

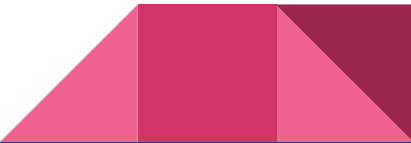
Databases

Basic Concepts (A.1)

What is a database?

- A structured set of **data** held in a computer, especially one that is accessible in various ways (dictionary.com)
- A database is an organized collection of structured **information**, or **data**, typically stored electronically in a computer system. (oracle.com)

Data vs. Information (A.1.1)

- What is data?
 - What is information?
 - What does a computer store, data or information?
- 

Computers store data. Data can be any one of several different types (e.g. numeric, text, Boolean etc.) but has no intrinsic meaning to a human. Data becomes information when it is put into a context that gives it meaning.

For example: **32 23 11 08 40 17** is data, but it has no meaning.

If we provide a *context* for that data, it becomes information, e.g.:

- The home and away scores for 6 soccer teams last Saturday
- The temperatures in degrees Celsius for 6 cities around the world at mid-day today
- The ages in years of the last 6 people to walk through the turnstiles of the Eiffel Tower in Paris.

Thus: **Information = Data + context**

Strictly speaking, databases store data, not information. However the terminology is commonly used loosely as there is an assumption that data stored in and retrieved from a database is interpreted by human beings within the scope of an appropriate context.

Data is meaningless. To be useful, data must be interpreted to produce information.

Information System vs. Database (A.1.2)

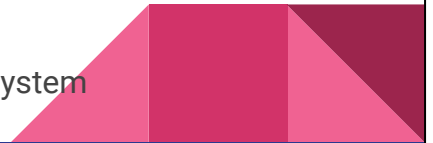
What is an Information System?

An information system is any organized system for the collection, organization, storage, and communication of information.

An information system is a group of components that interact to produce information.

Information systems are made up of six components: hardware, software, data, people, network, and process.

Databases are a component within an Information System



Students must be aware that these terms are not synonymous.

Database vs Spreadsheet

Databases and spreadsheets (such as Microsoft Excel) are both convenient ways to store information. The primary differences between the two are:

- How the data is stored and manipulated
- Who can access the data
- How much data can be stored

Spreadsheets were designed for one user.

Databases are designed for multiple users.



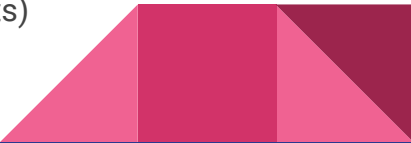
Spreadsheets were originally designed for one user, and their characteristics reflect that. They're great for a single user or small number of users who don't need to do a lot of incredibly complicated data manipulation. Databases, on the other hand, are designed to hold much larger collections of organized information—massive amounts, sometimes. Databases allow multiple users at the same time to quickly and securely access and query the data using highly complex logic and language.

Why do we need databases? (A.1.3)

Naïve implementations

- Spreadsheets...aren't they good enough?
- What about flat files? Can't I use those?
- CSV? JSON? XML? So many formats, what's wrong with them?

Why do we need more sophisticated implementations?

- Data size (millions, billions, trillions of data points)
 - Efficiency (dedicated algorithms developed by experts)
 - Abstraction
 - Faster implementation
- 

1. Size of Data: The small amount of data storing into spreadsheet is fine, however it might turn into a large amount of data then Spreadsheet solution will not work. Even if the size of data records goes into millions then storing data in multiple spreadsheet which will create a problem of speed. It will take you long time to find a record from the multiple spreadsheet files.

2. Ease of Updating Data: Multiple people cannot edit the same file on same time. Other people must wait until files are available to update which results in wastage of time.

3. Accuracy: When user doing data entry in files then it might be possible to incorrect data due to no validation present like you can enter wrong spelling, wrong dates, and wrong amount. So the Data accuracy is hard to maintain and accuracy is in question.

4. Security: You cannot secure the data in the text files and spreadsheet. Anyone can access the file and read any data present in the file. So storing data will not work with banking, healthcare application, payroll department where privacy is difficult to maintain.

5. Redundancy: The duplication of data can be possible using text files or spreadsheet. Chances of adding multiple copies of data cannot be limited here. This will lead to accuracy issues. Maintaining and updating multiple copies is not an easy task.

6. Incomplete Data: Some of the data is not considered not important, so such data not entered in the file as no validation in place which leads the data integrity is in question.

To prevent above problem associated with storing data in the text file or spreadsheet the database is required.

Types of Databases

- Relational databases
- Object-oriented databases
- Distributed Databases
- Data warehouses
- NoSQL databases
- Graph databases
- OLTP databases



There are many different types of databases. The best database for a specific organization depends on how the organization intends to use the data.

- **Relational databases.** Relational databases became dominant in the 1980s. Items in a relational database are organized as a set of tables with columns and rows. Relational database technology provides the most efficient and flexible way to access structured information.
- **Object-oriented databases.** Information in an object-oriented database is represented in the form of objects, as in object-oriented programming. Relational databases are traditionally composed of tables with fixed-size fields and records. Object databases comprise variable-sized blobs, possibly serializable or incorporating a mime-type. The fundamental similarities between Relational and Object databases are the start and the commit or rollback. After starting a transaction, database records or objects are locked, either read-only or read-write. Reads and writes can then occur. Once the transaction is fully defined, changes are committed or rolled back atomically, such that at the end of the transaction there is no

- inconsistency.
- **Distributed databases.** A distributed database consists of two or more files located in different sites. The database may be stored on multiple computers, located in the same physical location, or scattered over different networks.
- **Data warehouses.** A central repository for data, a data warehouse is a type of database specifically designed for fast query and analysis.
- **NoSQL databases.** A NoSQL, or nonrelational database, allows unstructured and semistructured data to be stored and manipulated (in contrast to a relational database, which defines how all data inserted into the database must be composed). NoSQL databases grew popular as web applications became more common and more complex.
- **Graph databases.** A graph database stores data in terms of entities and the relationships between entities.
- **OLTP databases.** An OLTP database is a speedy, analytic database designed for large numbers of transactions performed by multiple users.


These are only a few of the several dozen types of databases in use today. Other, less common databases are tailored to very specific scientific, financial, or other functions. In addition to the different database types, changes in technology development approaches and dramatic advances such as the cloud and automation are propelling databases in entirely new directions. Some of the latest databases include

- **Open source databases.** An open source database system is one whose source code is open source; such databases could be SQL or NoSQL databases.
- **Cloud databases.** A cloud database is a collection of data, either structured or unstructured, that resides on a private, public, or hybrid cloud computing platform. There are two types of cloud database models: traditional and database as a service (DBaaS). With DBaaS,

- administrative tasks and maintenance are performed by a service provider.
- **Multimodel database.** Multimodel databases combine different types of database models into a single, integrated back end. This means they can accommodate various data types.
- **Document/JSON database.** Designed for storing, retrieving, and managing document-oriented information, document databases are a modern way to store data in JSON format rather than rows and columns.
- **Self-driving databases.** The newest and most groundbreaking type of database, self-driving databases (also known as autonomous databases) are cloud-based and use machine learning to automate database tuning, security, backups, updates, and other routine management tasks traditionally performed by database administrators.

Database Management System (DBMS)

A DBMS is a software package that abstracts a database. It allows a user to

- Safely access, lock, and modify data
 - Provides
 - Centralized view of data that can be accessed by multiple users, from multiple locations, in a controlled manner
 - Security
 - Data integrity
 - Concurrency
 - Uniform Administrative procedures
 - Performance monitoring
 - Backup
 - Recovery
- 

This should address topics such as the benefits of data sharing.

A database typically requires a comprehensive database software program known as a database management system (DBMS). A DBMS serves as an interface between the database and its end users or programs, allowing users to retrieve, update, and manage how the information is organized and optimized. A DBMS also facilitates oversight and control of databases, enabling a variety of administrative operations such as performance monitoring, tuning, and backup and recovery.

Some examples of popular database software or DBMSs include MySQL, Microsoft Access, Microsoft SQL Server, FileMaker Pro, Oracle Database, and dBASE.

Multiple applications can access the same database at the same time.

Multiple users can access the same database at the same time.

DBMS makes sure modifications are complete and verified before sharing those changes with users/applications.

Advantages of a DBMS

Using a DBMS to store and manage data comes with advantages, but also overhead. One of the biggest advantages of using a DBMS is that it lets end users and application programmers access and use the same data while managing data integrity. Data is better protected and maintained when it can be shared using a DBMS instead of creating new iterations of the same data stored in new files for every new application. The DBMS provides a central store of data that can be accessed by

multiple users in a controlled manner.

Central storage and management of data within the DBMS provides:

- Data abstraction and independence
- Data security
- A locking mechanism for concurrent access
- An efficient handler to balance the needs of multiple applications using the same data
- The ability to swiftly recover from crashes and errors, including restartability and recoverability
- Robust data integrity capabilities
- Logging and auditing of activity
- Simple access using a standard application programming interface (API)
- Uniform administration procedures for data

Another advantage of a DBMS is that it can be used to impose a logical, structured organization on the data. A DBMS delivers economy of scale for processing large amounts of data because it is optimized for such operations.

A DBMS can also provide many views of a single database schema. A view defines what data the user sees and how that user sees the data. The DBMS provides a level of abstraction between the conceptual schema that defines the logical structure of the database and the physical schema that describes the files, indexes and other physical mechanisms used by the database. When a DBMS is used, systems can be modified much more easily when business requirements change. New categories of data can be added to the database without disrupting the existing system and applications can be insulated from how data is structured and stored.

Of course, a DBMS must perform additional work to provide these advantages, thereby bringing with it the overhead. A DBMS will use more memory and CPU than a simple file storage system. And, of course, different types of DBMSes will require different types and levels of system resources.

Transactions (A.1.4, A.1.5)

A transaction symbolizes a unit of work performed within a DBMS against a database, and treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in a database.

Transactions in a database environment have two main purposes:

1. To provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure, when execution stops (completely or partially) and with many operations upon a database remain uncompleted, with unclear status.
2. To provide isolation between programs accessing a database concurrently. If this isolation is not provided, the program's outcomes are possibly erroneous.

Database transactions can be used to introduce some level of fault tolerance and data integrity after recovery from a crash. A database transaction is a unit of work, typically encapsulating a number of operations over a database (e.g., reading a database object, writing, acquiring lock, etc.), an abstraction supported in database and also other systems. Each transaction has well defined boundaries in terms of which program/code executions are included in that transaction (determined by the transaction's programmer via special transaction commands).

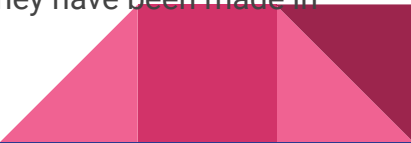
The acronym ACID describes some ideal properties of a database transaction: atomicity, consistency, isolation, and durability.

For example, to ensure data consistency when moving money between two accounts it is necessary to complete two operations (debiting one account and crediting the other). Unless both operations are carried out successfully, the transaction will be rolled back.

Concurrency (A.1.6)

Concurrency control is a database management systems (DBMS) concept that is used to address conflicts with the simultaneous accessing or altering of data that can occur with a multi-user system. If two or more users try to update the contents of a database simultaneously, locks and partitions are put into place to prevent it. Thus enabling greater concurrency.

Concurrent transactions must be locked before any modifications can be made and released only after all modifications have been completed and verified. Often, DBMS will provide a method to 'roll back' transactions if they have been made in error or were unable to complete.



Basic transactions (A.1.8)

CRUD - Create, Read, Update, Delete

CREATE - Inserting new data into a database using whatever schema has been defined.

READ - Accessing data in the database.

UPDATE - Modifying data that already exists in the database.

DELETE - Removing data from a database.

All transactions should follow all ACID properties.



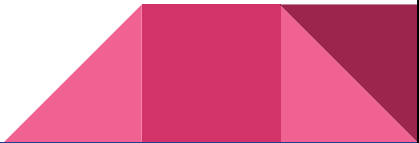
Read is the least risky transaction, as in it does not modify data

In a good database, all transactions must be able to be rolled back in case of failure or error. This often means writing changes to temporary files or writing previous data to temporary memory or temporary files before completing transactions.

ACID - Atomicity, Consistency, Isolation, Durability (A.1.7)

Atomicity

Transactions are often composed of multiple statements. Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors and crashes.



ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.

A series of database operations in an atomic transaction will either all occur, or none will occur. The series of operations cannot be separated with only some of them being executed, which makes the series of operations "indivisible". A guarantee of atomicity prevents updates to the database occurring only partially, which can cause greater problems than rejecting the whole series outright. In other words, atomicity means indivisibility and irreducibility.[4] Alternatively, we may say that a Logical transaction may be made of, or composed of, one or more (several), Physical transactions. Unless and until all component Physical transactions are executed, the Logical transaction will not have occurred – to the effects of the database. Say our Logical transaction consists of transferring funds from account A to account B. This Logical transaction may be composed of several Physical transactions consisting of first removing the amount from account A as a first Physical transaction and then, as a second transaction, depositing said amount in account B. We would not want to see the amount removed from account A before we are sure it has been transferred into account B. Then, unless and until both transactions have happened and the amount has been transferred to account B, the transfer has not, to the effects of the database, occurred.

Consistency

Consistency ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This prevents database corruption by an illegal transaction, but does not guarantee that a transaction is correct. Referential integrity guarantees the primary key - foreign key relationship.

ensures that the database properly changes states upon a successfully committed transaction.

Consistency is a very general term, which demands that the data must meet all validation rules. In the previous example, the validation is a requirement that $A + B = 100$. All validation rules must be checked to ensure consistency. Assume that a transaction attempts to subtract 10 from A without altering B. Because consistency is checked after each transaction, it is known that $A + B = 100$ before the transaction begins. If the transaction removes 10 from A successfully, atomicity will be achieved. However, a validation check will show that $A + B = 90$, which is inconsistent with the rules of the database. The entire transaction must be cancelled and the affected rows rolled back to their pre-transaction state. If there had been other constraints, triggers, or cascades, every single change operation would have been checked in the same way as above before the transaction was committed. Similar issues may arise with other constraints. We may have required the data types of both A and B to be integers. If we were then to enter, say, the value 13.5 for A, the transaction will be cancelled, or the system may give rise to an alert in the form of a trigger (if/when the trigger has been written to this effect). Another example would be with integrity constraints, which would not allow us to delete a row in one table whose Primary key is referred to by at least one foreign key in other tables.

Isolation

Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time). Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially. Isolation is the main goal of concurrency control; depending on the method used, the effects of an incomplete transaction might not even be visible to other transactions.



enables transactions to operate independently of and transparent to each other.

Durability

Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory.



ensures that the result or effect of a committed transaction persists in case of a system failure.

Consider a transaction that transfers 10 from A to B. First it removes 10 from A, then it adds 10 to B. At this point, the user is told the transaction was a success, however the changes are still queued in the disk buffer waiting to be committed to disk. Power fails and the changes are lost. The user assumes (understandably) that the changes persist.

Implementing ACID

Processing a transaction often requires a sequence of operations that is subject to failure for a number of reasons. Several methods have been designed to ensure ACID capabilities. The two most popular are

- **Locking**
 - Transactions mark data that it is accessing, preventing other transactions from accessing that data
 - Often uses two-phase locking:
 - Expanding phase
 - Shrinking phase
- **Multiversioning**
 - The database provides each READ transaction the prior, unmodified version of the data that is currently being modified.

Processing a transaction often requires a sequence of operations that is subject to failure for a number of reasons. For instance, the system may have no room left on its disk drives, or it may have used up its allocated CPU time. There are two popular families of techniques: write-ahead logging and shadow paging. In both cases, locks must be acquired on all information to be updated, and depending on the level of isolation, possibly on all data that may be read as well. In write ahead logging, atomicity is guaranteed by copying the original (unchanged) data to a log before changing the database.[dubious – discuss] That allows the database to return to a consistent state in the event of a crash. In shadowing, updates are applied to a partial copy of the database, and the new copy is activated when the transaction commits.

Locking vs multiversioning

Many databases rely upon locking to provide ACID capabilities. Locking means that the transaction marks the data that it accesses so that the DBMS knows not to allow other transactions to modify it until the first transaction succeeds or fails. The lock must always be acquired before processing data, including data that is read but not modified. Non-trivial transactions typically require a large number of locks, resulting in substantial overhead as well as blocking other transactions. For example, if user A is running a transaction that has to read a row of data that user B wants to modify, user B must wait until user A's transaction completes. Two phase locking is often applied to guarantee full isolation.

An alternative to locking is multiversion concurrency control, in which the database provides each reading transaction the prior, unmodified version of data that is being

modified by another active transaction. This allows readers to operate without acquiring locks, i.e., writing transactions do not block reading transactions, and readers do not block writers. Going back to the example, when user A's transaction requests data that user B is modifying, the database provides A with the version of that data that existed when user B started his transaction. User A gets a consistent view of the database even if other users are changing data. One implementation, namely snapshot isolation, relaxes the isolation property.

Distributed transactions

Guaranteeing ACID properties in a distributed transaction across a distributed database, where no single node is responsible for all data affecting a transaction, presents additional complications. Network connections might fail, or one node might successfully complete its part of the transaction and then be required to roll back its changes because of a failure on another node. The two-phase commit protocol (not to be confused with two-phase locking) provides atomicity for distributed transactions to ensure that each participant in the transaction agrees on whether the transaction should be committed or not.[5] Briefly, in the first phase, one node (the coordinator) interrogates the other nodes (the participants) and only when all reply that they are prepared does the coordinator, in the second phase, formalize the transaction.

Validation and Verification (A.1.9)

Data verification is a way of ensuring the user types in what he or she intends, in other words, to make sure the user does not make a mistake when inputting data.

Validation is about checking the input data to ensure it conforms with the data requirements of the system to avoid data errors. Data validation is the process of ensuring that data is valid. Data validation rules are used in data validation processes to ensure the validity. The term validity of data mostly denotes the meaningfulness and correctness of the data.

Data Validation is done on the original document whereas data verification is done on copies. This is the major difference between data validation and data verification.



It is imperative that data is entered correctly in a database. An automated database has the benefit of including automatic checks in order to prevent incorrect data from being recorded into the database. This is a process which isn't available in a manual database.

Validation is a process whereby the data entered in the database is checked to make sure that it is sensible. For example, validation can be utilized to check that only Male or Female is entered in a sex field. It cannot check whether or not the data entered is correct. It can only check whether or not the data makes sense.

Validation is a way of trying to lessen the number of errors during the process of data input.

Validation is carried out by the computer when you input data. It is a way of checking the input data against the set of validation rules.

The purpose of validation is to make sure that data is a) logical, b) rational, and c) complete and within acceptable limits.

Database Validation Methods

Type – If you make a specific field numeric then it won't allow you to input any letters or other non-numeric characters. Be wary when using the numeric data type. If you use it for fields like phone numbers, it won't allow you to enter spaces, or other human-friendly forms of formatting.

Some data types can carry out an extra type check. For example, a date data type will ensure that a date inputted existed at some point, or could exist in the future. It would not accept the date 30/02/2018.

Presence – This is sometimes called Allow Blank or Mandatory. This type of validation compels the user to enter data in the required field.

For example, in an address book, you can make either the address or phone number optional, while you must make the name field required. Leaving a mandatory field blank will trigger an error message that will prevent you from proceeding to the next step.

Unique Identifier – It is essential that one record can be plainly recognised from another record. Generally speaking, each record has one field that functions as a unique identifier for a record. An easy validation check can be done to make sure that a value occurs only once in this field—it doesn't matter if there are thousands of records in the database, the check can be carried out just the same.

Range Check – Range check is a validation check which can be applied to numeric fields. This is done to ensure that only numbers within a certain domain can be entered into a field. Remember that this does not necessarily mean that the data entered will be correct. But it will certainly lie within reasonable limits.

Format – This is used for a field that requires an entry in a specific format. Examples include date format, postal codes, and driver's license numbers.

Restricted Choice – There are times that fields in a database have a definite amount of data that can be entered into them. For example, the amount of days in a week are limited to Sunday, Monday, Tuesday, etc.

Programming a database to accept only one of a series of valid choices can prevent errors, and can also serve to lessen the time it takes to input data.

This has different forms like a list box, combo box, or radio button.

Benefits of Restricted Choice:

- Faster data entry, because it is typically much quicker to select from a list than to type each individual entry.
- Enhanced accuracy, because it lessens the risk of spelling mistakes.
- Limits the options to choose from by only displaying the essential choices.

Referential Integrity – If you're using a relational database, then you can impose referential integrity to validate inputs. You can check data inputs in certain fields against values in other tables. For example, in the job database, when a new hire is

entered, you could check the supervisor name against the employee table, just like you could check the department name against the department table.