Documentation Assignment on Terminologies Database

1.Entity:

An entity is an object that exists. It doesn't have to do anything; it just has to exist. In database administration, an entity can be a single thing, person, place, or object. Data can be stored about such entities. A design tool that allows database administrators to view the relationships between several entities is called the entity relationship diagram (ERD).

In database administration, only those things about which data will be captured or stored is considered an entity. If you aren't going to capture data about something, there's no point in creating an entity in a database.

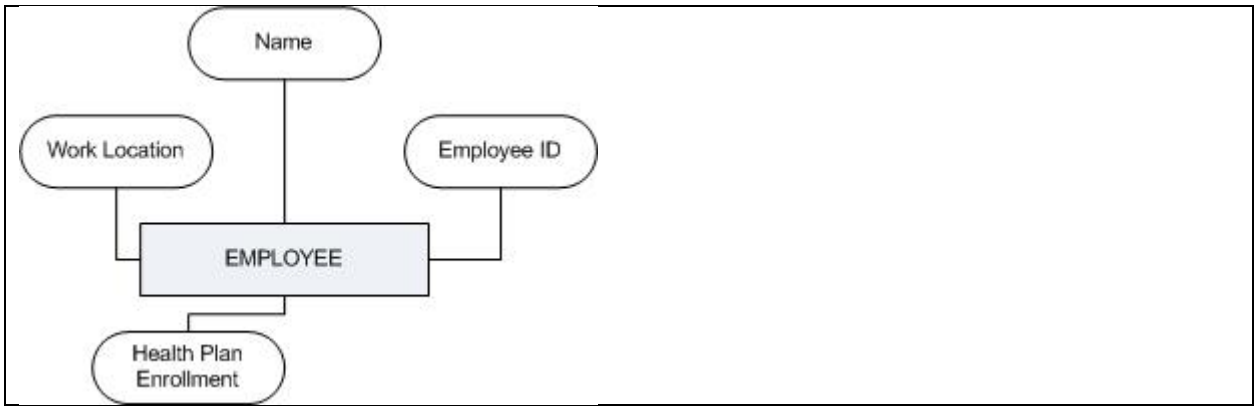
➢ Entity Attributes

An attribute defines the information about the entity that needs to be stored. If the entity is an employee, attributes could include name, employee ID, health plan enrollment, and work location. An entity will have zero or more attributes, and each of those attributes apply only to that entity. For example, the employee ID of 123456 belongs to that employee entity alone.

Attributes also have further refinements, such as domain and key. The **domain** of an entity describes the possible values of attributes. In the entity, each attribute will have only one value, which could be blank or it could be a number, text, a date, or a time. Here are examples of entity types and domains:

- Name: Jane Doe
- Employee ID: 123456
- Health Plan Enrollment: Premium Plan
- Work Location: RO, ME, Floor 2

Here's an example figure of an entity.



The **key** is the unique identifier that identifies the entity. A key is also a domain because it will have values. These values are unique to each record, and so it's a special type of domain. A key isn't always required, but it should be! In our example, a unique key value ensures that the Employee entity cannot have duplicate Social Security Numbers or Employee IDs.

2.Tuple:

In Database Management System (DBMS), most of the time we need to store the data in tabular format . This kind of data storage model is also called a Relational model and the system which leverages the relational model is called Relational Database Management System (RDBMS). These relations (or tables) consist of rows and columns. But in DBMS, we call these rows "Tuples" and a row "Tuple".

Example Of Tuple

Consider the table given below. We have data of some students like their id, name, age, etc. here, each row has almost all the information of the respective student. Like the first row has all the information about a student named **"Sufiyan"**, similarly, all other rows contain information about other students. Hence, a single row is also termed a **"record"** as it contains all the information of a student. This row or record is termed as Tuple in DBMS. Hence Tuple in DBMS is just a row representing some **inter-related data** of a particular entity such as student, employee, user, etc.

Table for reference:

ID	Name	Age	Subject	Marks
1	Sufiyan	21	Maths	80
2	Akash	23	Physics	90
3	Robin	29	Chemistry	75
4	Alina	24	Biology	95

A Tuple from the above-given table

In the above-given image, you can see that a Tuple is just a row having attributes of a particular entity like name, age, marks, etc.

Insuring unique rows

Since each tuple in a relation must be unique, no two tuples can have exactly the same values for every one of their attributes, that is, there can be no duplicate tuples in a relation. Unfortunately, the same cannot be said about SQL tables. A SQL table is bag (i.e., a multiset) of rows, unless constraints are placed on the table to ensure there be no duplicate rows. Thus, to implement a relation as a SQL table, there must be some set of attributes in each relation whose *values*, taken

together, guarantee uniqueness of each row. Any set of attributes that can do this is called a **super key** (SK). By the definition of a relation as a set of tuples, the set of all attributes must be a super key. If such a set were not a super key, it would allow two or more identical tuples in the relation which would violate the definition of a set. Since super keys are constraints on the data, they must be true for any relation (table) of a relation scheme, thus these super keys are shown in the relation scheme diagram. A super key is our first database constraint, we will learn more of them throughout.

The set of all attributes in a relation scheme R is just one super key of that scheme, there can be and usually there are more. The other super keys are proper subsets of the relation scheme. Out of all these super keys, the database designer picks one to serve as the **primary key** (PK) of the relation. (Notice that the PK is a SK, but not all SKs are PKs, since only one is chosen as a PK!) The PK is sometimes also called a *unique identifier* for each row of the table. This is not an arbitrary choice—we'll discuss it in detail on a later page. For our customers table, we'll pick the set {first_name, last_name, phone}. We are likely to have at least two customers with the same first and last name, but it is very unlikely that they will both have the same phone number.

In SQL, we specify the primary key of a table with a **data constraint** that lists the attributes that form the PK. We also give the constraint a name that is easy for us to remember later (as is done below using "customers_pk").

Normalization

Designing your database model is dependent on how your database will be used. OLTP systems are designed around a relatively standardized process called *normalization*. After you have completed the tasks of entity discovery or identifying the logical data entities in your system, the normalization rules provide guidelines for fine-tuning your data model to optimize performance, maintenance, and querying capabilities. Having said that, complete normalization is not always the best solution for your database. OLAP systems and some OLTP application requirements often result in a denormalized database or at least a denormalized segment. In OLAP solutions that typically contain mass amounts of historical data, the denormalized structure, including multiple copies of data and derived columns, can significantly increase analysis performance and justify its violation of normalization rules. The choice of complete normalization is always dependent on how your database will be used:

Normalization is a process of organizing the tables in a database into efficient, logical structures in order to eliminate redundant data and increase integrity. The physical results of normalizing a database are a greater number of smaller tables that are related to each other. Although there are up to seven normalization rules, called *forms*, the first three forms of normalization are the most significant and commonly used. The remaining normal forms are primarily academic. The primary normal forms are:

■

First normal form (1NF) Eliminate repeating groups and nonatomic attributes (or fields that contain multiple values).

■

Second normal form (2NF) Eliminate partial dependencies.

■

Third normal form (3NF) Eliminate nonkey dependencies and derived columns.

In order for the tables in your database to comply with the 1NF:

■

They must have no repeating groups.

■

Each field must be atomic (contains no multivalue data).

So what does that mean? The easiest way to understand this concept is to take a look at an example of a table that needs to be normalized. Table 4.5 is an un normalized table representing cities.

Table 4.5. An Unnormalized City Table

State (Key)	Governor	City (Key)	Founded	Years Old	Founders	Suburb1	Suburb2
NY	Pataki	Roberton	12/28/1941	59	Erwin, Patton	Michaelville	Alexopolis
NY	Pataki	Willville	8/24/1932	68	DeWolf	Auburn	

After you inspect the table for a moment, you will see that Suburb1 and Suburb2 are a repeating group. One of the reasons that repeating groups are troublesome is that they restrict how far you can extend your database to include all related information. After all, what would you do with a third suburb in this case? Another problem is that unnormalized database designs waste space. In this example, cities that have no or one suburb will not fully utilize the allocated space for the row. Finally, these designs make searching and sorting cities by their suburbs difficult. To eliminate the repeating group, you need to create

a new entity called Suburb and form a relationship between the two entities.

There is another problem with the City table: Founders is not atomic, because it contains more than one value that can be split. Again, this design will prevent you being able to sort or search on the data effectively. We could try to resolve this problem by splitting Founders into Founder1 and Founder2, However, this would create a repeating group like the one we had with Suburb1 and Suburb2. Once again, the solution is to add a new entity that represents the founders related to a city. After we add the two entities, we will comply with the 1NF and our tables should look like Tables 4.6–4.8.

Data normalization is the process of reorganizing data within a database so that users can utilize it for further queries and analysis. Simply put, it is the process of developing clean data. This includes eliminating redundant and unstructured data and making the data appear similar across all records and fields.

Keys in SQL

Before moving on to the different forms of data normalization, you need to first understand the concept of keys in SQL. A key can be a single column or a combination of columns that uniquely identify the rows (or tuples) in the table. It also helps to identify duplicate information and establish relationships between different tables.

Here are the most common type of keys:

- Primary key - A single column used to uniquely identify a table
- Composite key - A set of columns used to uniquely identify the rows in a table
- Foreign key - A key that references the primary key of another table

2NF - Second Normal Form

In a 2NF table, all the subsets of data that can be placed in multiple rows are placed in separate tables. For a table to be in the second normal form, it should satisfy the following rules:

- It should be in 1F
- The primary key should not be functionally dependant on any subset of candidate key

Let's divide the 1NF table into two tables - Table 1 and Table 2. Table 1 contains all the employee information. Table 2 contains information on their key skills.

Table 1

Employee ID	Salutation	Full Name	Address
1	Mr.	John Denver	12, Bates Brothers Road
2	Ms.	Mary Ann	34, Shadowman Drive
3	Ms.	Nancy Drew	4, First Plot Street

Table 2

Employee ID	Key skills
1	Content marketing
1	Social media marketing
2	Machine learning
2	Data science
3	DBMS

We have introduced a new column called Employee ID which is the primary key for Table 1. The records can be uniquely identified using this primary key.

3NF - Third Normal Form

For a table to be in the third normal form, it should satisfy the following rules:

- It should be in 2F
- It should not have any transitive functional dependencies

A transitive functional dependency is when a change in a column (which is not a primary key) may cause any of the other columns to change.

In our example, if there is a name change (male to female), there may be a change in the salutation (Mr., Ms., Mrs., etc.). Hence we will introduce a new table that stores the salutations

Table 1

Employee ID	Full Name	Address	Salutation
1	John Denver	12, Bates Brothers Road	1
2	Mary Ann	34, Shadowman Drive	2
3	Nancy Drew	4, First Plot Street	2

Table 2

Employee ID	Key skills
1	Content marketing
1	Social media marketing
2	Machine learning
2	Data science
3	DBMS

Table 3

Salutation ID	Salutation
1	Mr.
2	Ms.
3	Mrs.

Now, there are no transitive functional dependencies and our table is now in 3F. Salutation ID is the primary key in Table 3. Salutation ID in Table 1 is foreign to the primary key in Table 3.

BCNF - Boyce and Codd Normal Form

Boyce and Codd Normal Form is a higher version of 3NF and is also known as 3.5NF. A BCNF is a 3NF table that does not have multiple overlapping candidate keys. For a table to be in BCNF, it should satisfy the following rules:

- It should be in 3F
- For each functional dependency ($X \rightarrow Y$), X should be a super key

Benefits of Data Normalization

As data becomes more and more valuable to any type of business, data normalization is more than just reorganizing the data in a database. Here are some of its major benefits:

- Reduces redundant data
- Provides data consistency within the database
- More flexible database design
- Higher database security
- Better and quicker execution
- Greater overall database organization

A company can collect all the data it wants from any source. However, without data normalization, most of it will simply go unused and not benefit the organization in any meaningful way.

What is an ER diagram?

An Entity Relationship (ER) Diagram is a type of flowchart that illustrates how “entities” such as people, objects or concepts relate to each other within a system. ER Diagrams are most often used to design or debug relational databases in the fields of software engineering, business information systems, education and research. Also known as ERDs or ER Models, they use a defined set of symbols such as rectangles, diamonds, ovals and connecting lines to depict the interconnectedness of entities, relationships and their attributes. They mirror grammatical structure, with entities as nouns and relationships as verbs.

ER diagrams are related to data structure diagrams (DSDs), which focus on the relationships of elements within entities instead of relationships between entities themselves. ER diagrams also are often used in conjunction with data flow diagrams (DFDs), which map out the flow of information for processes or systems.

History of ER models

Peter Chen (a.k.a. Peter Pin-Shan Chen), currently a faculty member at Carnegie-Mellon University in Pittsburgh, is credited with developing ER modeling for database design in the 1970s. While serving as an assistant professor at MIT’s Sloan School of Management, he published a seminal paper in 1976 titled “The Entity-Relationship Model: Toward a Unified View of Data.”

In a broader sense, the depiction of the interconnectedness of things dates back to least ancient Greece, with the works of Aristotle, Socrates and Plato.

It's seen more recently in the 19th and 20th Century works of philosopher-logicians like Charles Sanders Peirce and Gottlob Frege.

By the 1960s and 1970s, Charles Bachman (above) and A.P.G. Brown were working with close predecessors of Chen's approach. Bachman developed a type of Data Structure Diagram, named after him as the Bachman Diagram. Brown published works on real-world systems modeling. James Martin added ERD refinements. The work of Chen, Bachman, Brown, Martin and others also contributed to the development of Unified Modeling Language (UML), widely used in software design.

Uses of entity relationship diagrams

- Database design: ER diagrams are used to model and design relational databases, in terms of logic and business rules (in a logical data model) and in terms of the specific technology to be implemented (in a physical data model.) In software engineering, an ER diagram is often an initial step in determining requirements for an information systems project. It's also later used to model a particular database or databases. A relational database has an equivalent relational table and can potentially be expressed that way as needed.
- Database troubleshooting: ER diagrams are used to analyze existing databases to find and resolve problems in logic or deployment. Drawing the diagram should reveal where it's going wrong.
- Business information systems: The diagrams are used to design or analyze relational databases used in business processes. Any business process that uses fielded data involving entities, actions and interplay can potentially benefit from a relational database. It can streamline processes, uncover information more easily and improve results.
- Business process re-engineering (BPR): ER diagrams help in analyzing databases used in business process re-engineering and in modeling a new database setup.

- Education: Databases are today's method of storing relational information for educational purposes and later retrieval, so ER Diagrams can be valuable in planning those data structures.
- Research: Since so much research focuses on structured data, ER diagrams can play a key role in setting up useful databases to analyze the data.

The components and features of an ER diagram

ER Diagrams are composed of entities, relationships and attributes. They also depict cardinality, which defines relationships in terms of numbers. Here's a glossary:

Entity
A definable thing—such as a person, object, concept or event—that can have data stored about it. Think of entities as nouns. Examples: a customer, student, car or product. Typically shown as a rectangle.

Entity type: A group of definable things, such as students or athletes, whereas the entity would be the specific student or athlete. Other examples: customers, cars or products.

Entity set: Same as an entity type, but defined at a particular point in time, such as students enrolled in a class on the first day. Other examples: Customers who purchased last month, cars currently registered in Florida. A related term is instance, in which the specific person or car would be an instance of the entity set.

Entity categories: Entities are categorized as strong, weak or associative. A strong entity can be defined solely by its own attributes, while a weak entity cannot. An associative entity associates entities (or elements) within an entity set.

Entity keys: Refers to an attribute that uniquely defines an entity in an entity set. Entity keys can be super, candidate or primary. Super key: A set of attributes (one or more) that together define an entity in an entity set. Candidate key: A minimal super key, meaning it has the least possible number of attributes to still be a super key. An entity set may have more than one candidate key. Primary key: A candidate key chosen by the database designer to uniquely identify the entity set. Foreign key: Identifies the relationship between entities.

Relationship

How entities act upon each other or are associated with each other. Think of relationships as verbs. For example, the named student might register for a course. The two entities would be the student and the course, and the relationship depicted is the act of enrolling, connecting the two entities in that way. Relationships are typically shown as diamonds or labels directly on the connecting lines.

Recursive relationship: The same entity participates more than once in the relationship.

Attribute

A property or characteristic of an entity. Often shown as an oval or circle.

Descriptive attribute: A property or characteristic of a relationship (versus of an entity.)

Attribute categories: Attributes are categorized as simple, composite, derived, as well as single-value or multi-value. Simple: Means the attribute value is atomic and can’t be further divided, such as a phone number. Composite: Sub-attributes spring from an attribute. Derived: Attributed is calculated or otherwise derived from another attribute, such as age from a birthdate.

Multi-value: More than one attribute value is denoted, such as multiple phone numbers for a person.

Single-value: Just one attribute value. The types can be combined, such as: simple single-value attributes or composite multi-value attributes.

Cardinality

Defines the numerical attributes of the relationship between two entities or entity sets. The three main cardinal relationships are one-to-one, one-to-many, and many-many. A one-to-one example would be one student associated with one mailing address. A one-to-many example (or many-to-one, depending on the relationship direction): One student registers for multiple courses, but all those courses have a single line back to that one student. Many-to-many example: Students as a group are associated with multiple faculty members, and faculty members in turn are associated with multiple students.

Cardinality views: Cardinality can be shown as look-across or same-side, depending on where the symbols are shown.

Cardinality constraints: The minimum or maximum numbers that apply to a relationship.

High and Low Database Cardinality Definition

The more important definition of cardinality for query performance is *data* cardinality. This is all about how many distinct values are in a column. We usually don't talk about cardinality as a *number*, though. It's more common to simply talk about "high" and "low" cardinality. A lot of distinct values is high cardinality; a lot of repeated values is low cardinality.

Cardinality in Time Series Databases

In addition to cardinality in databases, I also want to help simplify what it means to use cardinality in monitoring. If you've seen discussions of "high-cardinality dimensions" or "observability requires support for high-cardinality fields," this is what we're talking about. So what does it mean? Generally, this refers to the number of *series* in a time series database. A time series is a labeled set of values over time, stored as (timestamp, number) pairs. So, for example, you might measure CPU utilization and store it in a time series database:

This data model is the canonical starting point for most monitoring products. But it doesn't contain a lot of richness: what if I have a lot of servers and want to know the average CPU utilization of, say, database servers versus web servers? How can I filter one kind versus the other? To solve this problem, many monitoring systems nowadays

support *tags* with extra information. One way to conceptualize this is to make those data points N-dimensional instead of simply timestamps and numbers: `os.cpu.util = [(5:31, 82%, role=web), (5:32, 75%, role=web), (5:33, 83%, role=web)...]` This looks wasteful, doesn't it? We've repeated "role=web" again and again, and we should be able to do it just once. Plus, most time series software typically tries to avoid N-dimensional storage because time-value pairs can be encoded efficiently—it's much harder to build a database capable of storing these arbitrary name=value tags. The typical *time series* monitoring software solves this by storing the tags with the series identifier, making it part of the identifier: `(name=os.cpu.util,role=web) = [(5:31, 82%), (5:32, 75%), (5:33, 83%)...]` But what if "role" changes over time? What if it's not constant, even within a single server? Most existing time series software says, well, it'll become a new series when a tag changes because the tag is part of the series identifier:

```
(name=os.cpu.util,role=web) = [(5:31, 82%), (5:32, 75%)]
(name=os.cpu.util,role=db)  = [(5:33, 83%)...]
```

When people talk about cardinality in monitoring and how it's hard to handle high-cardinality dimensions, they're basically talking about how many distinct combinations of tags there are, and thus the number of series. And there can be lots of tags, so there can be lots of combinations! `(name=os.cpu.util,role=web, datacenter=us-east1, ami=ami-5256b825, …) = [...]` Most of these tags are pretty static, but when one of the tags has *high cardinality*, it simply explodes the number of combinations of tags. A tag with medium cardinality, for example, could be a build identifier for a deployment artifact. High-cardinality tags would come from the *workload* itself: customer identifier. Session ID. Request ID. Remote IP address. That type of thing. Most time series databases instantly crumble under these workloads because their data model and storage engine are optimized for storing points efficiently and aren't optimized for lots of series. For example:

Some of the more modern time series databases are built for many series. InfluxDB is an example. They've put a lot of work into handling tons of series, and they have quite a bit in their documentation about how InfluxDB deals with high cardinality. But this is all about *storage* and whether the time series database can handle *storing* lots of series. What about *retrieval*? Can the time series database handle arbitrary queries against its data without regard to the nature and cardinality? Typical time series databases can't because they're built around—and designed to operate within—the constraints of series. Think of a series as a "lane" of data: typical time series databases can only swim within the lane when they run a query, and they can't swim perpendicular to the lanes. This is because a series is a pre-aggregation of the original source data. The series identifier is what all the values in the series have in common, and when the data was serialized (so to speak), it was aggregated around the series identifier at write time. The series ID has the same function as GROUP BY fields in a SQL database. But unlike using GROUP BY in SQL, typical time series databases do the GROUP BY *when they ingest the data*. And once grouped, the data can't ever be ungrouped again. Some databases—again, InfluxDB, for example—have *both* tags-as-series-identifier *and* multidimensional values. It's sort of like this: `(name=os.cpu.util,role=web) = [(5:31, 82%, build_id=ZpPZ5khe)...]` But some of these tags are more special than others, so to speak. InfluxDB talks about which tags are "indexed"—which ones are part of the series ID and pre-grouped-by. In general, software with special and non-special tags like this usually has some restriction around operations on it: maybe you can't filter by some tags, or maybe you can't group by some tags, or so on. Druid is in the same vein as InfluxDB in this regard. And this is where the focus of products and technologies suddenly becomes clear. Traditional time series software was designed with an internal-facing SysAdmin worldview in mind, where we inspected our own systems/servers and

cardinality was naturally low. This is where RRD files came from. We looked inward to figure out whether our systems were working. But now, in the age of observability, forward-thinking engineers (and vendors) are focused on measuring and understanding the *workload* or *events.* Workload (query/event) measurements are very high-cardinality data sets by nature. What's the job of a database? To run queries. How do you know if it's working? Measure whether the queries are successful, fast, and correct! Don't look at the CPU and disk utilization—it's the wrong place to look. So when someone mentions high cardinality in monitoring, why it's important, and why it's hard, what does it mean? I'll summarize:

- Workload or event data is the right way to measure customers' actions and experiences.
- System workload is many-dimensional data, not just one-dimensional values over time, and it has very high cardinality.
- Traditional time series databases were designed with a system-centric worldview and thus weren't built to store *or* query workload data.

Using traditional tools to measure, inspect, and troubleshoot customers' experiences is basically impossible because of pre-aggregation and cardinality limitations, and this leads engineers to focus on what the tool *can* offer them—which is often the wrong place to look.

Importance of Cardinality for Database Performance

Cardinality has a big impact on database performance because it influences the query execution plan. The planner will examine column statistics and use them to figure out how many values a query is likely to match, among other things. Depending on what it finds, it might use different query execution plans to improve database performance. But this is a topic for a different blog post because it takes a bit of work to

explain. So next time someone drops “high cardinality” into a sentence without pausing, you know they really mean “a lot of different values.”

What is a DBA?

Short for database administrator, a DBA designs, implements, administers, and monitors data management systems and ensures design, consistency, quality, and security.

According to SFIA 8, database administration involves the installing, configuring, monitoring, maintaining, and improving the performance of databases and data stores. While design of databases would be part of solution architecture, the implementation and maintenance of development and production database environments would be the work of the DBA.

What does a DBA do?

The day-to-day activities that a DBA performs as outlined in ITIL® Service Operation include:

- Creating and maintaining database standards and policies
- Supporting database design, creation, and testing activities
- Managing the database availability and performance, including incident and problem management
- Administering database objects to achieve optimum utilization
- Defining and implementing event triggers that will alert on potential database performance or integrity issues
- Performing database housekeeping, such as tuning, indexing, etc.
- Monitoring usage, transaction volumes, response times, concurrency levels, etc.
- Identifying reporting, and managing database security issues, audit trails, and forensics
- Designing database backup, archiving, and storage strategy