modify them with `ALTER TABLE`, but also clarifies errors when they are violated. There's also an assertion, but they are difficult to efficiently implement, despite being a powerful feature.

Foreign Key Constraints

One method of ensuring referential integrity is to apply constraints on foreign keys, which specify that the value of some set of attributes in a table must reference (match) primary key values of another relation, in this case, we're dealing with the schema `casting(title, year, name)`;

```
...

CREATE TABLE casting (
    title VARCHAR(120),
    year INT,
    name VARCHAR(60),
    FOREIGN KEY (title, year) REFERENCES movie (title, year) ON UPDATE CASCADE ON
       DELETE CASCADE
)
```

Another important factor in maintaining referential integrity is to cascade updates, whether it being a field update, or a deletion. Any update to the referenced attribute will cascade down to the foreign key. If the referenced tuple (a `movie` in this case) is deleted, then the entry referencing tuple is also deleted (a `casting` here). Instead of cascading updates, or deletions, we can set use other policies to update the referencing tuple (although this will lead to unmatched tuples). In our case, if the referenced tuple is deleted, the referencing fields are set to null, and if it's updated, then the fields are set to the default values;

```
CREATE TABLE casting (
    title VARCHAR(120) DEFAULT '',
```

```
    year INT DEFAULT 2011,
    name VARCHAR(60),
    FOREIGN KEY (title, year) REFERENCES movie (title, year) ON UPDATE SET DEFAULT
        ON DELETE SET NULL
)
```

Views

Views are temporary relations that can be defined with a query. For example, given the same schemas previously used, this can then be queried as if it were a stored relation;

```
CREATE VIEW comedies AS
SELECT title, year
FROM movie
WHERE genre='comedy';

CREATE VIEW actorGenre(actorname, moviegenre) AS
SELECT DISTINCT name, genre
FROM movie JOIN CASTING USING (title, year);

SELECT * FROM comedies WHERE year=2010;
```

Indexing

If we were to have a large database of movies, it could be quite slow to check. As such, we can make copies, which are automatically updated by the RDBMS. Once again, it leads to a trade-off between space and time.

```
CREATE INDEX yearindex ON movies (YEAR);
```

Data Manipulation, and Advanced SQL

Insertion

When we run a query to insert items into a table, missing values are set to `NULL` (unless the schema doesn't allow that), or are set to a default value (if one is specified). If it fails any constraints, such as uniqueness, then an error is given, and the tuple is not added to the table, and any associated transaction is rolled back. We're also able to insert values using a subquery, for example, the second command adds studios which are present in the movie relation (which aren't present in the studios relation) to the studio relation. An insertion subquery will be fully evaluated before the insertion occurs to prevent any anomalies.

```
INSERT INTO movie (title, year, genre)
VALUES ('True Grit', 2010, 'Western'),
    ('The Kings Speech', 2010, 'Drama')

INSERT INTO studio (name)
SELECT DISTINCT studio FROM movies m
WHERE m.studio NOT IN (SELECT s.name FROM studio s)
```

Deletion

We're also able to delete from a relation. It's extremely important to state a `condition`, otherwise the entire relation will be cleared.

```
DELETE FROM movie
WHERE length > 180 OR
      (title, year) IN (SELECT title, year
                        FROM casting
                        WHERE name='Keanu Reeves')
```

Updating

We're able to update tuples that satisfy a condition as follows. Similar to insertion, the transaction is aborted if it violates any constraints;

```
UPDATE employee
SET salary = CASE
    WHEN salary <= 70000 THEN salary * 1.02
    WHEN salary <= 80000 THEN salary * 1.03
    ELSE salary * 1.04
    END,
    lastpayincrease = current_date
WHERE position='Professor'
```

Advanced Operations

This was its own set of slides, but it's actually not that long, so it's just a list now.

- `LIMIT` clause — used to specify (limit) the number of records to return

```
SELECT columns
FROM table
LIMIT number;

SELECT * FROM persons LIMIT 2; -- only gets the first two rows
```

- `LIKE` operator — used in a `WHERE` clause to search for specific patterns in a column

```
SELECT columns
FROM table
WHERE column LIKE pattern;

SELECT * FROM persons WHERE city LIKE 'S%'; -- only gets people from cities
   starting in 'S'
-- % is used to substitute zero, or more characters
-- _ is used to substitute exactly one character
```

- `AUTO_INCREMENT` — allows a unique number to be generated when new records are added

```
CREATE TABLE persons (
    p_id INT NOT NULL AUTO_INCREMENT,
    lastName VARCHAR(255) NOT NULL,
    firstName VARCHAR(255),
    address VARCHAR(255),
    city VARCHAR(255),
    PRIMARY KEY (P_Id)
)
```

Transactions

Consider a client program, executing three SQL statements in one go;

1. update
2. insert
3. update

However; the databases 'crashes' between statements 2, and 3. The reason for the crash could be anything; the server being unable to handle the request (powering off, disk crashing), or a failed connection between the client, and server. Now, consider the same scenario; another issue between 2, and 3, but instead of a server crash, it's a constraint violation from the insert. Due to this; we'd have more precise information on the failure, however in terms of consistency, we'd like to have the same semantics in both cases.

If we were to consider all three statements executed by the client as a single transaction, we need to consider its **atomicity**. It is supposed to guarantee that either the entire transaction executes, or none of it does. If a failure occurs, the whole transaction should be aborted and rolled-back - but the roll-back is handled by the DBMS, not the client.

Consistency

Once a transaction is successful, also known as **committed**, it must leave the database in a consistent state (given that the database was consistent before the transaction started). However, this is something the programmer must ensure, while SQL's constraints assist the programmer, more complex rules must be checked by the application.

Atomicity, and Durability

If transactions are aborted for any reason, the DBMS must revert to the state before the transaction started (including schema modifications). The recovery system of a DBMS maintains a log of all changes, which includes old values (as such, it also handles durability). When a DBMS starts up, the recovery system typically checks logs to ensure all successful transactions were saved to disk, and the rest are rolled back.

In SQL, we can manually start a transaction as follows (although this is automatically in some SQL clients);

```
1 START TRANSACTION; -- BEGIN in some systems
2 SELECT ...;
3 INSERT ...;
4 UPDATE ...;
5 ...
6 COMMIT;
```

Replicated Servers

In order to guard against disk failures, a common approach is to take a RAID set-up, which improves performance by striping, and also improves durability by having redundant mirrored data. In order to cope with server failures, the entire database server could be mirrored, such that if main server were to crash, then a replicated one would take over.

Isolation

While it would be trivial to have isolation by serialising our transactions, it would slow down a frequently accessed databases, and would waste resources. Therefore, concurrent transactions are essential for real databases, but require techniques to preserver consistency whenever there's concurrent access. Isolation is handled by a DBMS' **concurrency control system**, that maximises concurrency but still ensures serialisability. The DBMS can easily allow both transactions to run concurrently if it's one of the three following cases;

1. both transactions are read-only
2. they modify different relations
3. they modify the same relation, but different tuples

We therefore have different levels of isolation in SQL;

- read uncommitted

  uncommitted data changed by other transactions can be read (dirty reads) - all changes by other transactions are immediately visible (lowest isolation)

- read committed

  only data that has been committed by another transaction can be read

- repeatable read

  similar to read committed, but guarantees all rows returned by a query will be included if we repeat the query (however, may return new tuples)

- serialisable

  trades the degree of concurrency for guaranteed isolation

level	dirty reads	non-repeatable reads	phantom reads
read uncommitted	possible	possible	possible
read committed	prevented	possible	possible
repeatable read	prevented	prevented	possible
serialisable	prevented	prevented	prevented

At no level are dirty writes allowed, which means we cannot update data that has been modified, but not yet committed.

Remarks

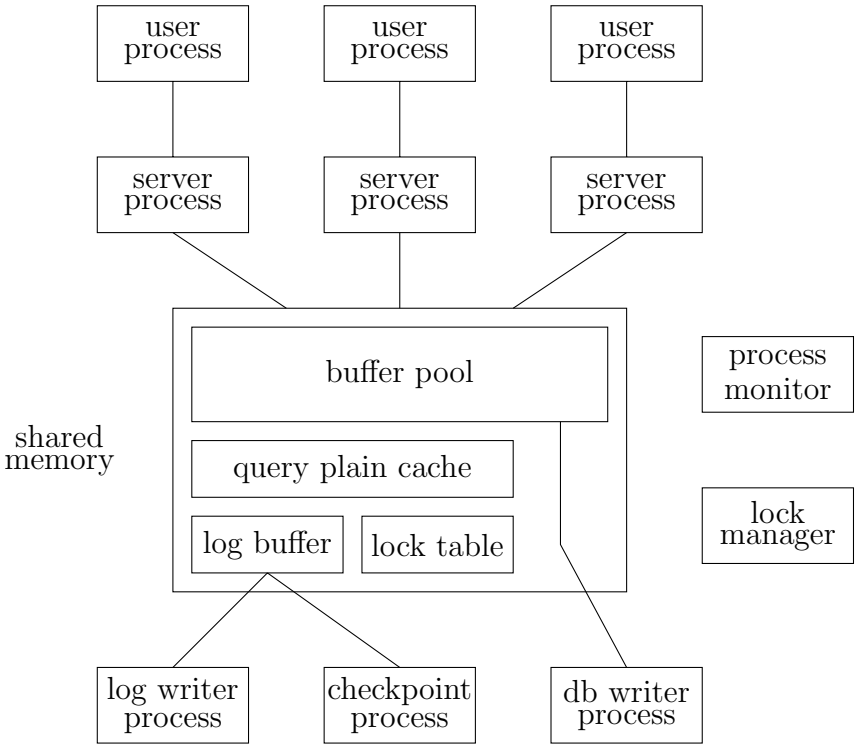
Transactions are one of the most important concepts in computing. ACID properties are fulfilled by the following; atomicity, and durability are implemented by a recovery system, consistency requires the programmer's implementation, although the DBMS has features to help, and finally isolation is handled by a concurrency control system which tries to maximise concurrency but maintain serialisability.

Storage

Requests are specified in SQL, and sent to a query / database server via a remote procedure call mechanism. Standard transaction servers consists of the following processes;

- server — receive user queries (transactions), executes them, and sends results back
- lock manager
- database writer — outputs the modified buffer to disks
- log writer — server processes add logs to the buffer, and the writer writes to stable storage

- checkpoint — performs periodic checkpoints
- process monitor — monitors other processes, and takes recovery action on failures



To ensure that multiple processes aren't accessing the same data, systems implement **mutual exclusion**, with either OS semaphores (controls access), or atomic instructions (test-and-set).

We can classify physical storage based on the following factors;

- access speed
- cost per unit of data
- reliability (data loss on system failure, and physical failure of storage)

The common types of physical storage used are as follows (ranked as a hierarchy);

1. cache

   fastest, and most costly - also volatile, and managed by the hardware

2. main memory

   fast access, but too small (or expensive) to store entire database - also volatile

3. flash memory

   non-volatile, and can survive power failure, data can be rewritten a limited number of times (10K - 1M cycles); fast reads, but slow writes, and erases

4. flash memory (NOR)

   fast reads, very slow erase, and low capacity

5. flash memory (NAND)

   high capacity

6. magnetic disk

   primary medium for long-term storage

7. optical storage

   reads, and writes slower than magnetic disk

8. tape

   extremely high capacity, primary used for backup (and archival data)

Indexing

When we create an index (briefly mentioned above), it helps to speed up access to desired data. Another basic concept is the **search key**, which is an attribute used to look up records in a file. The index file consists of records, referred to as index entries, in the form | search-key | pointer |. The index files are typically much smaller than the original files. Indices are either ordered, where the search keys are stored in a sorted order, or search keys are distributed across "buckets" with a hash function.

Ordered

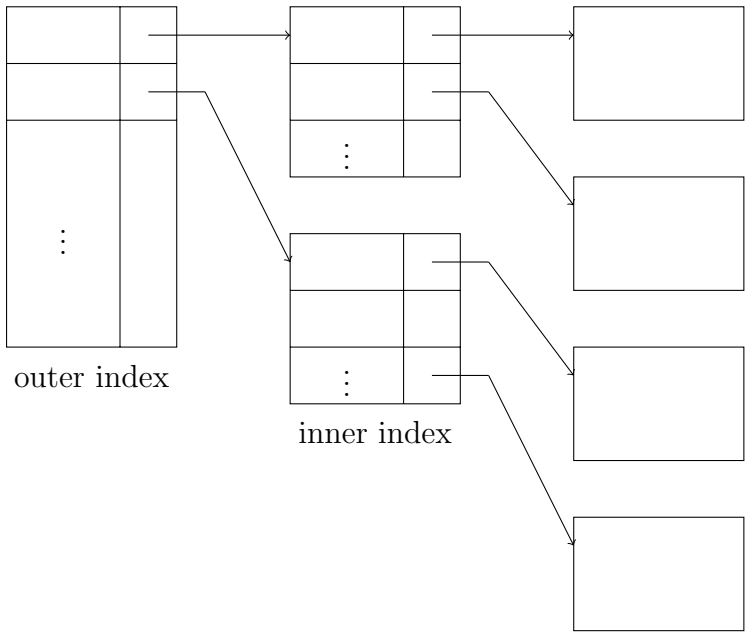
In an ordered index, entries are sorted on the search key. In a sequentially ordered file, the primary index is the index whose search key specifies the order of the file (also referred to as a clustering index). This is usually the primary key. On the other hand, a secondary index is one whose search key specifies a different order from the sequential order of the file.

Index Density

In a dense index, there is an index record for every search-key value in the file, as opposed to a sparse index file, where it only contains index records for some search key values. In order to find a record with the value K, we need to first find the index record with the largest search key, below K, and then search sequentially after that. A sparse index takes less space, and has less maintenance overhead for insertion and deletion, but is slower at locating records.

Multilevel Index

If a primary index cannot fit in memory, access then becomes expensive. We can solve this by treating the primary index as a sequential file, and then creating another sparse index on it, where we have an outer index, which acts as a sparse index of the primary index, and the inner index being the original primary index file. The same process can be applied to the outer index if it's also too large to fit in memory, and so on. All levels must be updated on insertion, or deletion.



Record Deletion

If a deleted record was the only record in the file with the particular search-key value, the search-key value is deleted from the index (this is propagated across the multi-level index). In a dense index, the search-key deletion is similar to the file record deletion, but in a sparse index, if the value exists in the index, the value is replaced with the next search-key value in the file. If the value already exists in the index, the index record is deleted.

Record Insertion

If the search-key value doesn't exist in the index, add it in (dense indices), on the other hand, with a sparse index, no change will be made, unless a new block has to be created. If a new block has to be created, the first search-key value in the block is inserted into the index. Multi-level insertion is done with the same process.

Secondary Indices

I find it easier to consider this as a multi-level index, with the inner index being dense (therefore the index records points to a bucket that contains pointers to the actual records)

B$^+$-Tree Index Files

Indexed-sequential files have some disadvantages in that the performance gets worse as the file gets bigger, since overflow blocks need to be created. Periodically, the entire file will also need to be reorganised. On the other hand, with B$^+$-tree index files, it can automatically reorganise itself with small changes when insertions, and deletions occur. However, there is additional overhead in insertion, deletion, as well as in space - but these disadvantages are outweighed by the advantages. The properties of a B$^+$-tree are as follows;

- all paths from the root to leaf are of equal length
- each node that is not a root, nor a leaf, has between $\lceil \frac{n}{2} \rceil$, and n children
- a leaf node has between $\lceil \frac{n-1}{2} \rceil$, and $n-1$ values
- if the root is not a leaf, then it has at least 2 children, otherwise if the root is a leaf (when there are no other nodes in the tree), it can have between 0, and $n-1$ values

Pictured below is the structure of a typical node. We have K_i as the search values, and P_i are pointers to children (for non-leaf nodes), or pointers to records (or record buckets) when it is a leaf node. The keys are ordered $K_1 < K_2 < ... < K_{n-1}$.

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

$\forall i \in [1..n-1]$ the pointer P_i either points to a file record with the search-key value K_i, or to a bucket of pointers, where all the pointers in the bucket have a search-key value of K_i. The bucket structure is only needed if the search-key doesn't form a primary key. If we have leaf nodes L_i, and L_j, and $i < j$, it follows that L_i's search-key values are less than L_j's search key values. P_n points to the next leaf node, in the search-key order.

On the other hand, non-leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers, all the search-keys in the subtree to which P_1 points, are less than K_1. $\forall i \in [2..n-1]$, all search-keys in the subtree to which P_i points have values in the range $[K_{i-1}, K_i)$. All the search-keys in the subtree to which P_n points to have values $\geq K_{n-1}$.

We can find all records with a search-key value of `k` via the following method;

```
1  N = root
2  do
3      examine N for the smallest search-key value > k
4      if value exists
5          assume it is K_i
6          N = P_i
7      else it follows k >= K_n - 1
8          N = P_n
9  until N is a leaf node
10 if for some i, K_i = k
```

```
    follow pointer P_i to record
else
    no record with search-key value k exists
```

Note that at some point in the next paragraph I will start using S-K V, because I think this degree is giving me RSI.

In order to insert values into the tree, we find the leaf node in which the value would appear. If the search-key value is already present, we add the record to the file, otherwise we add the record to the main file (and generate a bucket if needed). If there's room in the leaf node, insert the (key-value, pointer) pair in the leaf node, otherwise split the node. In order to split a leaf node, we take n (S-K V, P) pairs, in a sorted order (including the pair being inserted). Place the first in the original node, and the rest in a new node. We then create a new pair (k, p), in the parent of the node being split. This is then propagated upwards if the parent is also full.

To delete a record from the tree, we need to find the record and remove it from the main file (as well as from the bucket). Remove the pair from the leaf node, if the bucket becomes empty (or if there wasn't a bucket in the first place). If the node now has too few entries, and it is able to merge with a sibling to fit into a single node, we then merge siblings. This is done by inserting all the S-K Vs in the two nodes into a single node (the left-most one), and deleting the other. The deleted pair is then deleted from the parent, done recursively. However, if it cannot combine with a sibling, then we need to redistribute the pointers. Distribute it such that the number of pointers in both the current node, and the sibling node have more than the minimum number, and update the corresponding S-K V in the parent of the node.

The index file degradation problem is solved by using indices, and the data file degradation problem is solved by the tree file organisation. The leaf nodes in a B^+-tree file organisation store records, instead of pointers. Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is smaller than the number of pointers in a non-leaf node.

Hashing

We can also use a hash function to obtain the bucket for a record, directly from it's S-K V. Given some hash function h, it maps the set of S-K Vs K to the set of bucket addresses B. However, records with different S-K Vs can still be mapped to the same bucket, so the bucket will still need to be sequentially searched (however, on a smaller data set, provided the hash function is good). The worst hash function would map all S-K Vs to the same bucket, whereas an ideal function is uniform and assigns the same number to each bucket.

NoSQL

However, due to the strong consistency needed for ACID transactions, it can be difficult to scale, especially because the idea of transactions was built into the optimisation of relational databases. Changing schema can also be difficult. One of the main objectives of NoSQL is to distribute the state, such that different objects are stored on different servers, by giving up ACID constraints, it can improve performance, we also give up strong consistency for eventual consistency. The key features of Not Only SQL / Not Relational are as follows;

- horizontal scaling (more machines, instead of a more powerful machine)
- replicate / distribute data over servers
- simple call level interface
- weaker concurrency than ACID
- efficient use of main memory (and indices)
- flexible schema

NoSQL implementations can be split in to four categories; Key-Value, Graphs, BigTable, and Documents (nested values, like XML, or JSON).

Key-Value

Think of it more as a file system than a database, and there is only a primary key (hence we lookup by key). Given some table of pairs, where the key k_i, has a corresponding value v_i, we have a simple API where lookup(key) leads to a value, lookup(key range) leads to a set of values, and so on. Insertion requires the pair, and deletion requires the key.

Document Store

A pointerless object, and can have secondary indices, in addition to the KV stores.

Column Family

Most implementations are based on Google's **BigTable**. It supports semi-structured data, is scalable, and is naturally indexed, however it's poor for interconnected data.

Graph Databases

Modelled based on nodes, and relationships. Powerful data model, and general like RDBMS, easy to query, and connected data is indexed locally. A graph database has an explicit structure, where each node knows its adjacent nodes. The cost of a local step remains the same, even if the number of nodes increase.