# Database

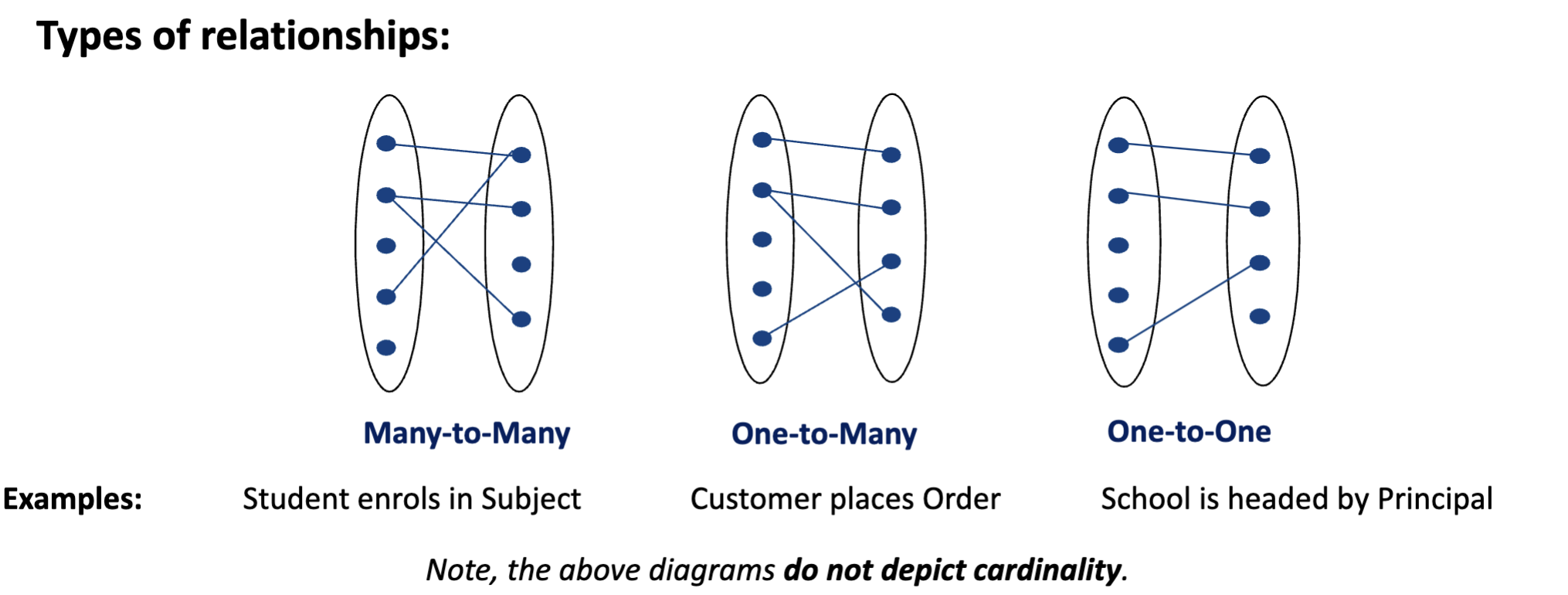
* The entity-relationship model
* ER model/ ER diagram does not actually store any data, it a plan how data represent and store in the database
* Basic ER modeling concepts:
* Uses entities to represent like people, object……

(Whatever you need to store about is your entity)

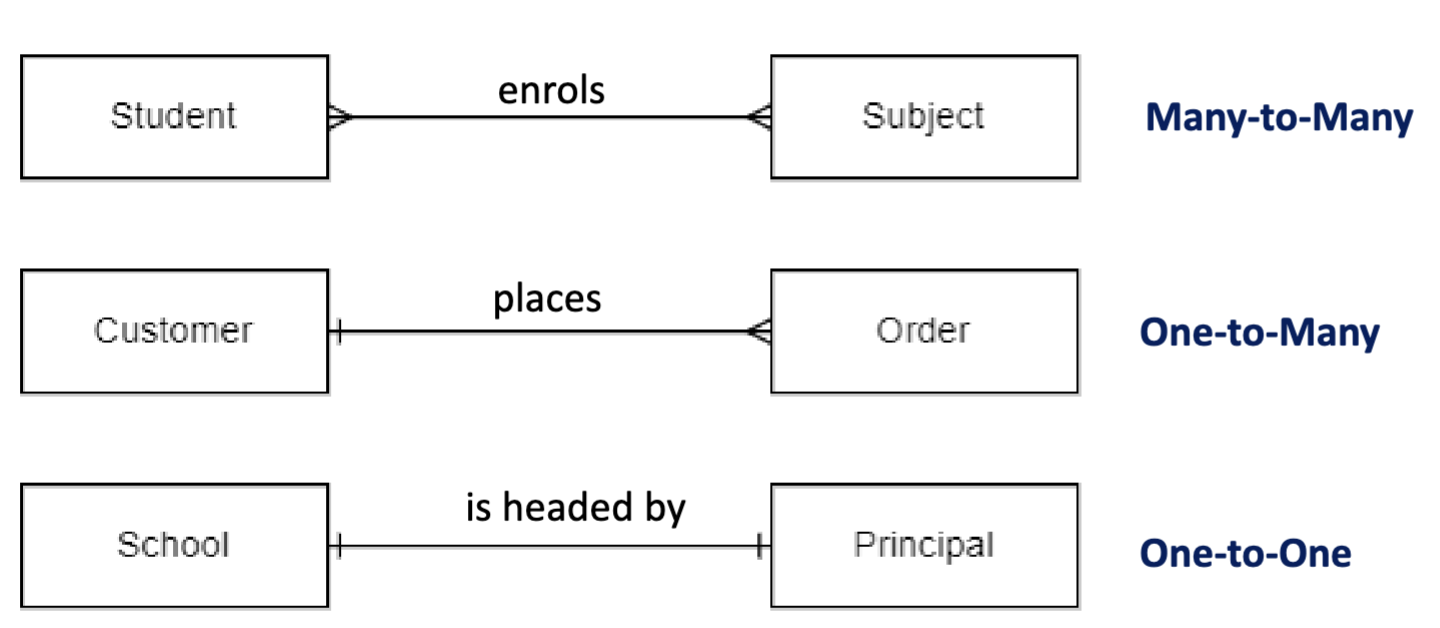
* Identifies relationships between various entities. Entity relates each other in some way.
* Based around business rules (constrain for entities)
* Entity and its attributes:
* We describe entity with attributes, because the same entity could be different base on domain
* Two annotation
  + Chen models
  + Crow’s Foot models
* Attributes:
* Name must not have space (use underscore or camelCase for readability)
* Have a that can set the possible values for a given attributes
* Domain is described by the company’s business rules (constrain)
* Attributes may share a domain (two attributes may need to follow same rules, like phone number of staff and customers adhere to the same rules)
* Required attributes: must have values
* Optional attributes: it’s ok to left blank
* Single-valued vs multi-valued attributes:
* A single-valued attribute can have only single value
* A multi-valued attribute can have many values (like the colors of the car)
* Composite vs simple attributes:
* Composite attribute can be further subdivided into additional attributes. Ex: address
* Simple attribute cannot be subdivided further. Ex: gender
* The different between multi-valued and composite attributes is that first one must be the same type of value and the later one contains different type of values
* Derived attribute: can be calculated using other attribute, data and function. Ex: age is derived by DOB and current date
* Identifier/ Key attribute/ Primary key
* Identifier is an attribute that unique identifies an instance of entity, which is also call primary key (PK)
* In order to avoid the same attribute values
* Rules:
* Every entity must have a unique identifier
* Have a unique identifier
* No duplicate
* The value cannot be null
* Relationship and relationship set
* Connectivity and Cardinality:
* Connectivity: used to describe the relationship classification
* Ex: 1:1, 1 to many (1: M), and many to many (M: M)
* Cardinality: expresses the specific (min and max) number of entity occurrences associated with one occurrence of the related entity
* Cardinality depends on business rules
* Ex: A credit card account can have up to 2 cards associated with it (minimum 1 card, maximum 2 cards) 1…2 or 1,2

– One credit card belongs to 1 account only

* ER models and business rules:
* Your ERD design and subsequent database design must meet the needs of the organization’s business rules
* Connectivity / Relationship Types:
* Connectivity determines the number of entities taking part in the relationship set (how many from each side)



* Crow’s foot notation:



* Relationship Participation/ Cardinality:
* Optional:
* One entity occurrence does not require a corresponding entity occurrence in a particular relationship
* Mandatory:
* One entity occurrence requires a corresponding entity occurrence in a particular relationship
* Notice the requirements. It’s may be mandatory or optional depend on the requirements.
* Foreign Key (FK):
* It’s a primary key from the other entity which you are linking
* FK always goes to many side
* Strong and Weak entities:
* Strong entity:
* An entity type that exists independent of other entity types
* can be identified by its own attributes, meaning a unique identifier can be chosen from its own attributes
* Strong entities are connected by non- identifying / weak relationships, represented by dashed line in Crow’s foot
* Weak entity:
* It cannot exist without another entities,
* They don’t have their own primary key; its primary key will consist of something derived from another entity
* Like parent and child, and usually have one to many relationship
* Weak entity can have another weak entity child
* Weak entity set must have mandatory participation in this relationship set
* Depicted through primary keys and solid line for the relationship
* Such relationship is called identifying or strong relationship
* Weak entity set must have mandatory participation in this relationship set
* It has a composite PK, part of which is borrowed from another entity. The borrowed part of the identifier is PK of another entity and becomes a Foreign Key (FK) in the weak entity. It is called Primary Foreign Key (PFK)
* How to identify weak entity: solid relationship line, PFK
* M:M and Weak entities:
* Many to many is a conceptual model, in logical and physical cannot represent many to many. So, we need to solve M:M
* Replace M:M by two 1:M
* Insert a weak entity to resolve M:M, and then add the primary key in both M:M to the weak entity which we created.
* And then check if the PFK in this weak entity can make primary key unique. If not, we need to add another primary key for this weak entity.
* Borrowing FK for PK
* Relationship degree:
* Indicated number of associated entities or participants
* Unary relationship: association is maintained within a single entity, also known as recursive relationship
* Binary relationship: two entities are associated
* In general, we can have n-ary relationships
* Developing an ER Diagram
* Iterative Process
* 1. List the major entities in the system.
* 2. Represent the entities graphically by a rectangle.
* 3. Search for relationships between the entities and represent them graphically with the proper symbol (e.g. a diamond if Chen’s notation is used).
* 4. Add attributes; remember to establish the primary key for every entity.
* 5. Model relationship connectivity between each pair of entities.
* 6. Model relationship cardinalities between each pair of entities (i.e. the minimum and maximum participation).
* 7. Determine whether there are weak entities. Refine M:M relationships. If required, refine the connectivity and cardinality of entities affected.
* 8. Verify the ERD you have created by going through each component you have created from Steps (1) to (7). Ensure that they properly represent the business rules of the system you are developing the database for.
* Relational database: a set of relations (definition)
* Relation make up of 2 parts:
* Schema: specifies name of relation, plus name and type of each column (attribute).

Example: Pizza(pID: string, pizzaName: string, price: real)

* Instance: a table, with rows and columns.

Number of rows = cardinality, Number of columns/fields = degree (or arity)

* all rows are distinct
* no order among rows
* The entire cycle:
* 1. Conceptual Design

Remove M:M, and only words

* 2. Logical Design

Physical need data type

* 3. Physical Design

Converting physical design into SQL and then start adding rule, like primary key

* 4. Implementation
* 5. Create instance

Put the value in

* Keys and integrity constraints
* Keys are a way to associated tuples in different relations
* Keys are one of integrity constraint (IC)
* Superkey:
* A set of fields is a superkey if no two distinct tuples can have the same values fields
* The set of all attributes is always a superkey
* A set of fields is a key for a relation if it is aa superkey and no subset of the fields is a super key (minimal subset, can’t be made smaller)
* Primary keys:
* Out of all keys one is chosen to be the primary key of the relation.
* Other keys are called candidate keys
* Each relation has a primary key (we cannot choose the attributes that will change to be the primary key)
* Selecting the primary Key
* Superkey – a set of fields that contains the key
* Candidate keys are all the possible key combinations that could be the Primary Key
* Of all candidate keys the database designer identifies the primary key. The primary key is the fewest number of columns that can uniquely identify a key. It should be an attribute that will not change throughout the life of the entity.
* N.B.\* Not all relations will have a candidate key. In cases when there is no candidate key, the database designer will add a surrogate key
* Surrogate keys
* A key that has no real world/ business meaning
* Is usually numeric
* Many databases around the world have been created with all their tables using surrogate keys.
* When adding a surrogate key – you don’t lose any data (the original PFK might be changed to FK, but not disappear)
* Natural key
* Foreign keys and referential integrity:
* If all foreign key constrains are enforced in a DBMS, we say a referential integrity is achieved.
* Integrity constraints (IC)
* Condition that must be true for any instance of the database
* This is known as schema on write
* Multi-valued attributes in logical design
* Multi-valued attributes need to be unpacked (flattened) when converting to logical design.
* Connectivity constraint rule:
* Primary key from the one side becomes a foreign key on the many side
* On delete no action and on delete cascade:
* On delete no action: generally, happen in two strong entities. When we want to delete the data on the side which is not many side (which is not the FK putting side), the action will be stop.
* On delete cascade: Generally, happen in one strong related to one weak entity. When we want to delete the data in the strong entity, the data in the weak entity also need to be deleted.
* One-to-one relationship
* In one-to-many situation, FK goes to many side
* Need to decide whether to put the foreign key inside which side
* The rule is the optional side of the relationship gets the foreign key
* Where would the least null value be
* Unary relationship:
* A unary relationship is when both participants in the relationship are the same entity
* In implementation, the foreign key needs to add on delete restrict and on update cascade
* Operate in the same way as binary relationship
* One-to-one
* One-to-many
* Many-to-many: need the associative entity, put two foreign in the associative entity. In this case, the associative entity has two primary foreign keys. So, need two on delete restrict and on update cascade.
* Choosing data type:
* Choose the correct data type:
* Can represent all possible values
* Minimize storage space
* Maximize performance
* Character types:
* Char(M): fixed length. It would have space, if not occupy all position. The range of M is 1 to 225
* VARCHAR(M): a variable-length string. Only the characters inserted are stored- no padding. The range of M is 1 to 65535 characters.
* Number types:
* Integer:

TINYINT, SMALLINT, MEDIUMINT, INT/INTEGER, BIGINT, BIT

* Real numbers (fractions)

FLOAT, DOUBLE, DECIMAL

* Boolean can be represented as TINYINT or BIT
* Don’t use (M) number for integers
* Date time types:
* DATE, TIME, DATETIME, TIMESTAMP, YEAR
* Normalization: without normalization, there will be too many duplicate data in the table
* Anomalies in Denormalized data:
* Update anomaly: when you update one of the same two data, but not update another one. It will make people reading the data confuse. An update anomaly is a data inconsistency that results from data redundancy and partial update when one or more instances of duplicated data are updated but not all.
* Deletion Anomaly: sometime when you delete the value, you may at the same time delete the data in the same row. A deletion anomaly is an unintentional loss of certain attribute values due to the deletion of other data for other attributes.
* Insertion data: If there are two primary key data in the table, you cannot add the one of the primary key data without another one. An insertion anomaly is the inability to add certain attributes to a database due to absence of other attributes.
* Functional dependency:
* A functional dependency concerns values of attributes in a relation
* The attribute Y is fully functionally dependent on the attribute X if each value of X determines one and only value of Y
* Notation: X → Y
* Determinants (X,Y→Z) – the attribute(s) on the left hand side of the arrow
* Key and Non-Key attributes – each attribute is either part of the primary key or it is not
* Partial functional dependency (Y→Z) – a functional dependency of one or more non-key attributes upon part (but not all) of the primary key, e.g. (StudentID, SubjectCode) -> StudentName
* Transitive dependency (Z→D) – a functional dependency between 2 (or more) non-key attributes, e.g SubjCode|SubjectName|LecturerID|LecturerName results in LecturerID->LecturerName
* Functional dependencies can be identified using Armstrong’s Axioms
* Reflexivity:

B is a subset of A, means A functionally determines B

* Augmentation: A->B 🡺 AC-> BC
* Transitivity: A->B and B-> C🡺 A-> C

If A functionally determines B and B functionally determines C, then A functionally determines C

* Normalization process
* First normal form (1NF)
* Remove repeating groups
* Identify Pk for each resulting relation
* For the removed repeating group, give the new relation a name, link it to the “leftover” relation with FK and identify PK (because the 1NF is to remove the repeat field, which is one to many. So, we have to add the FK to the repeat group and identify the PK)
* Second normal form(2NF)
* This step is relevant only for relations whose primary key is a COMPOSITE key (PK consisting of more than one attribute)
* Remove partial functional dependencies, where an attribute is dependent on only part of the primary key (i.e., not dependent on all the columns in a composite primary key)
* Create a separate relation, containing the functionally dependent data, and the part of the key on which it depends
* 2nd always create PFK on the original relation (set)
* Third normal form (3NF)
* Remove transitive dependencies, which are functional dependencies where one non-key attribute is functionally dependent on another non-key attribute. Thus, its value is only indirectly determined by the primary key.
* Create a separate relation containing the determinant attribute and the fields that are functionally dependent on it.
* Keep a copy of the key (FK) in the original relation and create a PK in a new relation
* Business transaction:
* Example:
* Placing order: Insert one row in Order table, then several in OrderItem table
* Funds transfer: Check amount < balance. If so, subtract amount from one row in BankAccount table, then add amount to another row. Record details in Transactions table
* Sending monthly statements: For all rows in Customer table, generate and send out monthly statements
* Each requires several distinct database operations
* Database transaction:
* A logical unit of work that must either be entirely completed or aborted (indivisible, atomic)
* These are a sequence of DML statements, such as
* a series of UPDATE statements to change values
* a series of INSERT statements to add rows to tables
* DELETE statements to remove rows
* Transactions will be treated as atomic
* A successful transaction changes the database from one consistent state to another
* All data integrity constraints are satisfied
* Transactions solve two problems:
* Users need the ability to define a unit of work
* Concurrent access to data by > 1 user or program
* A transaction should not include tasks that are not relevant to each other
* Problem 1 of transaction: Unit of work:
* Single DML or DDL command (implicit transaction)
* Example: update 700 records, but database crashes after 200 records processed
* Restart server: you will find no changes to any records
* Rollback every change -> there is a checkpoint. So, you don’t need to roll back every data.
* Changes are “all or none”
* – Multiple statements (user-defined transaction)
* START TRANSACTION; (or, ‘BEGIN’)

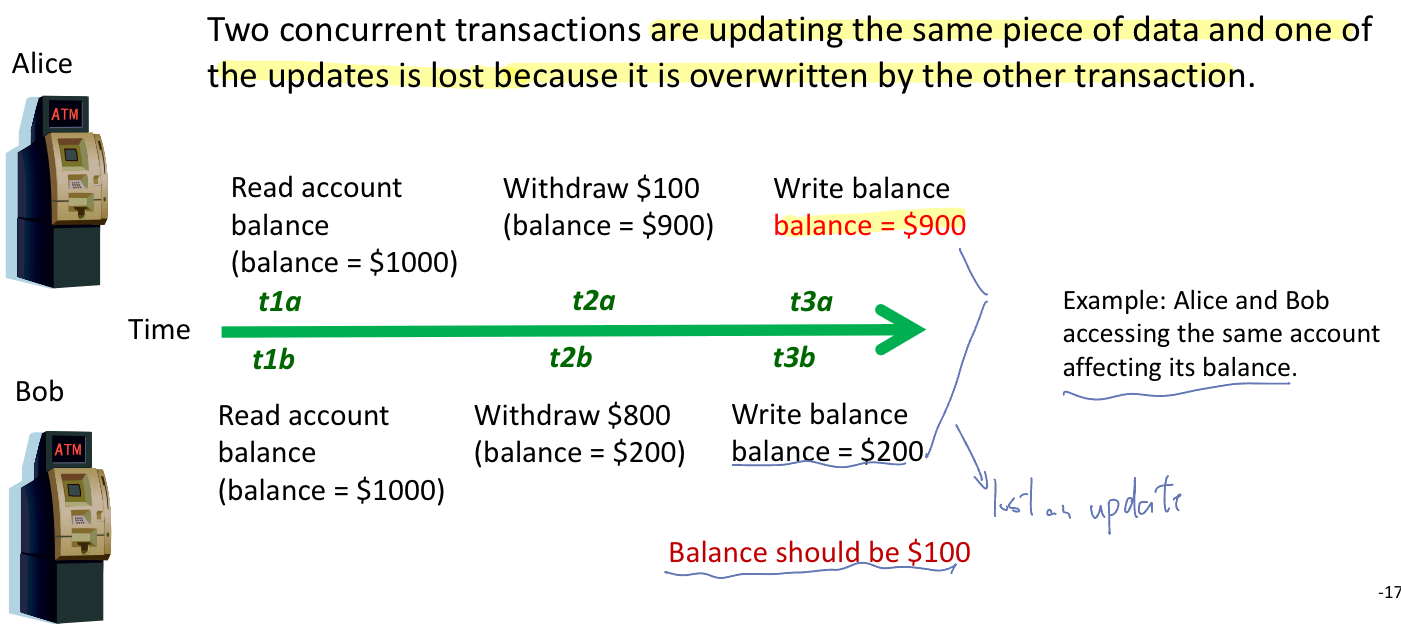
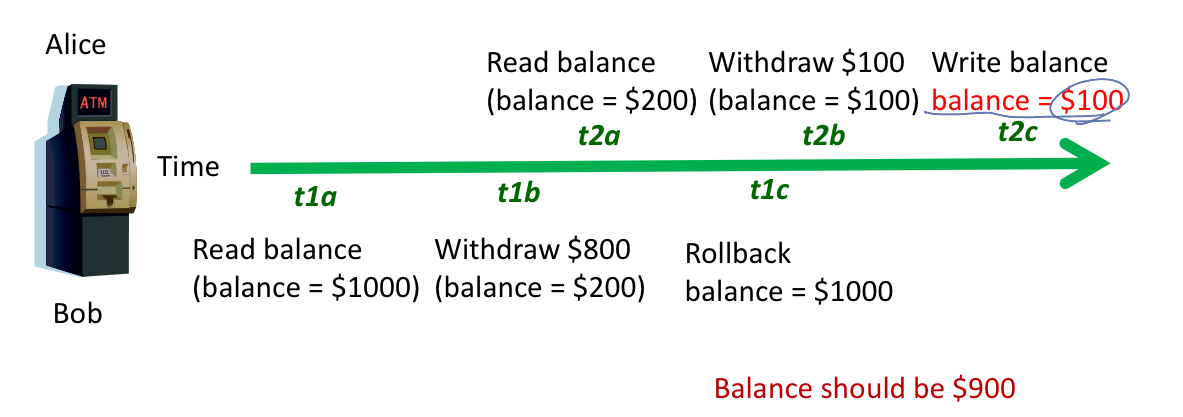
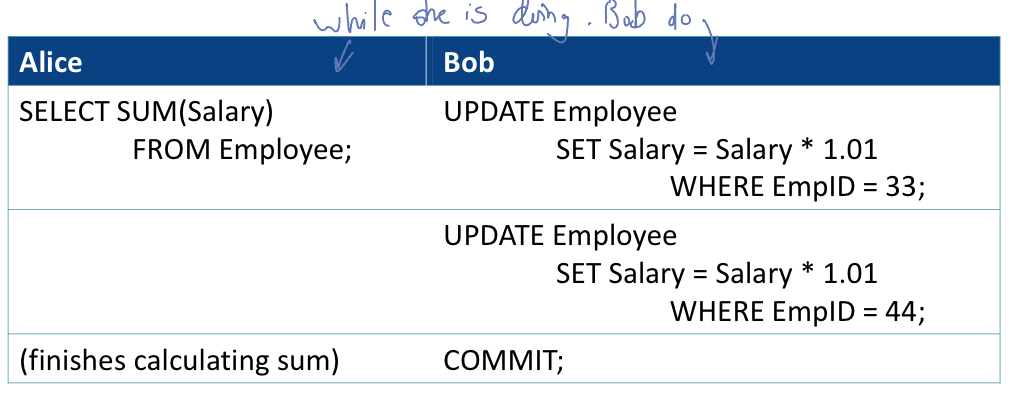
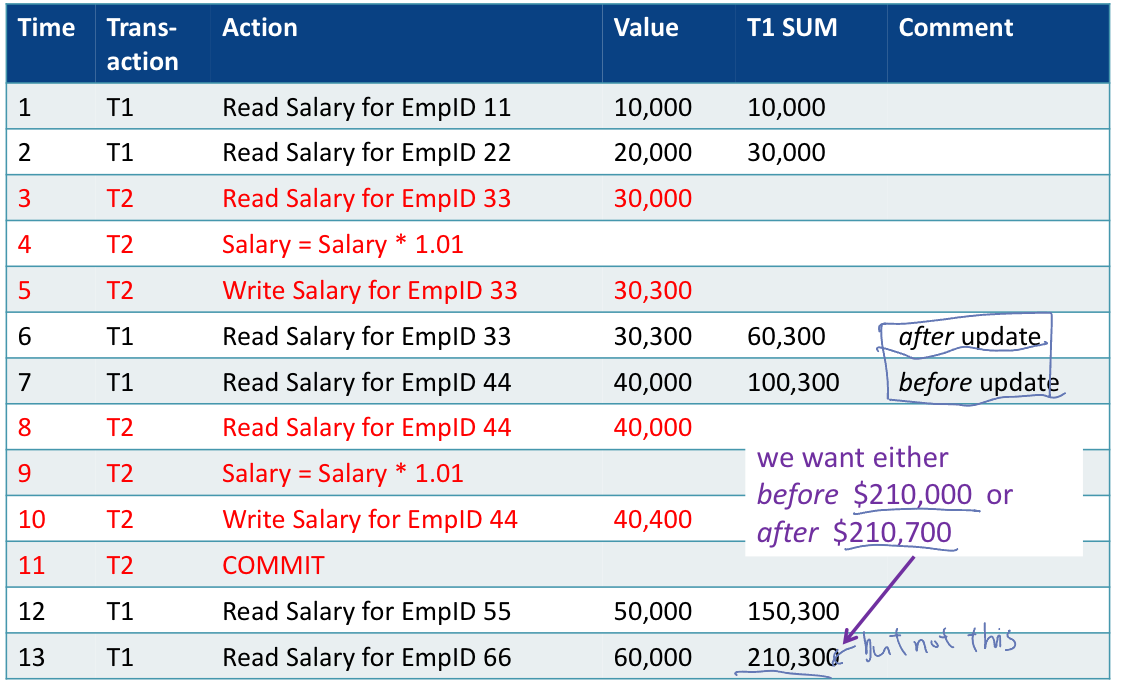
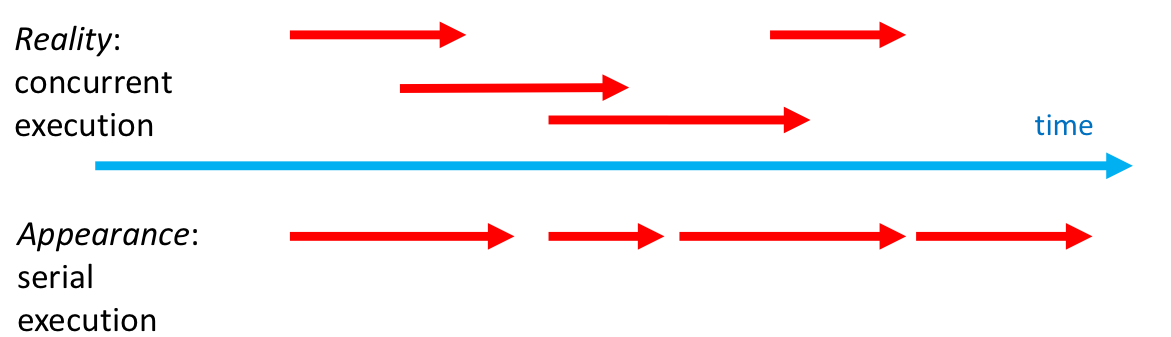
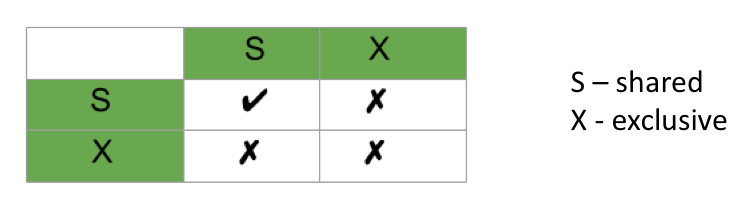
– SQL statement;

– SQL statement;

– SQL statement; –…

* COMMIT; (commits the whole transaction, i.e. saves changes permanently)

– Or ROLLBACK (to undo everything)

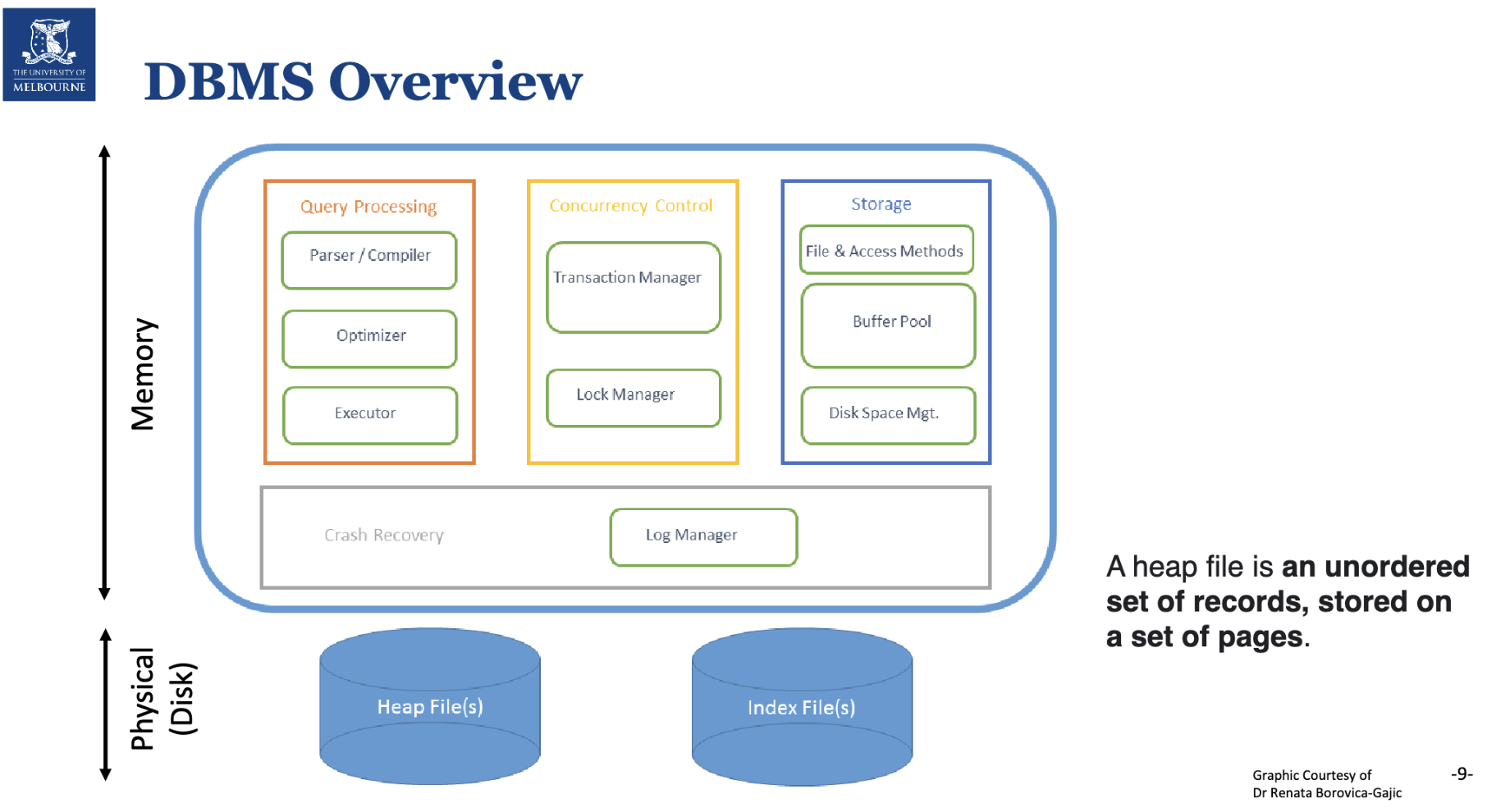
* SQL keywords: BEGIN; START TRANSACTION; COMMIT, ROLLBACK
* Business transactions as units of work:
* Transaction needs to be an indivisible unit of work
* Indivisible means that either the whole job gets done, or none gets done
* If an error occurs, we don’t leave the database with the job half done, in an inconsistent state.
* Transaction properties: (ACID)
* Atomicity
* A transaction is treated as a single, indivisible, logical unit of work. All operations in a transaction must be completed; if not, then the transaction is aborted
* It’s all or none.
* Consistency
* Constraints that hold before a transaction must also hold after it. A transaction must not violate any integrity constraints or rules that govern the database.
* (multiple users accessing the same data see the same value)
* Isolation
* Multiple transactions can occur concurrently without leading to the inconsistency of the database state. The intermediate states of transactions are invisible to other transactions until they are committed. The data required by an executing transaction cannot be accessed by any other transaction until the first transaction finishes. (This ensures Consistency)
* Durability
* When a transaction is complete, the changes made to the database are permanent, even if the system fails. The updates are permanent and are stored in non-volatile memory, such as a hard disk or solid-state drive.
* Problem 2 of transaction: concurrent access
* Concurrent execution of DML against a shared database
* Note that the sharing of data among multiple users is where much of the benefit of databases derives – users communicate and collaborate via shared data
* if we have multiple users accessing the database at the same time…
* something might happen:
* lost update
* uncommitted data
* inconsistent retrievals
* lost update:
* Two concurrent transactions are updating the same piece of data and one of the updates is lost because it is overwritten by the other transaction.
* 
* Uncommitted data problem
* Uncommitted data occurs when two transactions execute concurrently and the first is rolled back after the second has already accessed the uncommitted data
* 
* The inconsistent retrieval problem:
* Occurs when one transaction calculates some aggregate functions over a set of data, while other transactions are updating the data
* Some data may be read after they are changed and some before they are changed, yielding inconsistent results
*  
* Serializability:
* Transactions ideally are “serializable”
* Multiple, concurrent transactions appear as if they were executed one after another
* Ensures that the concurrent execution of several transactions yields consistent results
* Scheduler – special DBMS process that schedules in which order operations within concurrent transactions execute, ensuring serializability and isolation.
* 
* Concurrency control methods:
* Locking (the main methods used)
* Time stamp
* Optimistic methods
* Lock:
* Guarantees exclusive use of a data item to a current transaction
* T1 acquires a lock prior to data access; the lock is released when the transaction is complete
* T2 does not have access to data item currently being used by T1
* T2 has to wait until T1 releases the lock
* Required to prevent another transaction from reading inconsistent data
* Lock manager:
* Responsible for assignment and policing the locks used by the transactions
* Lock granularity:
* Database-level lock
* Entire database is locked
* Good for batch processing but unsuitable for multi-user DBMSs
* T1 and T2 cannot access the same database concurrently even if they use different tables
* Table-level lock:
* Entire table is locked - as above but not quite as bad
* Good when updating most of the rows in a table
* T1 and T2 can access the same database concurrently as long as they use different tables
* Can cause bottlenecks, even if transactions want to access different parts of the table and would not interfere with each other
* Not suitable for highly multi-user DBMSs
* Page-level lock: (most frequently used)
* An entire disk page is locked (a table can span several pages and each page can contain several rows of one or more tables)
* So, only part of the table is locked
* Row-level lock:
* Allows concurrent transactions to access different rows of the same table, even if the rows are located in the same page
* Improves data availability but with high overhead (each row has a lock that must be read and written to)
* Higher overhead than Page level lock
* Automatically escalated to page level if several rows in the same page need to be locked.
* Field-level lock: (like lock certain column)
* Allows concurrent transactions to access the same row, as long as they access different attributes within that row
* A user may see some columns but not others (e.g. in online banking transactions listed but running balance not available)
* Most flexible lock but requires an extremely high level of overhead
* Not commonly used
* Types of locks: (binary locks)
* Has only two states: locked (1) or unlocked (0)
* Every transaction will apply a lock and then terminate a lock (release the locked data)
* eliminates “Lost Update” problem
* the lock is not released until the statement is completed
* considered too restrictive to yield optimal concurrency, as it locks even for two READs (when no update is being done)
* The alternative is to allow both Shared and Exclusive locks
* often called Read and Write locks
* Shared and Exclusive locks:
* exclusive lock:
* Access is reserved for the transaction that locked the object
* Must be used when transaction intends to WRITE
* Granted if and only if no other locks are held on the data item
* In MySQL: “SELECT … FOR UPDATE” or “LOCK TABLES … WRITE”;
* Shared lock:
* Other transactions are also granted Read access
* Issued when a transaction wants to READ data, and no Exclusive lock is held on that data item
* – multiple transactions can each have a shared lock on the same data item if they are all just reading it
* In MySQL: “SELECT … LOCK IN SHARE MODE” or “LOCK TABLES … READ;
* Basic lock compatibility matrix:
* 
* In simple words, if transaction T1 is reading a data item A, then same data item A can be read by another transaction T2 but cannot be written by another transaction.
* Similarly, if an exclusive lock (i.e. lock for read and write operations) is held on the data item in some transaction then no other transaction can acquire a lock, neither Shared nor Exclusive lock.
* Strict two-phase locking
* Two Phases
* A transaction T must obtain all locks before it can start reading or writing data (growing phase). No unlocking can happen if another lock is required.
* All locks held by transaction T are released when the transaction is completed (shrinking phase).
* If two transactions are accessing completely different parts of the database, they concurrently obtain the locks and proceed.
* If two transactions T1, T2 are accessing the same object and one wants to modify it, their actions are ordered serially
* Transaction isolation levels:
* isolation levels define the degree to which the transactions are isolated from each other, impacting data integrity and concurrency. Among these isolation levels, Read Uncommitted, Read Committed, and Repeatable Read represent different strategies.
* Isolation levels provide the level any given transaction is exposed to the actions of other executing transactions
* Purpose – increase transaction concurrency
* Read uncommitted
* Does not obtain shared locks before reading, which increases performance but at the cost of data consistency
* May read uncommitted data from other transactions (dirty read). A transaction could be reading data that may become non-existent because the other transaction that was updating the data rolled-back the changes.
* No isolation between transactions

(Sometime we will violate the isolation, in order to get the things that we want in faster progress

* Dirty Reads: This level allows dirty reads. A dirty read occurs when a transaction reads data that another transaction has modified but not yet committed, and if the modifying transaction rolls back its changes, the first transaction will have read data that was never technically committed.
* No Protection Against Any Concurrency Problems: Apart from dirty reads, read uncommitted transactions are also susceptible to non-repeatable reads and phantom reads.
* Read committed
* Only reads committed changes by other transactions
* No value written by T1 is changed by another transaction until T1 completes (commits)
* However, a value read by T1 can be modified by other transactions Tn while T1 is in progress.
* Read committed holds exclusive locks (x) before writing objects and shared locks(s) before reading objects.
* No Dirty Reads: The read committed level prevents dirty reads, ensuring more reliable and consistent data than read uncommitted.
* Non-repeatable Reads Possible: This level does not prevent non-repeatable reads, where during a transaction, a row can be retrieved twice and return a different value each time if another transaction modifies and commits a change to that row in between these reads.
* Phantom Reads Possible: Similar to non-repeatable reads but involves new records appearing between reads in the same transaction.
* Repeatable read
* ensures that if a transaction reads a row, any subsequent reads of that row within the same transaction will see the same data as was read the first time, regardless of changes made by other transactions.
* A transaction running in this isolation level does not consider any changes to data made by other transactions, regardless of whether the changes have been committed or not.
* Prevents Dirty and Non-repeatable Reads: At this level, once a row is read by a transaction, other transactions cannot modify or delete that row until the first transaction is complete.
* Phantom Reads Possible: While this level prevents modifications or deletions of the rows read by the ongoing transaction, it does not necessarily prevent the insertion of new rows. Therefore, the transaction may experience phantom reads where additional "phantom" rows might appear in subsequent queries within the same transaction.
* Serializable: the highest degree of isolation
* Locks are obtained for both reads (R) and writes (W)
* Transaction T only reads committed transactions
* Similar to Repeatable Read, with the additional restriction that a row selected by transaction T1 cannot be changed by T2, T3, … until T1 finishes.
* Deadlock:
* Condition that occurs when two transactions wait for each other to unlock data
* T1 locks data item X, then wants Y
* T2 locks data item Y, then wants X
* Each waits to get a data item which the other transaction is already holding
* Could wait forever if not dealt with
* Deadlock can only happen with exclusive locks
* Deadlocks cannot happen with shared locks
* Dealing with deadlock:
* Timeout mechanisms
* Prevention: preventing deadlocks by careful design of transactions, such as always acquiring locks in the same order or releasing locks as soon as possible.
* Detection
* Avoidance: avoiding circular wait conditions
* Alternative concurrency control methods
* Timestamp
* Assigns a global unique timestamp to each transaction
* Each data item accessed by the transaction gets the timestamp
* Thus, for every data item, the DBMS knows which transaction performed the last read or write on it
* When a transaction wants to read or write, the DBMS compares its timestamp with the timestamps already attached to the item and decides whether to allow access
* Optimistic:
* Based on the assumption that the majority of database operations do not conflict
* Transaction is executed without restrictions or checking
* Then when it is ready to commit, the DBMS checks whether any of the data it read has been altered – if so, rollback
* Logging transaction:
* If transaction cannot be completed, it must be aborted and any changes rolled back
* To enable this, DBMS tracks all updates to data
* This transaction log contains:
* a record for the beginning of the transaction
* for each SQL statement
* the ending (COMMIT) of the transaction
* If a system failure occurs, the DBMS will examine the log for all uncommitted or incomplete transactions and it will restore the database to its previous state.
* Database Administrator (technical role)
* analyze and design DB
* select DBMS / tools / vendor
* install and upgrade DBMS
* tune DBMS performance
* manage security, privacy, integrity
* backup and recovery
* Data administrator (CDO/ CSO) (management role)
* Data policies, procedures and standards
* Compliance with legislation (EU GDPR, AUS Privacy Act)
* Compliance with company policy (e.g. Uni of Melbourne privacy policy)
* Planning
* Data conflict resolution
* Conflict resolution is often possible in customer management systems. For example,
* salespeople can maintain customer address information at different databases in a replication environment. Should a conflict arise, the system can resolve the conflicting updates by applying the most recent update to a record
* Managing information repository:
* A corporate repository is a place where a large amount of a corporation's information is stored, typically for the long term. Corporate repositories are not made public. Only users with specific roles and permissions granted may access documents
* Internal marketing and education:
* Internal marketing is the promotion of a company's objectives, products and services to employees within the organization. The purpose is to increase employee engagement with the company's goals and foster brand advocacy.
* Architecture of a database management system (DBMS)
* A database management system exists as one entity in two places:
* In memory
* Physically on disc
* Program sitting in the hard disc, but it runs in the memory (RAM, CPU)
* Both places manage:
* Data (the reason we have the DBMS)
* Performance (how it performs as it is used and grows)

(how quality your database work and the program process data)

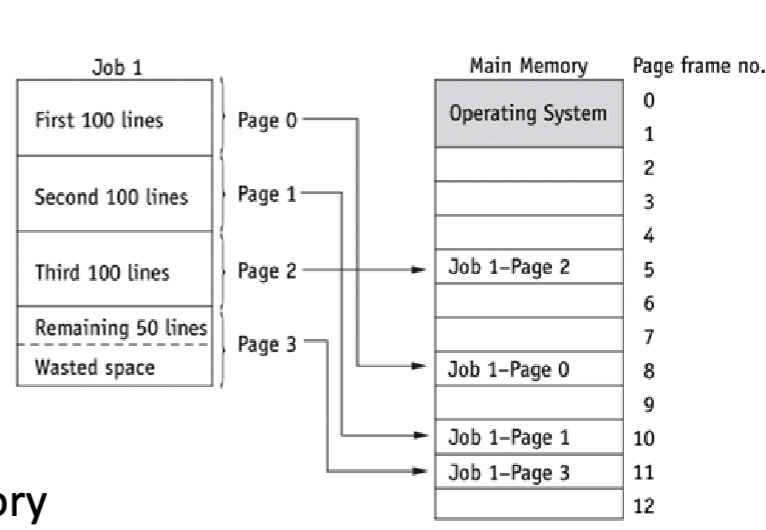
(and how to find the data in the disc)

* Concurrency (manages high volumes of users)
* Recoverability (assist in recovery and availability)
* One place is persistent, and other id transient
* Disk representation is always present
* Memory is transient, only exists when DBMS is running
* DBSM overview:
* 
* Quick notes:
* A heap file is an unorder set of records, sorted on a set of pages
* Heap file: big file. When they’re needed for processing, it will split into pages and stored in virtual memory but still hold on the hard disk.
* Index files: to speed up the search in the disk
* The table is so big that we cannot put them all in the memory (only relevant part put into memory), so we need index to speed up finding process to get our data
* For the storage part (blue one above). They need the program to talk to data on the hard disc. Operating system need to know how to find the data on the hard disk.
* Query processing: the orange one in the diagram above
* Parsing:
* Syntax is correct and can compile (syntax analysis)
* DBMS user permissions. (permission to the access)
* Resources (Data, Code, ability to Record/Change/Retrieve results)
* Optimizing:
* How to run the program faster
* This will check if the table is too big. If it needs the index or not.
* Which algorithm ask to use
* Execution Plan and Execution Cost
* Evaluate indexes, table scans, hashing
* Eliminate worst, consider best options
* Lowest cost theoretically “best”
* Execution:
* Meet the ACID (atomicity, consistency, isolation and durability)
* Atomic: all rows succeed, or all fail
* Ensure resource are available: Data, Log changes, Memory, Cursor to do the work for the USER
* Query processing -SQL Parsing Phase
* The optimization process includes breaking down the query into smaller units
* The original SQL query is transformed into slightly different version of original SQL code which is fully equivalent and more efficient
* SQL parsing is performed by the query optimizer, which analyzes the SQL query and finds the most efficient way to access data
* An access plan is the result of parsing a SQL statement; it contains a series of steps the DBMS will use to execute the query and return the result set in the most efficient way
* If an access plan exists for the query in SQL cache, the DBMS reuses it
* If there is no access plan, the optimizer evaluates various plans and chooses one to be placed in SQL cache for use
* Concurrency control:
* Manages the work of the DBMS
* Transaction manager handles all aspects of the SQL transaction -which DBMS user/ process wants which resource
* Lock manager maintains a list of what resources are locked and by which user at what level (and who is waiting). It will schedule, which transaction goes in which order.
* Concurrency control also check what is in the buffer or what is on the hard disk and virtual memory
* Essential to manage large scalable DBMS
* Which transactions have completed, in progress, compiled.
* What resources are involved with that transaction
* Who last used, is using and wants those resources – SQL, Cursor, Index, Table, Rows, File Access, Recovery Logs
* Storage:
* File and Access Methods
* Disk to Memory to Disk
* Read a buffer (Database processes request pages from the buffer manager)
* Buffer Pool
* a main-memory area used to cache database pages
* Data in memory

– Row data

– Index data

(must stay in memory -> index: for fast search for table)

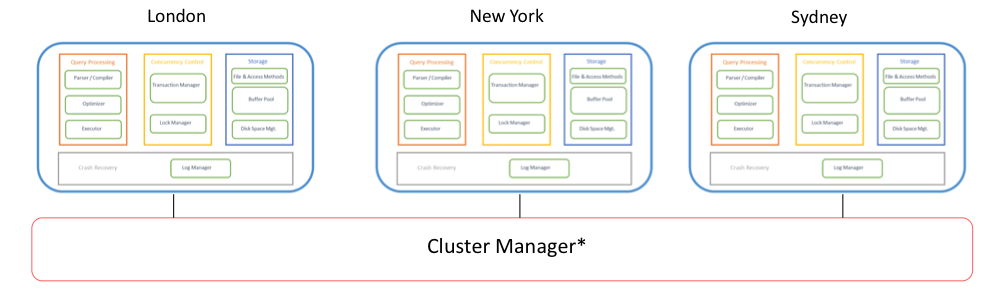
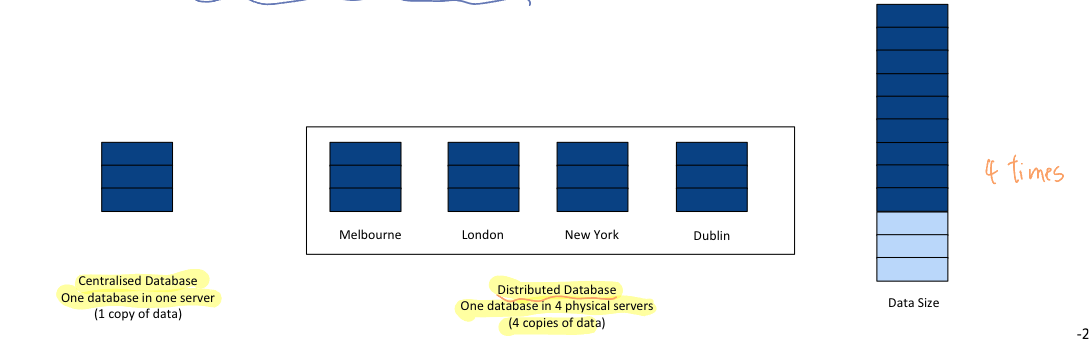
* Buffer manager minimizes the number of secondary memory accesses by keeping needed pages in the buffer
* If the data is not in the buffer, it need to be reference from hard disk -> cost time. This is what optimizing tries to reduce read and write operation.
* For write operation: cannot be eliminated because if transaction is complete result needs to be saved on the hard disk for durability.
* For read operation: can be reduce. If same part of table is needed frequently, might better keep in the buffer.
* Disk Space Management
* How to organize growth of data on disk efficiently by writing efficiently.
* Paging:
* Big file -> in memory split into pages and only core pages are sitting in the memory permanent, others swaps in and out.
* A technique for memory organization
* Programs that are too big to fit on a single page or take too much memory as a whole are split into equal size blocks
* Paging allows programs to use non-contiguous memory
* It is not necessary for the whole program to be in memory
* To use paging, the OS memory manager:
* Breaks physical memory into Frames
* Breaks logical memory into Pages of same size
* Uses a Page Table to link Pages to Frames
* 
* Files, pages and frames
* A process consists of a program, associated data and the process context
* How to access the file: there are different algorithm, program know how to deal with it.
* Full scan (full table scan)
* Read the entire table sequentially, form the first row to the last low, one row at a time
* Partial scan (index range)
* Index use to search -> faster than the above one
* Reads the index first to obtain the rowIDs and then accesses the table rows directly
* Using an index to scan a range of values
* Storage – buffer pool and Disk space:
* Buffer pool: (is part of memory allocated to database system management)
* table data and indexes are read from disk into the buffer
* May contain multiple copies of the same data -> like dirty data of transaction, you commit but not finished it can roll back and not to save
* What is keep in buffer is what actually being working
* Each buffer contains rows, pages, etc.
* Each buffer can have one of four status types:
* Current: in use current committed version of data
* Active: most recent change (may not be committed)
* Stale: an old version of the data
* Aged: old and about to be removed from buffer pool
* Disk space management:
* How to allow files to grow on disk, where data or index need to be saved
* File organization:

– e.g. index reorganization;

– varchar growth,

* Log manager:
* Recovery
* Log Manager records ALL changes Crash Recovery
* Statement
* Transaction
* – Statement (update, select …..)
* – Rollback values
* – Before and After values
* – Timestamp begin: transaction, savepoint\* and commit timestamps (when commit happen)
* Database
* – Data Dictionary Changes
* How a transaction work:

1. The code is parsed and optimized. Permissions for data access are checked. (parsing code -> check syntax is correct -> permission is available -> optimize the algorithm (to find the best way))
2. DBMS checks if the table and index data are in the buffer pool (check if already in the buffer pool -> yes: skip communication with the hard disk; no: talk to the hard disk first)
3. The data is read up from the table and index on disk (If it is heap file, read it from here. If not easy to find, use index to find it faster)
4. The transaction manager ensures all the tables and index is in the buffer pool and ready to be used in the execution of the SQL
5. When the rows and indexes are locked, execute the query (lock for no lost update)
6. (a) In the buffer pool record the before value, the after value and the change to the value. (b) record a timestamp and the before value, the after value and the change to the value for all transaction in the Log Manager
7. The change is committed to make it permanent
8. The changed data and transaction log are written to disk

* What affects database performance:
* Caching data in memory, ex: data buffers (data file access)
* Our goal is to minimize reads and write
* Avoids a read from Disk, better to read in buffer pool
* Reads are expensive
* “in memory databases”
* All code and all data loaded into memory on database start and stays until shutdown
* Help with increasing performance demands of modern database applications
* Placement of data files across disc drives (multi disc is slower -> we should make more space on disc)
* Fast reliable storage (what kind of hard disc we used)
* Database replication and server clustering
* Use of indexes to speed up searches and join
* Good choice of data types (especially Primary key, which prefer numeric)
* Good program logic (no long running CRUD) CRUD: Create, Read, Update, and Delete
* Good query execution (avoid to use subquery, which is slower)
* Good code (no deadlocks)
* Distribution and replication:
* Distributed data: (put the database across multi-servers)
* Spreads the load
* Data kept only where it is needed
* faster response time in application
* Only core parts of data are overlap
* Replicated Data:
* For reliability -> if one crashed, there still have data in another server
* May get inconsistent database
* Less work per physical server- read data in replicated data is faster than distribution data.
* Database replication is the frequent electronic copying of data from a database in one server to a database in another
* Good execution plans:
* The best execution plan has the lowest “cost” Known as
* Cost Based Optimization (CBO)
* Position of where clause is important.
* Create the index also helpful
* Sequential record access:
* Read each record one by one from the beginning of the table until the required record is founded
* Indexing
* The more rows we have the slower the sequential search we have.
* Index will have every row but only two columns
* Index value: record number meaning like PK
* Location: where it’s on the disk
* Inserting data into a simple index:
* When a table has a primary key, an index based on that primary key is automatically created. (done by default)
* As each row is added, the DBMS automatically inserts the search key and the row location into the index file
* In the index file, you can sort the table in the order you want
* Searching the index:
* Index is much smaller than the table data, and searching index is quicker than sequentially searching table
* By using binary search, which eliminated half of data each time
* Once a match has been found, the RDBMS can quickly retrieve the appropriate record
* Advantages of using an index:
* Reduce input/ output which reduced communication with hard disc
* Index can be stored in memory because index table is small only two columns
* Using an index is a quick method to locate rows
* Additional information for index:
* A binary search is only one type of search method available to a DBMS
* It is not possible to display the contents of an index
* Indexes may be stored in a non-human readable format or compressed format to reduce their size and improve search speeds.
* Index does not contain a record number, the record number we see in the datasheet doesn’t really exist
* Index speed up report generation:
* Create foreign key speed up operation
* Creating indexes on fields central to the report (e.g. used in WHERE and HAVING clauses) have often reduced report generation time from many hours to a few minutes.
* but why every table have index -> because it’ll slow down the operation, especially when you need to update some data. Index also need to be updated. Another reason is that it takes space.
* A multi-level index;
* Some large indexes may split into smaller index segments.
* An index entry is created for all segments. It’s an index of indexes.
* When to create index:
* For each table, choose the columns you will index:
* queried frequently (used in WHERE clauses)
* used for joins (PK to FK)
* primary keys (automatic in most DBMS)
* foreign keys (automatic in MySQL)
* unique columns (automatic in most DBMS)
* Large tables only - small tables do not require indexes
* if you frequently retrieve less than about 15% of the rows
* Wide range of values (good for regular indexes).
* Small range of values (good for bitmap/hash indexes).
* Centralize server is good for local company, but not good for global company. For global company is better to use distribution database.
* Distribution Database:
* a single logical database physically spread across multiple computers in multiple locations that are connected by a data communications link
* appears to users as though it is one database
* logical database, which is spread across multiple servers geographically and physically. And replicated only data which is popular all over the world-> not every data will be replicate, it is by company decision.
* For distribution database, the replication data has updated, other replicated date will also be updated. So, the database is consistent.
* Decentralized Database
* a collection of independent databases which are not networked together as one logical database
* appears to users as many databases
* Separate database. They’re not talking to each other’s. If I made an update in location one, it will not update in other location.
* Distributed memory structures:
* 
* Each Physical Server has one of these memory structures
* Often accessing their own and shared physical storage between all physical servers
* Cluster Manager coordinates communication between physical servers including Query Processing areas, Multiple Concurrency Control, Storage, Crash Recovery
* Distributed DBSM advantages:
* Allow modular growth, add new servers as load increase
* Because it’s distributed database. It’s easy to add more server
* Increase reliability and availability of data
* Lower danger of a single point of failure (SPOF)
* Supports database recovery
* Data may be replicated across multiple sites
* Recovery logs are always replicated.
* Because it’s a distributed database. Even if you don’t replicated data across all servers, you keep only relevant data on each sever. The logs files are everywhere, they are replicated, these files are small. So, if London server crashed, you can get the data (log file) from Paris server and restore London server.
* Disadvantage of distributed DBMS
* Complexity of management and control: this means that your data, like response to every query, needs to be collected together from different servers from different sites. This may cause the problems below:
* Which one is the latest version -> what if one answer is update an another one hasn’t yet
* Who is waiting, and where are they
* How does the logic display this to the web and application server?
* Data integrity:
* Additional exposure to improper updating (especially data across severs)
* If two users in two locations update the record at the exact same time who decides which statement should “win”?
* Solution: Transaction Manager or Master-slave design
* Transaction manager: it decides who, when to get the access first
* Master-slave design:
* Master is the server which hold the latest update
* When the writing happened. It happens at the master sever (for update) only.
* When someone need to read data (like create some report) it can happen at any slave server.
* However, it could also mean that recording data from slaves are still dealing with the old version of record.
* So, we will need synchronous update, meaning that as soon as masters updated, it sends update to all slaves. But this means it has to freeze all slaves for responding to query while they’re updating data.
* A synchronous update means that master is update, but it waits for slow down (less user) in traffic.
* Security:
* Many server sites -> higher chance of breach (break, hacker)
* Multiple access sites require protection from both cyber and physical attacks
* Lack of standards:
* Different relational DDBMS vender use different protocols
* Challenge with interoperability (互操作性): these day we often go into cloud for distributed database system (like outsource Microsoft, google). Example: when we want to change the all students Gmail in Google to Microsoft oracle. There might be a problem with compatibility.
* Challenge to migration
* Increased training and maintenance costs
* Objectives and trade-offs in distributed database
* Location transparency:
* A user or program accessing data do not need to know the location of the data in the network of DBSMs
* Requests to retrieve or update data from any site are automatically forwarded by the system to the site or sites related to the processing request. -> basically, this cluster manager decides what is the nearest site where the data is available and looks after that request being processed.
* All data in the network appears to users as a single logical database stored at one site. A single query can join data from tables in multiple sites. (but in reality, a single query might form multiple sites, so it could be union or it could be joined, depend on how data distributed) (union is faster than join table)
* Local autonomy:
* It means that the database can be sitting on the servers and can operate without other servers. So, if there are breakdown communications, like London lost connect with Paris server, London still operate using local data. And also, be able to look after security, apply updates, use logs to restore whatever was lost during crashes and son on.
* Users can administer their local database
* control local data
* administer security
* log transactions
* recover when local failures occur
* provide full access to local data
* Being able to operate locally when connections to other databases fail
* Trade-offs:
* Availability vs Consistency
* The more data consistency enforced, the lower availability is
* Synchronous vs Asynchronous update:
* Synchronous waits for all involved database nodes to update, before confirming with query requestor (stock exchange)
* Asynchronous writes to the master database node and then confirm with the query requestor, without the other nodes necessarily having been updated (they may be updated a while later) (social media)
* Consistency is about the same record on any server would be exactly the same. But you have fully or partially replicated, it’s impossible to have every simultaneously.
* For consistency, synchronous is the best one, because as soon as I updated something. The update will send all over the world -> but this mean, those servers need to forget everything right now to do the update to ensure the consistency.
* To ensure everyone get the latest version of data, it slows down the servers.
* Distribution options:
* In partitioned, every server will have different data, because it’s a distributed database, the managing software know how to collect it from different servers.
* Replicated: everything is replicated
* Most often: there is a combination. Important things -> replication; others -> partitioning
* Data replication: data copied across sites
* Horizontal partitioning (by rows): table rows distributed across sites
* Vertical partitioning (by columns): table columns distributed across sites
* Combinations of the above
* Horizontal partitioning: row to row where it’s needed most (ex: popular in London, stored in London server)
* Different rows of a table at different sites
* Advantages:
* Data stores close to where it is used: efficiency
* Local access optimization: better performance
* Only relevant data is stored locally: security
* If the location is compromised, only part of things might be stolen (we might lose some data, but not everything)
* Unions across partitions: ease of query, collect data across partitions use union rows.
* Disadvantage:
* Accessing data across partitions: inconsistent access speed
* No data replication (difficult to restore it)
* Backup vulnerability (SPOF)
* Vertical partitioning:
* The partitioning is created base on the data needs, it can overlap. Like the same group of columns can be stored on more than one server. For aggregating function, vertical partitioning is more useful.
* Different columns of a table at different sites
* Also, possible to have all columns stored at some location
* The advantage and disadvantage are the same as for horizontal partitioning, except:
* Combining data across partitions is more difficult because it requires join (instead of unions, which use in horizontal partitioning)
* Advantage for aggregate column queries (vertical) faster than queries that select entire rows (horizontal)
* Replication advantage:
* High reliability due to redundant copies of data
* Fast access to data at the location where it is most accessed
* May avoid complicated distributed integrity routines
* replicated data is refreshed at scheduled intervals
* Decoupled nodes don’t affect data availability (because we have local autonomy)
* transactions proceed even if some nodes are down
* Reduced network traffic at prime time
* if updates can be delayed (only asynchronous)
* This is currently popular as a way of achieving high availability for global systems.
* Most SQL and NoSQL databases offer replication
* Replication disadvantage:
* Need more storage space: each server stores a copy of whole rows
* Data integrity: retrieve incorrect data if update have not arrived, the consistency that integrity can be violated
* Takes time for update operations
* High tolerance for out-of-date data may be required
* Updates may cause performance problems for busy nodes
* Network communication capabilities
* Updates can place heavy demand on telecommunications/networks
* Cost - high speed networks are expensive ($$$$$)
* 
* Synchronous updates: It ensure data integrity, but slow response
* Data is continuously kept up to date
* Users anywhere can access data and get the same answer.
* If any copy of a data item is updated anywhere on the network, the same update is immediately applied to all other copies or the update is aborted.
* Ensures data integrity and minimizes the complexity of knowing where the most recent copy of data is located.
* Can result in slow response time and high network usage
* The DDBMS spends time checking that an update is accurately and completely propagated across the network.
* The committed updated record must be identical in all servers
* Asynchronous updates: accept delay in propagating data
* Some delay in propagating data updates to remote databases
* Some degree of at least temporary inconsistency is tolerated
* May be acceptable if it is temporary and well managed
* Acceptable response time
* Updates happen locally and data replicas are synchronized in batches and predetermined intervals
* May be more complex to plan and design
* Need to ensure the right level of data integrity and consistency
* Suits some information systems more than others
* Compare commerce/finance systems with social media
* CAP theorem: three choose two (consistency, availability, and partitioning)
* The CAP theorem is a concept in distributed systems that states that it is impossible for a distributed database system to simultaneously provide more than two out of the following three guarantees ->
* During a network partition, a distributed system must choose between consistency and availability.
* Distributed database catalogue:
* (catalogue is a kind of group of files which contain metadata, Metadata is data about data, where data is and whether it’s old version or updated version) It is smaller than actual data table.
* Distributed Catalog Management refers to the process of managing metadata about data objects that are distributed across multiple nodes or servers in a distributed database system.
* This catalog contains information about the location and structure of the data objects, as well as the access rights and other attributes associated with each object.
* Since catalogs are themselves databases that contain metadata about the distributed database system, they need to have a storage framework.
* Three popular management schemes for distributed catalogs are:
* Centralized Catalogues -> catalogue is on one server and all server know where it is, whatever changes are made, the metadata is written into catalogue
* Fully Replicated Catalogues -> metadata replicated on every server. So once server make update, it’s urgently distributed to all other servers.
* Partially Replicated Catalogues -> metadata about every object there will have more than one copy, If one place fails other place can give the can give the catalogues.