

Relational Databases And SQL

David Liou

July-12-2023

Abstract

The Relational Databases and SQL class will introduce attendees to the basics of data modeling, normalization, predicate logic, and structured query language (SQL). Additionally, this class will introduce NoSQL databases and their uses. Hands-on exercises will use example scientific data to illustrate the principles of relational structures and data querying. As part of the hands-on training, attendees will learn how to query databases using Python and R. No software installation is required prior to the class.

Contents

What Is a Database?	8
Data Modeling: Data As Entities	9
Introduction	9
Tables	9
Rows	9
Columns	9
Example	10
Database Diagrams	11
Primary Keys	11
Foreign Keys	12
Discussion	14
Conclusion	15
Normalization	16
Introduction	16
Reasons for Normalization	16
Data Duplication and Modification Anomalies	19
Definition of Normalization	19
1NF -- First Normal Form	20
2NF -- Second Normal Form	22
Issues With Our Example	22
Adjusted to 2NF Standards	22
3NF -- Third Normal Form	24
What It Means to Be Transitive	24
Dependence	24
Transitive Dependence	24
Transitive Dependence Applied to Our Example	26
Adjusted to 3NF Standards	27
Exercise	28
Conclusion	30
Ternary Logic	31
Introduction	31
Three-valued logic	31
Truth Tables	32
A Couple of Thought Experiments	33

Why Is False AND Unknown Equal False?	33
Why does True OR Unknown Equal True?	34
Discussion	34
Conclusion	35
Database Predicates	36
Introduction	36
Exercise: Clinical Report Form	37
Assessment of Severe Malaria (For children only)	37
Blantyre Coma Scale	38
Query 1:	40
SELECT participant_id	
FROM severe_malaria_crfs	
WHERE	
(prostrate OR fitting OR jaundiced OR dehydration)	
AND	
(eye_movement + verbal_response + motor_response) <= 3	40
C	40
Query 2:	40
C & F	40
Query 3:	40
Discussion	41
Conclusion	42
CAP Theorem	43
Introduction	43
Brewer's Conjecture	44
Consistency - All nodes see the same data at the same time.	44
Availability - Every request gets an immediate response.	44
Partitions - Requests arrive from remote clients without a direct connection	44
Illustration - A Tale of Two Kingdoms	45
Discussion	47
Relevance to Modern Database Systems	47
Conclusion	49
SQL Lab	50
SQL Language	51
Terminology	51
Common SQL Commands	52
NTP Data Collections: Statistically Analyzed NTP Pathology Lesions (with Incidence)	53
Querying from a Table	55

Retrieve all rows from a table and return the specified columns as the result set.	55
Retrieve all rows from a table and return all columns as the result set.	55
Retrieve a result set with a condition.	55
Retrieve rows from a table and return a unique result set.	56
Retrieve rows from a table and sort the result set.	56
Join Multiple Tables	57
Retrieve rows from two tables with an INNER JOIN	59
Retrieve rows from two tables with a LEFT OUTER JOIN	59
Retrieve rows from two tables with an RIGHT OUTER JOIN	59
Retrieve rows from two tables with a FULL OUTER JOIN	60
Query Aggregates	61
Retrieve rows from a table and return an aggregated result set.	61
Retrieve rows from a table and return a filtered aggregated result set.	61
SQL operators	63
Combine rows from two queries	63
Query rows using pattern matching %, _	63
Query rows in a list	63
Query rows between two values	63
Check if values in a table is NULL or not	63
Modifying data	64
Insert one row into a table	64
Insert rows from t2 into t1	64
Update new value in the column c1 for all rows	64
Update values in the column c1, c2 that match the condition	64
Delete all data in a table	64
Delete subset of rows in a table	64
Managing a schema	66
Create a new table with three columns	66
Delete a table from the schema	66
Create an index	66
Delete an index	66
Exercise: Normalizing and Adjusting the NTP Dataset	67
Connecting to MySQL with R	68
Connecting to the database	68
SQL with R	69
Connecting to MySQL with Python	70
Connecting to the database	70

SQL with Python	71
SQL Injection	72
Examples of SQL Injection Attacks	72
Preventing SQL Injection	72
Appendix	74
Importing the Pathology Lesions dataset to MySQL	74
MySQL Data Types	76
MySQL Workbench	78
Installing MySQL to Your Local Computer	79
Useful Online Resources	81
NIEHS Library	82

A SQL query goes into a bar, and walks up to two tables and asks...

„Can i join you?“

What Is a Database?

A database management system (DBMS) is a collection of interrelated data and a set of programs to access that data. The collection of data, usually referred to as the database, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information in an *efficient and secure* manner.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for the storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Most database systems seek to mitigate the problems of

- **Data consistency** - interrelated sets of data may become inconsistent when updates result in the separate components no longer agree
- **Difficulties with accessing data** - modern database systems offer querying languages that allow users to succinctly specify data to be retrieved
- **Data isolation** - information not stored in database systems tend to become scattered in different locations and formats
- **Concurrent access** - modern database systems allow concurrent access to stored information.
- **Security** - databases allow administrators to specify rules regarding who and what may be viewed and modified

Perhaps the most important feature of modern databases is the flexibility it offers for data storage. Administrators must be able to structure the data in ways that make sense to their business. This process is called **data modeling**.

Data Modeling: Data As Entities

Introduction

A data model is a logical abstraction of real-world entities. It organizes elements of data and standardizes how they relate to one another. The following is a brief introduction to relational data modeling.

Tables

A table is a collection of related data held in a structured format. Each table may be interpreted as a collection of homogeneous entities.

Rows

A row, also called a record or tuple, represents a single structured data item in a table. Each row may be viewed as a single entity.

Columns

The structure of a table is expressed as a list of column definitions. Each column definition specifies the name and the type of values permitted for that column. A database column holds analogous values for each row of the table. As such, the columns are representations of entity attributes.

Example

Let's consider a data table with study participant information:

ParticipantId	ParticipantSex	ParticipantAge
101	M	29
102	F	31
103	M	23

Each row is a representation of an entity. In our example, the entity is a study participant. The columns are attributes for each participant. In this case, the attributes are sex and age.

Also, notice that each column conforms to a specific data type; the ParticipantId is an integer, ParticipantSex is a character (M or F), and ParticipantAge is an integer. The columns also serve to constrain the allowable data type for each entity attribute.

Database Diagrams

Below is a representation of a table as an entity and its column attributes. This is part of an **entity-relationship diagram**:




Participants
ParticipantId: INTEGER
ParticipantSex: CHARACTER(1)
ParticipantAge: INTEGER

Primary Keys

A primary key is a column that uniquely identifies a row.

In our example dataset, the ParticipantId is designed to be a unique identifier for each participant. The ParticipantId guarantees that we can always specify a single participant entity unambiguously.

In entity-relationship diagrams the primary key is designated with a key icon:

Participants
 ParticipantId: INTEGER
 ParticipantSex: CHARACTER(1)
 ParticipantAge: INTEGER

Foreign Keys

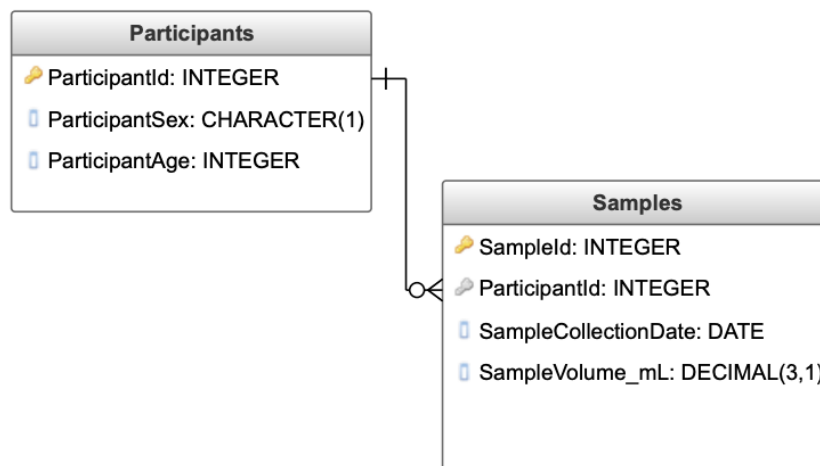
A foreign key is a column in one table that references the primary key of another table. This reference represents relationships between tables within a database.

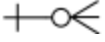
Let's expand our previous example to include samples drawn from the study participants.



ParticipantId	ParticipantSex	ParticipantAge	SampleId	SampleCollectionDate	SampleVolume_mL
101	M	29	1001	4-Apr-2017	10.2
101	M	29	1002	5-Apr-2017	11.5
102	F	31	1003	6-Apr-2017	11.3
102	F	31	1004	7-Apr-2017	10.5
103	M	23	1005	8-Apr-2017	10.8
103	M	23	1006	9-Apr-2017	11.0

In this example, there are two entities, *participants* and *samples*. A logical relationship exists between each participant and the samples taken from that participant.

Expanding our entity-relationship diagram to include the sample entity looks like:

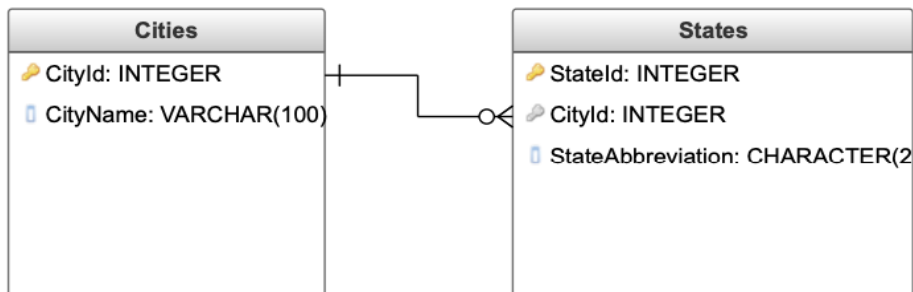
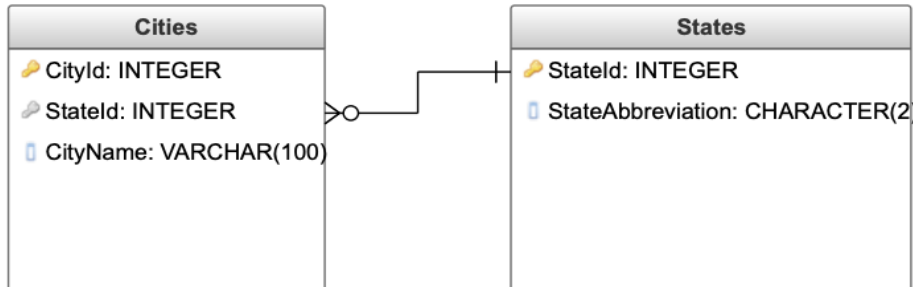


Entity-relationship diagrams represent table relationships with the diagram connector . This is also called a **crow's foot** due to its reminiscently avian claw-like semblance.

The  end identifies the parent, whereas the  is attached to the child. In our example, a study participant is connected to samples. The connector represents the relationship between a sample and the participant from which it was taken.

Discussion

- Which of the following diagrams is a correct representation of the relationship between US cities and states? How should each representation below be interpreted?



- Consider how each entity-relationship diagram would handle modifications. For example, in 1891 the North Carolina town of Hamburg changed its name to Glenville. What would happen in each relationship above?

City	State
Hamburg	MI
Hamburg	MN
Hamburg	NC
Hamburg	NJ
Hamburg	NY
Hamburg	PA

Conclusion

Tables may be viewed as a collection of homogeneous data entities. Each row is a specific entity instance with its attributes represented by the columns. The ability to establish relationships between tables adds depth to data not easily representable in a flat table. The process of translating flat data to tables with relationships is called normalization.

Normalization

Introduction

Database normalization is the process used to organize a database into tables and columns. The idea is that a table should be about a specific topic and that only those columns which support that topic are included. For example, a spreadsheet containing information about scientific projects serves several purposes:

- List studies conducted by research organizations
- Identify pertinent keywords and search terms for individual projects
- Name the lead researcher(s) for each effort

By limiting a table to one purpose you reduce the number of duplicate data that is contained within your database, which helps eliminate some issues stemming from database modifications. To assist in achieving these objectives, some rules for table organization have been developed. The stages of organizing are called normal forms; there are three normal forms most databases adhere to using. As tables satisfy each successive normalization form, they become less prone to database modification anomalies and more focused toward a sole purpose or topic.

Reasons for Normalization

There are three main reasons to normalize a database. The first is to minimize duplicate data, the second is to minimize or avoid data modification issues, and the third is to simplify queries. As we go through the various states of normalization we'll discuss how each form addresses these issues, but to start, let's look at some data which hasn't been normalized and discuss some potential pitfalls. Once these are understood, it's easier to appreciate the reason to normalize the data.

Consider the following table downloaded from the **NIH Research Portfolio** site:

 NIH Portfolio (Google Sheets)

ProjectId	ProjectTitle	Keywords	StartDate	Organization	City	State	District
9001	Developing reciprocal chromosomal translocations for wild population replacement ...	breeding; disorder prevention; fitness; insect genetics; mediating; population genetics; positioning attribute; structure; zika	12/19/16	Univ of California Riverside	Riverside	CA	41
9002	Zika virus evolutionary dynamics in host adaptation	biological; disease; disease outbreaks; event; laboratories; laboratory experiment; locales; methods; mosquito control; neurologic; perinatal; zika	2/7/17	Univ of Wisconsin-Madison	Madison	WI	2
9006	A Rapid and Specific Diagnostic for Immunoglobulin Response to Zika Virus Exposure ...	capsid proteins; color; development; diagnostic reagent; fetus; lateral; link; paper; pathogen; point of care; staging; zika	8/20/16	University of Washington	Seattle	WA	7
9007	Discovering host factors impacting ZIKV infection via forward genetic screens	andes virus; human genome; insertional mutagenesis; pathogen; response; venezuela; viral; west nile; zika	1/1/17	University of Pennsylvania	Philadelphia	PA	2
9008	Molecular and Antibody Detection of Zika Virus in Saliva at the Point of Care	design; development; heating; infection; laboratory facility; neurologic; pennsylvania; point-of-care diagnostics; tool; zika	9/9/16	University of Pennsylvania	Philadelphia	PA	2
9009	Zika virus in the human genital tract and implications for transmission	care seeking; case study; epidemic; nicaraguan; novel; pregnancy; vero cells; zika	2/15/17	Univ of North Carolina Chapel Hill	Chapel Hill	NC	4
9133	Mechanisms of sexual Zika virus transmission and early immunopathogenesis	cell type; dendritic cells; injection of therapeutic agent; transmitted diseases; viral transmission; virus replication; zika	12/1/16	University of Washington	Seattle	WA	7
9085	A New Filter Paper Technology for Flavivirus Collection, Shipping, and Analysis	chemicals; collection; dna; eligibility determination; formulation; mutate; new technology; small business innovation research grant; zika	3/8/17	GenTegra, LLC	Pleasanton	CA	15

Column Descriptions

- **ProjectId:** A unique number assigned to each project number.
- **ProjectTitle:** The grantee submitted title descriptive to the project.
- **Keywords:** Text used for site searches taken from project titles, abstracts, and scientific terms.
- **StartDate:** The date the project began.
- **Organization:** A generic term used to refer to an educational institution or other entity, including an individual, which applies for or receives an NIH grant, contract, or cooperative agreement.
- **City:** The business address city of the grantee organization.
- **State:** The business address state of the grantee organization.

- **District:** Congressional District: The congressional district to which each grant is assigned is based on the business address of the grantee organization.

Data Duplication and Modification Anomalies

Notice that for each research organization, the primary city and state for that entity is also listed. This information is duplicated for each organization.

Duplicated information presents two problems:

1. It increases storage
2. Duplication decreases performance
3. Enacting changes is more complicated

Additionally, let's suppose the census necessitates redistricting of some cities requiring congressional district numbering changes within the dataset. For large datasets, this would involve an extensive number of individual updates.

This situation is an example of a modification anomaly. There are three modification anomalies that can occur:

1. Insert Anomalies
2. Update Anomalies
3. Deletion Anomalies

These anomalies can be mitigated by apportioning the data into separating tables by function and domain. This process is called normalization. Normalization is assessed by a progressive series of criteria called normal forms.

Definition of Normalization

There are three common forms of normalization: 1st, 2nd, and 3rd normal form. The forms are progressive, meaning that to qualify for the 3rd normal form a table must first satisfy the rules for the 2nd normal form, and the 2nd normal form must adhere to those for the 1st normal form.

1. **First Normal Form** -- The information is stored in a relational table and each column contains atomic values and there are no repeating groups of columns.
2. **Second Normal Form** -- The table is in first normal form and all columns depend on the table's primary key.
3. **Third Normal Form** -- The table is in second normal form and all of its columns are not transitively dependent on the primary key.

1NF -- First Normal Form

The first steps to making a proper SQL table is to ensure the information is in first normal form. Once a table is in the first normal form it is easier to search, filter, and sort the information. The rules to satisfy first normal form, abbreviated 1NF, are:

- The table stores information in rows and columns where one or more columns, called the primary key, uniquely identify each row.
- Each column contains atomic values and there are not repeating groups of columns.

Atomic values are distinct terms that cannot be further subdivided. For example, values should not be concatenated into a single string separated by a delimiter such as a semi-colon (;). The text "zika" is atomic; whereas a concatenation of terms such as "zika; vector; transmission" is not. The keywords column of the NIH Research Portfolio example table contains non-atomic values.

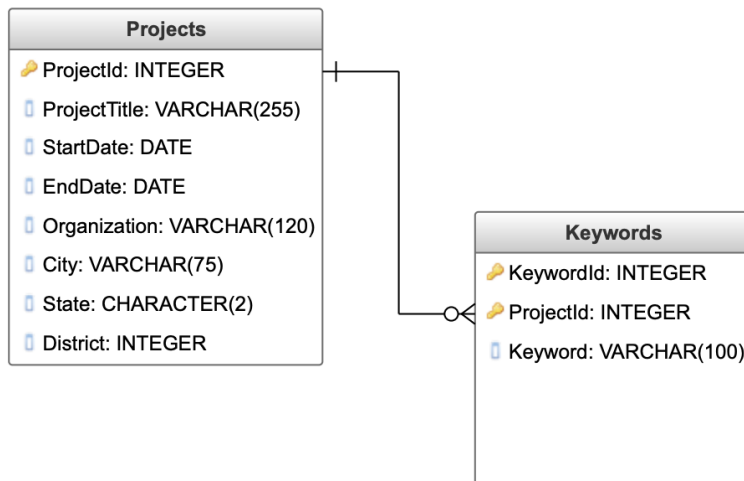
Related to this requirement is the concept that a table should not contain repeating groups of columns which are functionally equivalent to concatenating several values with a delimiter. Extending on the keywords example, simply splitting "zika; vector; transmission" into separate columns called *Keyword1*, *Keyword2*, and *Keyword3* still violates 1NF.

The Keywords column holds a list of values separated with a semicolon, e.g. "transmission;vector;zika". This is non-atomic and therefore violates 1NF.

Projects	
ProjectId:	INTEGER
ProjectTitle:	VARCHAR(255)
Keywords:	VARCHAR(500)
StartDate:	DATE
Organization:	VARCHAR(125)
City:	VARCHAR(75)
State:	CHARACTER(2)
District:	INTEGER

The Keywords column holds a list of values separated with a semicolon, e.g. "transmission;vector;zika". This is non-atomic and therefore violates 1NF.

Our example table is transformed to 1NF by placing the individual Keywords values into their own table.



The concatenated values are split and become separate rows in the Keywords table linked to Projects. The data is now 1NF.

This design is improved in several ways:

- The original table limited the maximum length of the concatenated keywords text to 500 characters. In the new design, the possible number of keywords associated with each project is unlimited.
- It is straightforward to sort and filter by keyword values. Instead of having to read and parse the concatenated keywords text, it's now possible to directly match values in the Keywords table.
- Potential modification anomalies for Projects have been reduced. Previously, modifying any single keyword would entail updating the entire string of keywords. Now updates to specific keywords do not disturb other non-targeted keywords.

2NF -- Second Normal Form

A table is in 2nd Normal Form or **2NF** if:

1. The table is 1NF
2. All the value (non-key) columns are directly dependent on all keys belonging to the table

The primary key serves to uniquely identify each row in a table. When all the columns directly relate to the primary key, they naturally share a common purpose. A table is in **2NF** if it solely serves a single entity. For the Keywords table, the purpose is to track user-submitted keywords.

An equivalent way of stating the second requirement is: a table infracts **2NF** if there is a value column not dependent on at least one of the table keys. An important consequence of this criteria is that **2NF** only applies to tables with multiple keys. **If a table is 1NF and only has a single key, it automatically satisfies the requirements for 2NF.**

Issues With Our Example

In our example, the only possible violation of **2NF** can occur with our Keywords table. The only other table, Projects, has a single primary key.

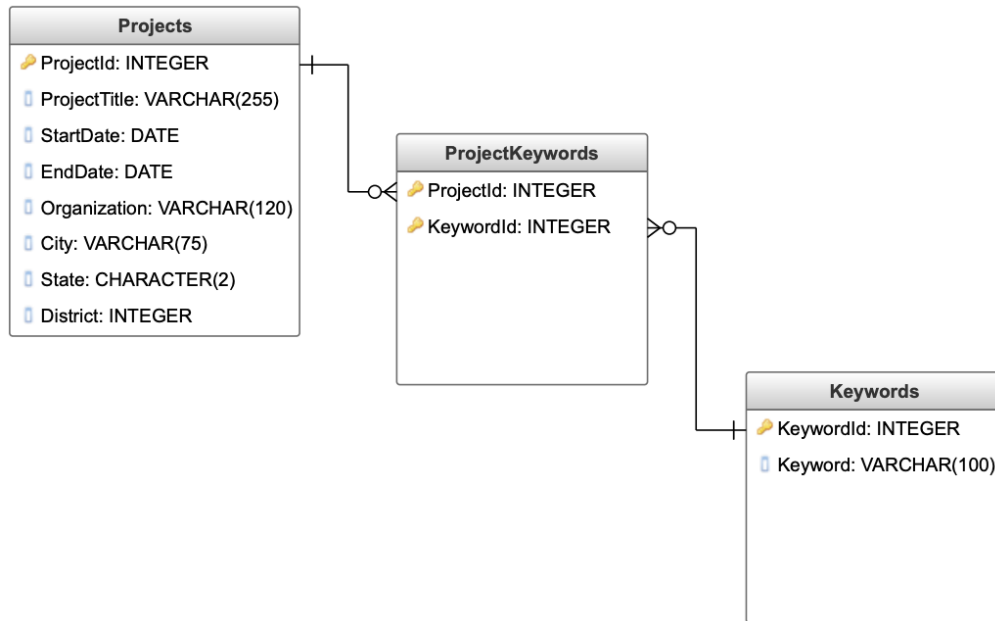
The Keywords table, the way it is, has two separate functions. The first role is to hold the actual value. The second is to keep track of the project to which it belongs. Additionally, the Keyword value doesn't conceptually seem to be dependent on the ProjectId. The Keywords table violates 2NF.

Adjusted to 2NF Standards

Our problems are

1. The Keywords table is maintaining occurrences in projects
2. The keyword value isn't dependent on the ProjectId key

Since the Keywords column isn't directly dependent on the ProjectId key, it is a reasonable strategy to fix this by moving the ProjectId field somewhere else. We also have to address the issue with the Keywords having an extraneous function. Creating a new table appeals to both of these needs.



3NF -- Third Normal Form

A table is in third normal form, **3NF**, if:

1. It satisfies **2NF**
2. It contains only columns that are non-transitively dependent on the primary key

What It Means to Be Transitive

When something is transitive, then a meaning or relationship is the same in the middle as it is on the whole. It helps to think of the prefix trans as meaning "across." When something is transitive, then if something applies from the beginning to the end, it also applies from the middle to the end.

Since ten is greater than five, and five is greater than three, you can infer that ten is greater than three.

$$10 > 5 > 3 \text{ therefore } 10 > 3$$

In this case, the greater than comparison is transitive. In general, if A is greater than B, and B is greater than C, then it follows that A is greater than C.

If you're having a hard time wrapping your head around "transitive" I think for our purpose it is safe to think "through" as we'll be reviewing to see how one column in a table may be related to others, through a second column.

Dependence

An object has a dependency on another object when it relies upon it. In the case of databases, when we say that a column has a dependence on another column, we mean that the value can be derived from the other. For example, my age is dependent on my birthday. Dependence also plays an important role in the definition of the second normal form.

Transitive Dependence

In mathematics and economics, the Axiom of Transitivity is formally described as:

If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

If X defines Y , and Y defines Z , then X (also) defines Z

Similarly, with databases, given three columns A, B, and C:

If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$

If A is defined by B , and B is defined by C , then A and C are transitively dependent.

Transitive Dependence Applied to Our Example

Let's examine the first entry of our example data model, specifically the ProjectId, Organization, City, State, and (Congressional) District:

ProjectId	...	Organization	City	State	District
9001	...	Univ of California	Riverside	CA	41

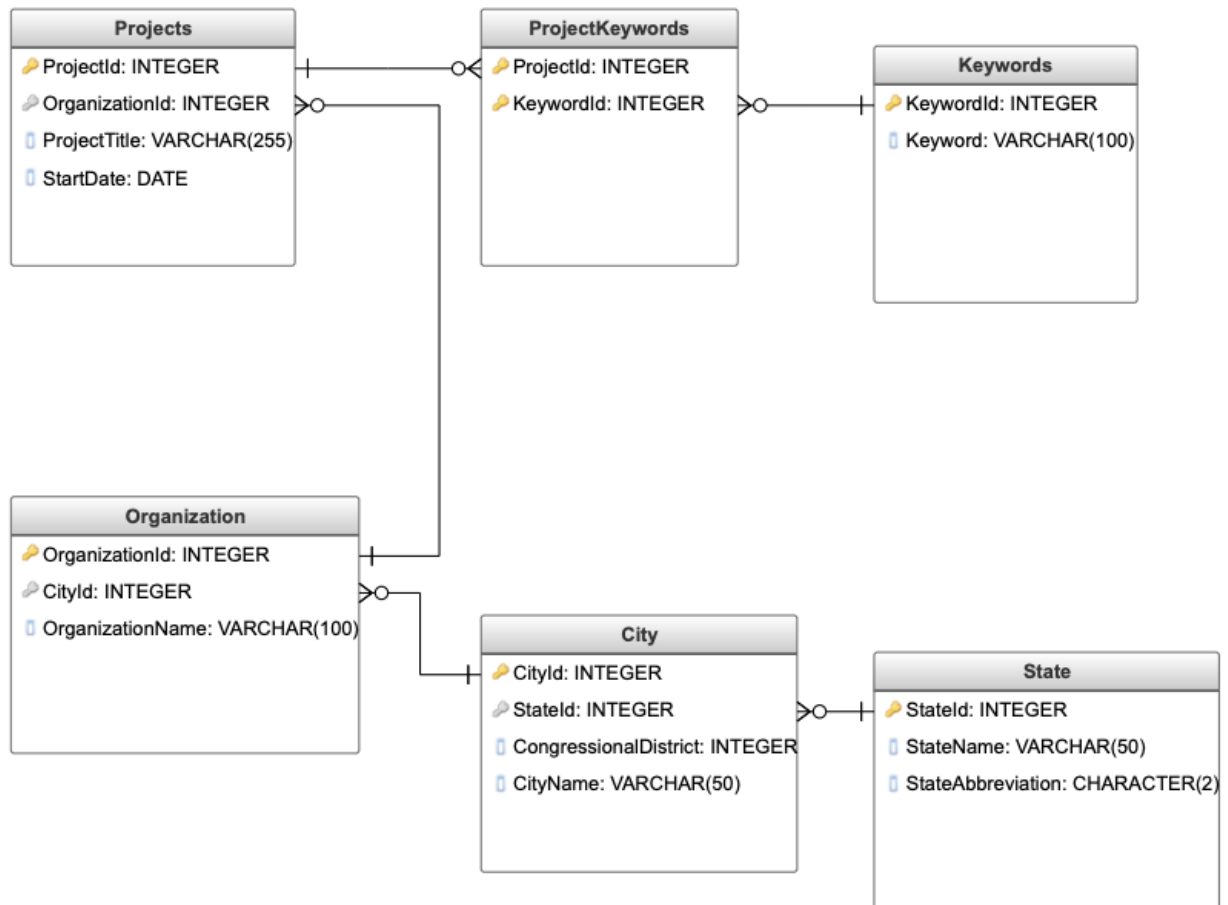
It is reasonable to state,

"The project is dependent on the University of California Riverside which, in turn, is defined by the city Riverside. Therefore the ProjectId is dependent on the city of Riverside."

The columns ProjectId, Organization, and City are parts of a transitive dependency.

Are there other transitive dependencies that exist between these columns?

Adjusted to 3NF Standards



Exercise

+ NIH Portfolio (Google Sheets)

ProjectId	ProjectTitle	Keywords	StartDate	Organization	City	State	District	PI	OtherPI
9001	Developing reciprocal chromosomal translocations for wild population replacement ...	breeding; disorder prevention; fitness; insect genetics; mediating; population genetics; positioning attribute; structure; zika	12/19/16	Univ of California Riverside	Riverside	CA		41	Alkbari, Omar
9002	Zika virus evolutionary dynamics in host adaptation	biological; disease; disease outbreaks; event; laboratories; laboratory experiment; locales; methods; mosquito control; neurologic; perinatal; zika	2/7/17	Univ of Wisconsin-Madison	Madison	WI		2	Aliota, Matthew
9006	A Rapid and Specific Diagnostic for Immunoglobulin Response to Zika Virus Exposure ...	capsid proteins; color; development; diagnostic reagent; fetus; lateral; link; paper; pathogen; point of care; staging; zika	8/20/16	University of Washington	Seattle	WA		7	Baker, David Yager, Paul
9007	Discovering host factors impacting ZIKV infection via forward genetic screens	andes virus; human genome; insertional mutagenesis; pathogen; response; venezuela; viral; west nile; zika	1/1/17	University of Pennsylvania	Philadelphia	PA		2	Bates, Paul
9008	Molecular and Antibody Detection of Zika Virus in Saliva at the Point of Care	design; development; heating; infection; laboratory facility; neurologic; pennsylvania; point-of-care diagnostics; tool; zika	9/9/16	University of Pennsylvania	Philadelphia	PA		2	Bau, Haim
9009	Zika virus in the human genital tract and implications for transmission	care seeking; case study; epidemic; nicaraguan; novel; pregnancy; vero cells; zika	2/15/17	Univ of North Carolina Chapel Hill	Chapel Hill	NC		4	Becker-Dreps, Sylvia Bucardo, Filemon
9133	Mechanisms of sexual Zika virus transmission and early immunopathogenesis	cell type; dendritic cells; injection of therapeutic agent; transmitted diseases; viral transmission; virus replication; zika	12/1/16	University of Washington	Seattle	WA		7	Vojtech, Lucia
9085	A New Filter Paper Technology for Flavivirus Collection, Shipping, and Analysis	chemicals; collection; dna; eligibility determination; formulation; mutate; new technology; small business innovation research grant; zika	3/8/17	GenTegra, LLC	Pleasanton	CA		15	Nasarabadi, Shanavaz Hogan, Michael; Langenbach, Kurt

Column Descriptions

- **ProjectId:** A unique number assigned to each project number.
- **ProjectTitle:** The grantee submitted title descriptive to the project. **Keywords:** Text used for site searches taken from project titles, abstracts, and scientific terms.
- **StartDate:** The date the project began
- **PI & OtherPI:** Individual(s) designated by the grantee to direct the project or activity being supported by the grant. He or she is responsible and accountable to the grantee and NIH for the proper conduct of the project or activity. Also known as Program Director or Project Director.
- **Organization:** A generic term used to refer to an educational institution or other entity, including an individual, which applies for or receives an NIH grant, contract, or cooperative agreement.
- **City:** The business address city of the grantee organization.

- **State:** The business address state of the grantee organization.
- **District:** Congressional District: The congressional district to which each grant is assigned is based on the business address of the grantee organization.

The table above is our example table with two additional columns, PI and OtherPI.

1. Modify the diagram created for the original table to include PI and OtherPI while observing the three normal forms.
2. Let's suppose, a policy to discourage usage of the term 'swine flu' mandates that all public sites within your organization's purview replace instances of this phrase with 'Influenza A (H1N1)'. Given that your database is properly normalized, how would you remove and replace all keyword entries of "swine flu" with 'Influenza A (H1N1)'? Let's assume that there originally is no 'Influenza A (H1N1)' entry; you don't need to worry about merging existing occurrences.

Conclusion

"Colorless green ideas sleep furiously"

-- Noam Chomsky

"Colorless green ideas sleep furiously" is an example by the linguist Noam Chomsky of a sentence that is grammatically correct, but semantically nonsensical. Similarly, a formulaic application of normal forms is likely to result in the same outcome; a database design that is technically and structurally sound but is senseless in the context of the subject matter to which it is applied and information it houses.

The cardinal rule of database design, and arguably of any engineering effort, is that form should naturally reflect the subject for which it is intended. An intuitive design, elegance, and simplicity are always prized above technical configuration.

I believe the forms presented in this section should be used as the criterion rather than the procedure. Planning is an iterative effort. Each cycle begins with an intuitive design and ends with a technical scrutinization. When conflicts are discovered the cycle should begin again with a revisit to the relevant segment.

Ternary Logic

Introduction

Consider the question "Is it raining on Easter Island?"

Chances are most people cannot immediately answer this question with a definitive yes or no (especially without the use of Google). And, as a practical truth, the collection of information sets are never 100% complete.

Modern databases take this into account through the use of **ternary logic**.

Three-valued logic

In logic, **ternary** logic, also called **three-valued** logic, is any of several many-valued logic systems in which there are three truth values indicating *true*, *false* and a third value representing an *unknown* result.

This is contrasted with the more commonly known boolean logic which provides only for *true* and *false* values and results.

Truth Tables

Truth tables are used to summarize the results for expressions in logic. It codifies the outcomes for all legitimate input values. Most people are familiar with the boolean logic tables. Ternary logic tables differ only in that they include the third *unknown* value.

The most common logical operations are *not*, *and*, & *or*.

A	$\neg A$	NOT A
T	F	If A is true then NOT A is false
F	T	If A is false then NOT A is true
U	U	If A is unknown (not known to be true or false), then NOT A is (still) unknown

A \wedge B		B			A AND B		
		T	F	U			
A	T	T	F	U	If A is true and B is true, then A AND B is true	If A is true and B is false, then A AND B is false	If A is true and B is unknown (not known to be true or false), then A AND B is unknown
	F	F	F	F	If A is false and B is true, then A AND B is false	If A is false and B is false, then A AND B is false	If A is false and B is unknown (not known to be true or false), then A AND B is false
	U	U	F	U	If A is unknown (not known to be true or false) and B is true, then A AND B is unknown	If A is unknown (not known to be true or false) and B is false, then A AND B is false	If both A and B are unknown unknown (not known to be true or false), then A AND B is unknown

A \vee B		B			A OR B		
		T	F	U			
A	T	T	T	T	If A is true and B is true, then A OR B is true	If A is true and B is false, then A OR B is true	If A is true and B is unknown (not known to be true or false), then A OR B is true
	F	T	F	U	If A is false and B is true, then A OR B is true	If A is false and B is false, then A OR B is false	If A is false and B is unknown (not known to be true or false), then A OR B is unknown
	U	T	U	U	If A is unknown (not known to be true or false) and B is true, then A OR B is true	If A is unknown (not known to be true or false) and B is false, then A OR B is unknown	If both A and B are unknown unknown (not known to be true or false), then A OR B is unknown

A Couple of Thought Experiments

false AND unknown = false

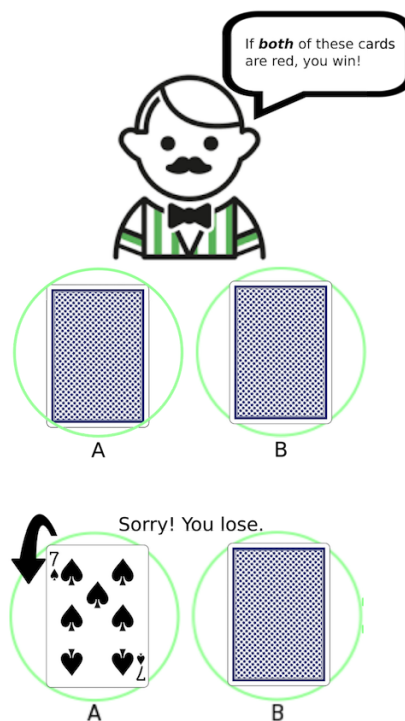
true OR unknown = true

Most of the ternary truth table outcomes can be accepted superficially. However, there are a couple of assertions which may not be clear at first glance. Specifically, why are we able to derive a known result from **false AND unknown** and **true OR unknown**. Instinctively, it would seem that any equation with an unknown value would preclude a definite result.

How can we add unknown to something and get a known result?

Why Is False AND Unknown Equal False?

Let's suppose we're in Las Vegas playing at a card table. The rules are simple. The player places a bet. The dealer shuffles a deck of cards, then deals two cards on the table. If both cards are red, either the suit of hearts or diamonds, then the player wins and doubles his or her wager amount. Otherwise, the player loses the bet placed.

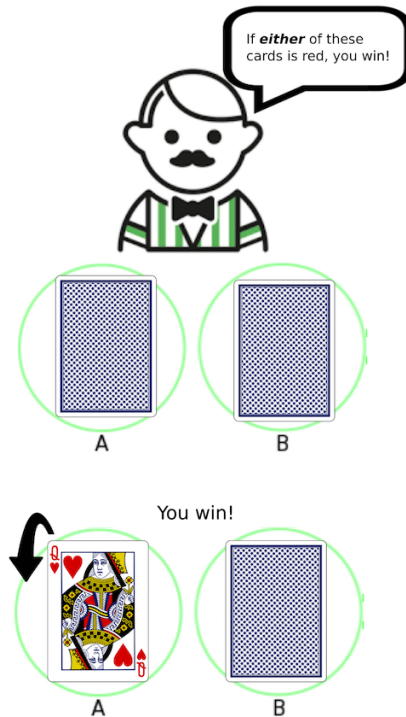


Even though the color of the second card is still unknown, we know we've lost. This is because the facedown card will not change the outcome.

This is why False AND Unknown must evaluate to False.

Why does True OR Unknown Equal True?

After losing at this game, we decide to try our luck at a different game. We find a game that is similar to the first, but the rules are slightly changed. The player is dealt two cards, but in this game, the player wins if *either* card is red.



The color of the second doesn't need to be known to determine the outcome. This is similar to the prior game, but with a happier outcome.

This is the result of True OR Unknown evaluating to True.

Discussion

Discuss how the situation of these games are adjusted to illustrate the statements:

- If A is unknown and B is true, then A OR B is true
- If A is unknown and B is false, then A AND B is false

Conclusion

Ternary logic may at first feel academic. However, familiarity with these principles is essential to both the design and use of databases. It is important to understand that ternary logic seeks to more accurately reflect real-world situations under which data is stored, queried, and interpreted.

Database Predicates

Introduction

Familiarity with the principles of ternary logic is essential to both the design and use of databases. It is important to understand that ternary logic seeks to more accurately reflect real-world situations under which data is collected.

Consider the following item asked in standard United States blood donor eligibility questionnaires:

From January 1, 1980, through December 31, 1996, have you spent (visited or lived) a cumulative time of 3 months or more, in the United Kingdom (UK)?

☐ Yes ☐ No

This example illustrates the potential of a true/false question with three possible outcomes. The first two possible values are fairly obvious; the result may be true or false depending on the selected option. But what if the question was left unanswered? How would this be indicated in the database? This is where the ternary value **unknown** becomes applicable.

The three possible ways to record how the example question was answered are:

1. **true** - Yes, I spent 3 months or more in the UK from Jan 1980 to Dec 1996.
2. **false** - No, I did not spend the specified amount of time within the specified date range in the UK
3. **unknown** - if the answer is ambiguous and cannot be determined as *true* or *false*, e.g. neither option was checked

It is important to note that it is common practice for software developers and database administrators to avoid unknown (also known as null) values by using appropriate default values. However, default values are highly context specific therefore applied by explicit declaration by individuals with subject matter expertise.

In this section we will apply ternary logic to the evaluation database selection statements also known as **predicates**.

Exercise: Clinical Report Form

Assessment of Severe Malaria (For children only)

A diagnosis of severe malaria in children is typically based on a polythetic type classification system. Polythetic refers to the fact that a diagnosis is defined by multiple symptoms, and not all listed symptoms are necessary to classify an individual with the severe malaria criteria. Rather, it is sufficient that a number of symptoms must be observed to consider the diagnosis present.

Clinical Presentations	Yes	No
1. Can the child sit unaided?	<input type="checkbox"/>	<input type="checkbox"/>
2. Is the child fitting now?	<input type="checkbox"/>	<input type="checkbox"/>
3. Is the child jaundiced (i.e., is there scleral icterus)?	<input type="checkbox"/>	<input type="checkbox"/>
4. Signs of dehydration: (sunken eyes or decreased skin turgor)	<input type="checkbox"/>	<input type="checkbox"/>

Blantyre Coma Scale

The Blantyre coma scale is derived from questions which assess responsiveness in children. It is used to judge the presentation of malarial coma in children, an indicator of cerebral malaria. A total score of 5 is considered a normal and healthy result. Any score under 5 is considered abnormal with increasing severity towards a score of 0.

5	Assessment	Score
5a.	Eye movements	1 = Directed (e.g., follows mother's face) 0 = Not directed
5b.	Verbal response	2 = Appropriate cry 1 = Moan or inappropriate cry 0 = None
5c.	Best motor response	2 = Localized painful stimulus 1 = Withdraws limb from pain 0 = None specific or absent response

Following is a theoretical set of 6 returned clinical case report forms (CRFs). The values have been recorded to a database verbatim. Assume that no custom data validation has been applied to the data entry process.

Which participants will be returned by the queries that follow?

Participant Id: A				
1	Is the child prostrate?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
2	Is the child fitting now?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
3	Is the child jaundiced?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
4	Signs of dehydration?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
5	5a	Eye movements	1 = Directed 0 = Not directed	1
	5b	Verbal response	2 = Appropriate cry 1 = Moan or inappropriate cry 0 = None	2
	5c	Best motor response	2 = Localized painful stimulus 1 = Withdraws limb from pain 0 = None specific / absent response	2

Participant Id: B				
1	Is the child prostrate?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	
2	Is the child fitting now?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	
3	Is the child jaundiced?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
4	Signs of dehydration?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	
5	5a	Eye movements	1 = Directed 0 = Not directed	✓
	5b	Verbal response	2 = Appropriate cry 1 = Moan or inappropriate cry 0 = None	✓
	5c	Best motor response	2 = Localized painful stimulus 1 = Withdraws limb from pain 0 = None specific / absent response	✓

Participant Id: C				
1	Is the child prostrate?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	
2	Is the child fitting now?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	
3	Is the child jaundiced?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	
4	Signs of dehydration?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	
5	5a	Eye movements	1 = Directed 0 = Not directed	0
	5b	Verbal response	2 = Appropriate cry 1 = Moan or inappropriate cry 0 = None	1
	5c	Best motor response	2 = Localized painful stimulus 1 = Withdraws limb from pain 0 = None specific / absent response	0

Participant Id: D				
1	Is the child prostrate?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
2	Is the child fitting now?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
3	Is the child jaundiced?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
4	Signs of dehydration?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
5	5a	Eye movements	1 = Directed 0 = Not directed	
	5b	Verbal response	2 = Appropriate cry 1 = Moan or inappropriate cry 0 = None	2
	5c	Best motor response	2 = Localized painful stimulus 1 = Withdraws limb from pain 0 = None specific / absent response	0

Participant Id: E				
1	Is the child prostrate?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
2	Is the child fitting now?	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
3	Is the child jaundiced?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
4	Signs of dehydration?	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	
5	5a	Eye movements	1 = Directed 0 = Not directed	2
	5b	Verbal response	2 = Appropriate cry 1 = Moan or inappropriate cry 0 = None	1
	5c	Best motor response	2 = Localized painful stimulus 1 = Withdraws limb from pain 0 = None specific / absent response	2

Participant Id: F				
1	Is the child prostrate?	<input type="checkbox"/> Yes	<input type="checkbox"/> No	
2	Is the child fitting now?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
3	Is the child jaundiced?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
4	Signs of dehydration?	<input type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	
5	5a	Eye movements	1 = Directed 0 = Not directed	0
	5b	Verbal response	2 = Appropriate cry 1 = Moan or inappropriate cry 0 = None	2
	5c	Best motor response	2 = Localized painful stimulus 1 = Withdraws limb from pain 0 = None specific / absent response	1

Query 1:

```

SELECT participant_id
FROM severe_malaria_crfs
WHERE
    (prostrate OR fitting OR jaundiced OR dehydration)
    AND
    (eye_movement + verbal_response + motor_response) <= 3

```

C

Query 2:

```

SELECT participant_id
FROM severe_malaria_crfs
WHERE
    (prostrate AND fitting AND jaundiced AND dehydration)
    OR
    (eye_movement + verbal_response + motor_response) <= 3

```

C & F

Query 3:

```

SELECT participant_id
FROM severe_malaria_crfs
WHERE
    prostrate
    AND
    (fitting OR jaundiced OR dehydration OR (eye_movement + verbal_response +
motor_response) <= 3)

```

B&C

These queries are formatted as SQL, this is covered with more depth in the lab portion. For the purposes of this exercise, evaluate the conditional part of the statement and decide which participant ids will be returned.*

Discussion

Are the following two questions equivalent?

1. Is the child prostrate? ☐ Yes ☐ No
2. Is the child prostrate? (*Check for presentation, otherwise leave blank*): ☐ Yes

Conclusion

GIGO is an old acronym in the computer industry standing for "garbage in, garbage out." It is a venerable maxim used by technology experts (sometimes as an excuse) coupling the use of their systems to the quality of the data fed into the system. Frighteningly, its first widespread use may have been the result of an AP reporter's observations while visiting the IRS.

Needless to say, database administrators proactively attempt to prevent the potential ambiguities and problems that crop up from these situations. Modern relational database implementations have many mechanisms to provide *data consistency*. This may be a constraint that prevents unknown or null values from being committed in the system. However, implementing safeguards and rules to a data-sharing system is a double-edged sword. There is the danger of defining a course of rules which results in a rigid and inflexible container. And there is a danger that enforcing consistency may inherently reduce other desirable characteristics. One instance of this phenomenon is asserted by the CAP Theorem.

CAP Theorem

Introduction

Relational database systems are designed with data consistency as their primary dogma. Database consistency, loosely defined, is a guarantee that every transaction returns a reliable outcome. This means that all transactions are applied only after prior requests have been fully completed and other requests wait until the current request is complete. However, guaranteed consistency is a complex problem. For database systems, this is accomplished by locking access when its state is in flux. This makes performance problematic, especially for large systems with high volumes of concurrent requests.

Relational databases are centralized systems; they are designed to run on single servers. This makes scaling available resources (memory, disk space, etc.) complicated and costly. The architecture of these systems prevents a distributed layout offered by the cloud for many other computer systems.

In 2000, a computer scientist named Eric Brewer presented a conjecture stating that a shared data system will inherently have trade-off characteristics. These characteristics are fundamentally relevant to the operation of databases.

Brewer's Conjecture

The CAP theorem, also known as Brewer's theorem, makes the following assertion. At most only two of the following three characteristics can be true of a shared data system:

1. The state of the data is always consistent
2. A view of the data is always available
3. The system is partitioned or remotely distributed

These components of *consistency* (**C**), *availability* (**A**), and *partitions* (**P**) across shared systems were coined by Brewer as the **CAP Theorem**.

Consistency - After a modification, any read request receives the latest value

Simply put, performing a read operation will return the value of the most recent write operation causing all nodes to return the same data. A system has consistency if a transaction starts with the system in a consistent state, and ends with the system in a consistent state. In this model, a system can (and does) shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error during any stage in the process.

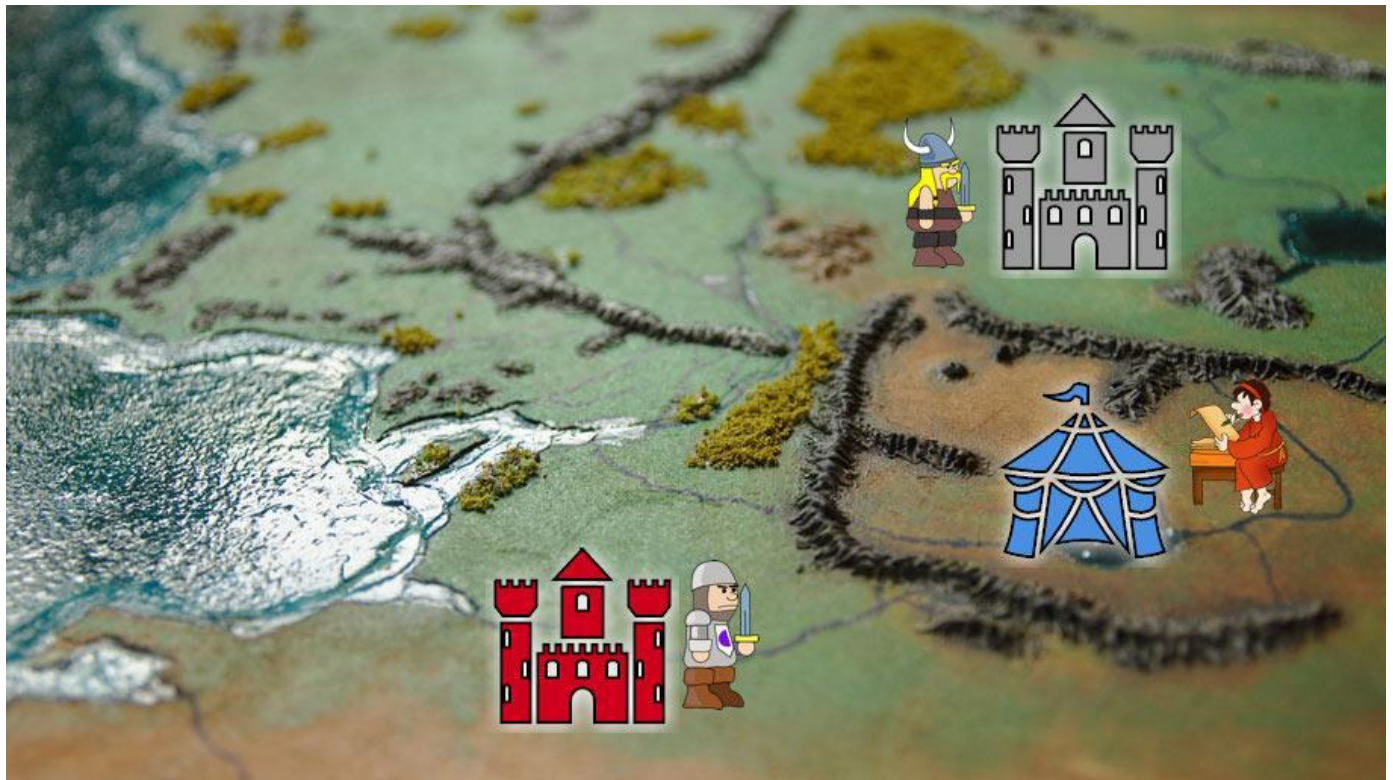
Availability - Every request gets an immediate response.

Achieving availability in a distributed system requires that the system remains operational 100% of the time. Every client gets a timely response to each request, regardless of the state of the system.

Partitions - Requests arrive from remote clients without a direct connection

The internet is inherently partitioned. Messages are sent over a network and therefore are always at risk to be lost. Outages are a common occurrence and occur in separate regions at different times; connections are partitioned.

Illustration - A Tale of Two Kingdoms



ONCE UPON A TIME...

There were two warring kingdoms. The rulers of these lands, King Alfred of Southfolk and King Ivar of Northlandia, decided to make peace. However, years of fighting created a bitter distrust between the two sides. The two leaders decided it was best to call upon a neutral mediator to facilitate the terms of a truce.

After a fortnight of deliberation, the mediator sent both sides an initial draft:

"Northlandia shall give their apple harvest to Southfolk as a tribute of good-will and a gesture of peace."

King Alfred of Southfolk receives the plan and immediately sends the mediator an amendment. His message to the mediator instructs him to delete everything after the 4th word and append ***"fairest princess for marriage to the crown prince of Southfolk"***.

The mediator makes the requested change: ***"Northlandia shall give their ~~apple harvest to Southfolk as a tribute of good-will and a gesture of peace~~ fairest princess for marriage to the crown prince of Southfolk."***

Meanwhile, King Ivar, after pondering the original proposal, also decides to change the terms. The people of Northlandia cannot part with their cherished apples. He counters with

the following instructions. ***"Instead of our apple harvest, we will send the finest Northlandian dairy cows. Change the 4th, 5th, and 6th words from ~~their apple harvest~~ to 30 dairy cows and we'll have peace."***

The honest but simple-minded mediator follows King Ivar's explicit instructions to replace the 4th, 5th, and 6th words of the proposal *he has in hand*. The final version is sent to both rulers. ***"Northlandia shall give 30 dairy cows for marriage to the crown prince of Southfolk."***

Needless to say, King Alfred believed he had been insulted. The war resumed for many more generations.

THE END

This story is a depiction of a non-consistent system. It illustrates a silly but possible consequence when shared data is edited simultaneously by two parties.

Discussion

- How could the mediator enforce consistency?
- If consistency is observed, how would this change availability in the context of the story?
- How does partitioning apply to this situation?
- How would the story change if they were not partitioned?

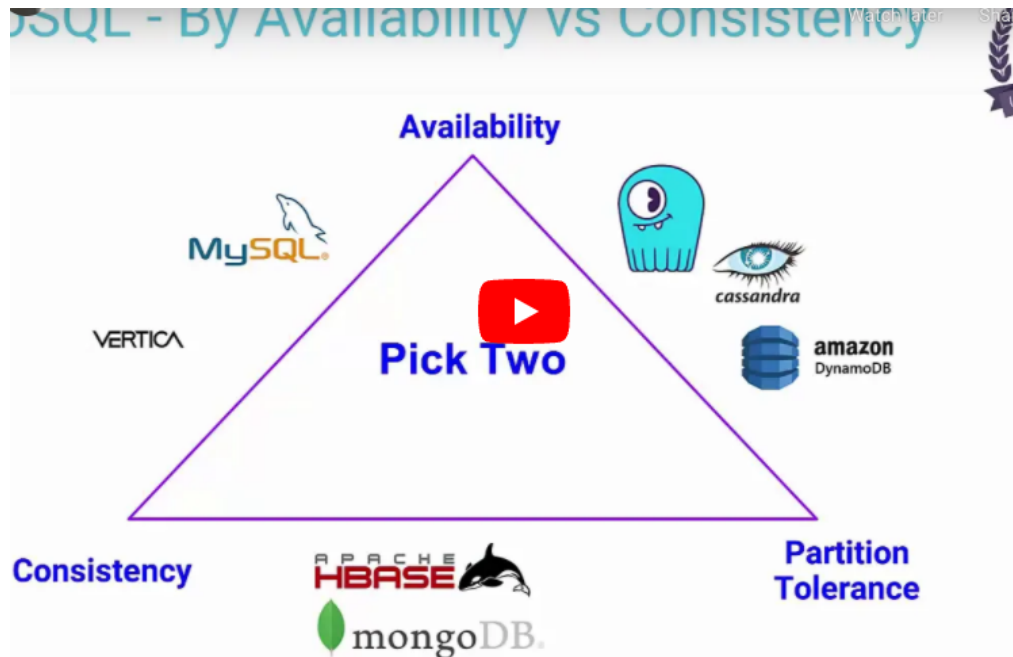
Relevance to Modern Database Systems

Nearly all digital transactions today are done over computer networks and the internet. This means that the partitioned characteristic in the CAP theorem is a given. The consequence is that database systems must choose between consistency or availability.

If this was a game of musical chairs, then there is only one chair left and the last two standing contestants are Consistency and Availability.

[YouTube: Musical Chairs Video](#)

Networked database systems must be designed for **consistency** or **availability**. They cannot be engineered for both.



<https://university.scylladb.com/courses/scylla-essentials-overview/lessons/introduction/topic/nosql-and-scylla/>

NoSQL databases diverge from the traditional relational database model and provide alternative data models for handling large-scale distributed systems. They aim to address scalability, performance, and flexibility requirements that are challenging to achieve with traditional relational databases.

The CAP theorem suggests that in the event of a network partition (i.e., when communication between nodes is interrupted), a distributed system must choose between consistency and availability. Consistency refers to all nodes in a distributed system having the same view of data at all times, while availability refers to the ability of the system to continue responding to client requests even in the presence of failures or network partitions. Partition tolerance refers to the system's ability to continue operating despite network partitions.

NoSQL databases, as a response to the limitations of traditional databases, often prioritize availability and partition tolerance over strong consistency. This design choice allows them to scale horizontally and handle large amounts of data by distributing it across multiple nodes in a cluster. By relaxing the constraints of strong consistency, NoSQL databases can achieve high availability and fault tolerance, making them suitable for applications that require low-latency and high throughput, such as real-time analytics, content management systems, and large-scale web applications.

Different NoSQL databases adopt different consistency models, which define the level of consistency they provide. Some common consistency models found in NoSQL databases include:

1. **Strong consistency:** Provides linearizability, where all read and write operations appear to be executed atomically and in a sequential order. However, achieving strong consistency in a distributed system often comes at the cost of reduced availability during network partitions.
2. **Eventual consistency:** Allows for temporary inconsistencies between replicas, but guarantees that if no further updates are made to a particular data item, all replicas will eventually converge to the same value. Eventual consistency favors availability over strong consistency and is commonly adopted by distributed NoSQL databases.
3. **Read-your-writes consistency:** Ensures that a read operation always returns the most recent write operation by the same client. This consistency model provides a stronger guarantee than eventual consistency but may still exhibit inconsistencies between replicas.

Modern NoSQL databases, such as Apache Cassandra, MongoDB, and Amazon DynamoDB, offer various consistency models to cater to different application requirements. They provide tunable consistency, allowing developers to choose the desired level of consistency for their specific use cases. This flexibility enables application developers to strike a balance between availability, consistency, and performance based on their unique needs.

In summary, the CAP theorem has influenced the design and implementation of modern NoSQL databases by highlighting the trade-offs between consistency, availability, and partition tolerance in distributed systems. NoSQL databases prioritize availability and partition tolerance, relaxing the constraints of strong consistency to achieve scalability and fault tolerance. They provide different consistency models, giving developers the ability to choose the right balance of consistency and availability for their applications.

Conclusion

High availability is becoming more important. In many situations, availability is more desirable than consistency. A new generation of databases has been architected to favor availability. They are called NoSQL databases.

NoSQL Databases

NoSQL (Not only SQL) is a term used to describe a wide range of database management systems that differ from traditional, relational databases. NoSQL systems are designed to handle large volumes of unstructured or semi-structured data, providing high scalability, flexibility, and performance. There are several types of NoSQL databases, each with its own characteristics, pros, and cons. Let's discuss some of the main types:

Document Databases

Document databases store data in flexible, semi-structured documents, typically using JSON or XML formats. These databases are suitable for handling hierarchical or nested data structures. Examples include MongoDB, CouchDB.

Pros:

- Schema flexibility allows easy changes to data structures.
- High performance for read-intensive workloads.
- Good scalability for large datasets.

Cons:

- Limited support for complex joins and ad-hoc queries.
- Lack of standardization for querying languages.

Key-Value Stores

Key-value stores are the simplest form of NoSQL databases, where each data item is stored as a key-value pair. The value can be a complex data structure or a simple string. Examples include Redis, Riak, Amazon DynamoDB.

Pros:

- Very high read and write performance due to simplicity.
- Scalability across distributed systems.

- Excellent for caching and real-time applications.

Cons:

- Lack of query capabilities beyond simple key-based retrieval.
- Limited support for complex data relationships and transactions.

Columnar Databases

Columnar databases store data in columns rather than rows, which allows for efficient data compression and fast querying of specific columns. Examples include Apache Cassandra, HBase.

Pros:

- Excellent scalability and fault tolerance.
- High performance for read and write operations.
- Well-suited for handling large datasets and time-series data.

Cons:

- Less efficient for transactional workloads and updates to individual rows.
- Limited support for complex joins and ad-hoc queries.

Graph Databases

Graph databases focus on storing and querying relationships between entities, represented as nodes and edges. They are used for applications that heavily rely on complex network and relationship analysis. Examples include Neo4j, Amazon Neptune.

Pros:

- Efficient for traversing and querying complex relationships.
- Suitable for social networks, recommendation systems, and fraud detection.
- Flexible data model for evolving relationships.

Cons:

- Less performant for simple read and write operations compared to other types.
- Limited scalability for very large datasets.
- Higher learning curve due to the graph-based query language.

It's worth noting that some NoSQL databases can blur the boundaries between these categories or combine features from multiple types. Additionally, the suitability of a particular NoSQL system depends on the specific use case and requirements of your application.

SQL Lab

We will use downloaded the data files from the National Toxicology Program FTP site and query from this data. The main goal of this lab is to create tables with proper data types, relationships, constraints, and normalization practices.

A database is a collection of data consisting of a physical file residing on a computer. The collection of data in that file is stored in different tables where each row in the table is considered as a record. Every record is broken down into fields that represent single items of data describing a specific thing.

More technically, a database can also be defined as an organized structured object stored on a computer consisting of data and metadata. Data, as previously explained, is the actual information stored in the database, while metadata is data about the data. Metadata describes the structure of the data itself, such as field length or datatype. For example, in a company database, the value of 6.95 stored in a field is data about the price of a specific product. The information that this is a number data stored to two decimal places and valued in dollars is metadata.

Databases are usually associated with software that allows for the data to be updated and queried. The software that manages the database is called a Relational Database Management System (RDBMS). These systems make storing data and returning results easier and more efficient by allowing different questions and commands to be posed to the database. Popular RDBMS software includes Oracle Database, Microsoft SQL Server, MySQL, and IBM DB2. Commonly, the RDBMS software itself is referred to as a database, although theoretically, this would be a slight misnomer. When working with databases we will participate in the design, maintenance, and administration of the database that supplies data to our website or application. In order to do this, however, we will need to access that data and also automate the process to allow other users to retrieve

and perhaps even modify data without technical knowledge. To achieve this we will need to communicate with the database in a language it can interpret. Structured Query Language (SQL) will allow us to directly communicate with databases and is thus the subject of this book. We will learn the basics of SQL. SQL is composed of commands that enable users to create database and table structures, perform various types of data manipulation and data administration, and query the database in order to extract useful information.

SQL Language

The Structured Query Language (SQL) is the standard language for relational database management systems (RDBMS). The language has standards developed by the American

National Standards Institute (ANSI) and the International Organization for Standardization (ISO). However, most (if not all) vendors use proprietary keywords and formats which do not allow code to be portable without modifications.

Relational databases heavily borrow concepts from two areas of mathematics, set theory, and predicate logic. This is reflected in the SQL language. While an explanation of mathematics is beyond the scope of this course, it is important to highlight one of the more impactful consequences.

Relational databases and SQL utilize ternary logic; values, regardless of their data type, may be set as unknown. As a consequence, conditional expressions may evaluate as true, false, or unknown. In SQL the unknown value is represented as NULL.

Terminology

There are often slight variances in how technical terms are defined dependent on individual platforms. Since we will be using MySQL the terminologies used in this lab will prefer this context.

Database & Schema: Most database vendors define the schema as a collection of tables and a database as a collection of schemas. MySQL, however, uses database and schema interchangeably with the former definition; a MySQL schema and database refer to a collection of tables. The MySQL documentation uses the plural form of schema (schemas) as its collection.

Table Row Column Statement Query Result set Predicate: specifies conditions that can be evaluated to one of true, false, or NULL.

SQL statements may be classified into the following categories:

Data Query Language (DQL): Statements that query and do not modify data. These statements begin with SELECT.

Data Manipulation Language (DML): Statements that modify data. These statements generally begin with INSERT, UPDATE, DELETE.

Data Definition Language (DDL): Statements that create and modify the schema. DQL and DML process data within the schema, DDL changes the structures which contain that data. These statements usually begin with CREATE, ALTER, or DROP.

Common SQL Commands

Command	Description
SELECT	extract database data
UPDATE	updates database data

DELETE	deletes database data
INSERT	inserts new database data
CREATE DATABASE	creates new database
ALTER DATABASE	modifies database
CREATE TABLE	creates a table
ALTER TABLE	modifies a table
DROP TABLE	removes or deletes a table
CREATE INDEX	creates search key or index
DROP INDEX	deletes an index

NTP Data Collections: Statistically Analyzed NTP Pathology Lesions (with Incidence)

The [Pathology Lesions dataset and its description](#) used by the subsequent exercises are available for download at the [NTP FTP](#) site.

Downloading the data

The url for this data collection is:

[ftp://anonftp.niehs.nih.gov/ntp-cebs/datatype/NTP_Data_Collections/Statistically Analyzed NTP Pathology Lesions with Incidence 2020 03 05.xlsx](ftp://anonftp.niehs.nih.gov/ntp-cebs/datatype/NTP_Data_Collections/Statistically_Analyzed_NTP_Pathology_Lesions_with_Incidence_2020_03_05.xlsx)

Examining the data

Statistically_Analyzed_NTP_Pathology_Lesions_with_Incidence_2020_03_05-100

The first step to creating a database is to gain familiarity and understanding of the data to be housed. Consider the following questions for discussion:

- Are there any inherent relationships within the columns?
- Which columns within the datasets are good candidates for keys?
- Which data types should we use for various columns – integer, float, date/time, bit, etc.?
- Are there any constraints we might expect to exist within the data, e.g. which columns should always have values, which are optional, which should be unique, are there any implied values?
- Are there opportunities for normalization?

Querying from a Table

The SQL **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Retrieve all rows from a table and return the specified columns as the result set.

```
SELECT <column> [,<column>...]  
FROM <table>;
```

Example:

```
SELECT species, sex  
FROM lesions;
```

Retrieve all rows from a table and return all columns as the result set.

```
SELECT *  
FROM <table>;
```

Example:

```
SELECT *  
FROM lesions;
```

Retrieve a result set with a condition.

```
SELECT <column> [,<column>...]  
FROM <table>  
WHERE <condition>;
```

Example:

```
SELECT species, sex
FROM lesions
WHERE sex = 'Male';
```

Retrieve rows from a table and return a unique result set.

```
SELECT DISTINCT <column> [, <column>...]
FROM <table>
```

Example:

```
SELECT DISTINCT sex FROM lesions;
```

Retrieve rows from a table and sort the result set.

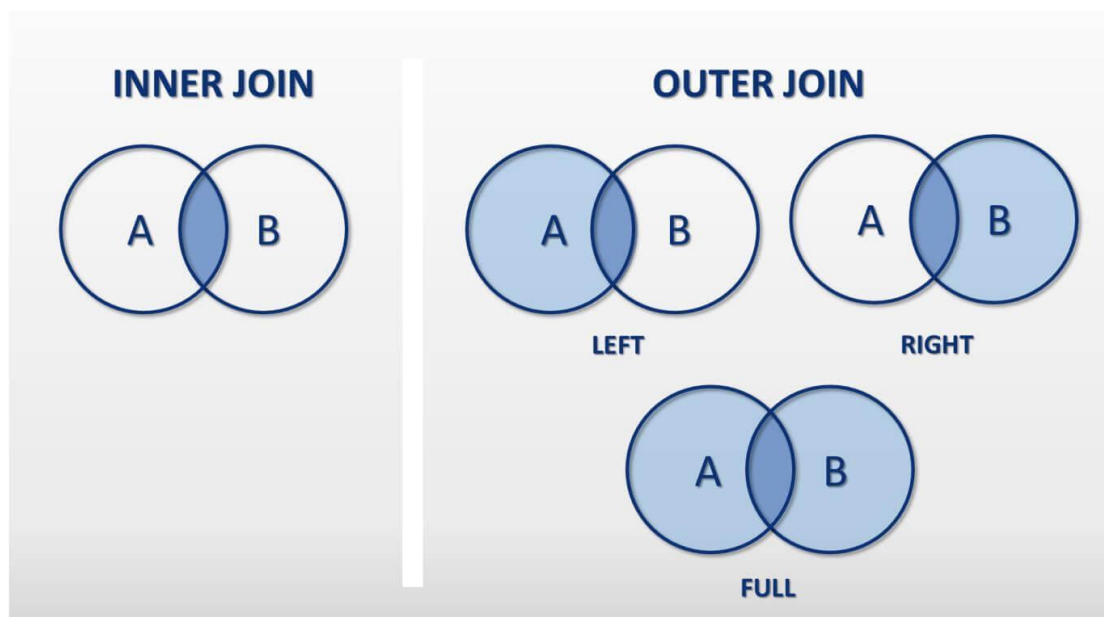
```
SELECT <column> [, <column>...]
FROM <table>
ORDER BY <column> [ASC|DESC] [, <column> [ASC|DESC]...];
```

Example:

```
SELECT organ, sex
FROM lesions
ORDER BY organ, sex DESC
```

Join Multiple Tables

The SQL join clause is used to combine rows from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each. Most relational databases support four basic join types: INNER, LEFT JOIN, RIGHT JOIN, and OUTER JOINS. Note: MySQL does not support outer joins.



Let's say you have a Students table and a Lockers table. In SQL, the first table you specify in a join, Students, is the LEFT table, and the second one, Lockers, is the RIGHT table.

Each student can be assigned to a locker, so there is a locker number column in the student table. More than one student could potentially be in a single locker, but especially at the beginning of the school year, you may have some incoming students without lockers and some lockers that have no students assigned.

For the sake of this example, let's say you have 100 students, 70 of which have lockers. You have a total of 50 lockers, 40 of which have at least 1 student and 10 lockers have no student.

INNER JOIN is equivalent to "show me all students with lockers".

Any students without lockers, or any lockers without students are missing.

Returns 70 rows

LEFT OUTER JOIN would be "show me all students, with their corresponding locker if they have one".

This might be a general student list, or could be used to identify students with no locker.

Returns 100 rows

RIGHT OUTER JOIN would be "show me all lockers, and the students assigned to them if there are any".

This could be used to identify lockers that have no students assigned, or lockers that have too many students.

Returns 80 rows (list of 70 students who have lockers, plus the 10 lockers with no student)

FULL OUTER JOIN would be silly and probably not much use.

Something like "show me all students and all lockers, and match them up where you can"

Returns 110 rows (all 100 students, including those without lockers. Plus the 10 lockers with no student)

For the following exercises, we will be using the `join_example` database

Retrieve rows from two tables with an INNER JOIN

```
SELECT <column> [,<column>...]  
FROM <table1>  
INNER JOIN <table2> ON <condition>;
```

Example:

```
SELECT  
    *  
FROM  
    students  
INNER JOIN  
    lockers  
ON  
    students.locker_num = lockers.locker_num
```

Retrieve rows from two tables with a LEFT OUTER JOIN

```
SELECT <column> [,<column>...]  
FROM <table1>  
LEFT JOIN <table2> ON <condition>;
```

Example:

```
SELECT *  
FROM    students  
        LEFT JOIN lockers  
            ON students.locker_num = lockers.locker_num
```

Retrieve rows from two tables with an RIGHT OUTER JOIN

```
SELECT <column> [,<column>...]  
FROM <table1>  
RIGHT JOIN <table2> ON <condition>;
```

Example:

```
SELECT *  
FROM   students  
       RIGHT JOIN lockers  
         ON students.locker_num = lockers.locker_num
```

Query Aggregates

Aggregate functions compute a result against a column within a result set. These functions are typically statistical functions.

Retrieve rows from a table and return an aggregated result set.

```
SELECT <aggregate>(<column>) [, <column>...]  
FROM<table>  
GROUP BY <column> [, <column>...]
```

Example:

```
SELECT organ, COUNT(lesions_id) AS lesions_count  
FROM lesions  
GROUP BY organ;
```

Retrieve rows from a table and return a filtered aggregated result set.

```
SELECT <aggregate>(<column>) [, <column>...]  
FROM<table>  
GROUP BY <column> [, <column>...]  
HAVING <condition>
```

Example:

```
SELECT  
    organ,  
    sex,  
    COUNT(lesions_id) AS lesions_count  
FROM  
    lesions  
GROUP BY  
    organ,  
    sex  
HAVING
```



```
sex = 'Male'
```

SQL operators

Using UNION, wild card patterns, lists, BETWEEN, and NULL

Combine rows from two queries

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```

Example:

```
SELECT *
FROM lesions
WHERE sex = 'MALE'

UNION

SELECT *
FROM lesions
WHERE sex = 'FEMALE'
```

Query rows using pattern matching %, _

```
SELECT c1, c2
FROM t1
WHERE c1 [NOT] LIKE pattern;
```

Example:

```
SELECT *
```

```
FROM lesions
WHERE sex LIKE '%MALE'
```

Query rows in a list

```
SELECT c1, c2
FROM t
WHERE c1 [NOT] IN (list);
```

Example:

```
SELECT *
FROM lesions
WHERE species IN ( 'Hamster', 'Rat' )
```

Query rows between two values

```
SELECT c1, c2
FROM t
WHERE c1 BETWEEN low AND high;
```

Example:

```
SELECT *
FROM lesions
WHERE number_of_animals_with_tumors BETWEEN 100 AND 110
```

Check if values in a table is NULL or not

```
SELECT c1, c2
FROM t
WHERE c1 IS [NOT] NULL;
```

Example:

```
SELECT *  
FROM lesions  
WHERE is_trend_significant IS NULL
```

Modifying data

For the next few exercises, we will be manipulating data. Let's create a copy of the lesions table to play with.

```
CREATE TABLE lesions_copy  
  
    SELECT *  
  
    FROM    lesions
```

Insert one row into a table

```
INSERT INTO t (column_list) VALUES (values);
```

Example:

```
INSERT INTO lesions_copy  
    (chemical_name,  
     species,  
     sex,  
     organ)  
VALUES ('Test',  
       'Rat',  
       'FEMALE',  
       'Liver')
```

Delete subset of rows in a table

```
DELETE FROM t WHERE condition;
```

Example:

```
DELETE FROM lesions_copy  
WHERE species = 'Rat'
```

Insert rows from t2 into t1

```
INSERT INTO t1 (columns)  
SELECT column_list FROM t2;
```

Example:

```
INSERT INTO lesions_copy  
SELECT *  
FROM lesions  
WHERE species = 'Rat'
```

Update values in the column c1, c2 that match the condition

```
UPDATE t SET c1=value, c2=value WHERE condition;
```

Example:

```
UPDATE lesions_copy SET species = 'Mus' WHERE species = 'Mouse'
```

Managing a schema

Create a new table

```
CREATE TABLE t (  
    species_id TINYINT UNSIGNED PRIMARY KEY,  
    species VARCHAR(15) NOT NULL  
);
```

Delete a table from the schema

```
DROP TABLE t;
```

Create an index

```
CREATE INDEX <index name> ON <table> (<column>);
```

```
CREATE INDEX idx_chemical_name ON lesions (`chemical_name`);
```

Delete an index

```
DROP INDEX <index name> ON <table>;
```

```
DROP INDEX idx_chemical_name ON lesions;
```

Exercise: Normalizing and Adjusting the NTP Dataset

In this section we will use several concepts reviewed in this course to further normalize the NTP lesions dataset. Let's start by going over the steps used to normalize the species column in the lesions table.

```
/*
  Create a new species table
*/
CREATE TABLE species (
  species_id TINYINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  species_name VARCHAR(10) NOT NULL,
  UNIQUE KEY unq_species_name (species_name)
);

/*
  Populate the species table
*/
INSERT INTO species (species_name)
SELECT DISTINCT species FROM lesions
ORDER BY species;

/*
  Create a species_id column
*/
ALTER TABLE lesions
ADD COLUMN species_id TINYINT UNSIGNED NOT NULL AFTER species;

/*
  Populate the species_id column
*/
UPDATE lesions
INNER JOIN species ON lesions.species = species.species_name
SET lesions.species_id = species.species_id;

/*
  Create a foreign key index
```



```

*/
ALTER TABLE lesions
ADD CONSTRAINT fk_species_id
FOREIGN KEY (species_id) REFERENCES species(species_id)
ON DELETE RESTRICT
ON UPDATE CASCADE;

```

```

/*
  Remove the original species column
*/
ALTER TABLE lesions
DROP COLUMN species;

```

Normalize the columns **chemical_name**, **organ**, and **morphology**. For each, follow these general steps:

1. Count the number of distinct values for the column to be normalized
2. Create a new table with the smallest data type for the primary key
3. Copy the distinct values to the new table
4. Add a new foreign key column to the lesions table (used to reference the values table)
5. Update the new reference column lesions table using an update and join
6. Drop the original non-normal column

Connecting to MySQL with R

We will use the RMariaDB package to connect to MySQL. The documentation for the RMariaDB package is [here](#).

Ensure the library RMariaDB is installed. Open an R console and execute the following:

```
install.packages("RMariaDB")
```

Connecting to the database

We will connect to the **ntp** MySQL database, test the connection by listing the available tables, then close the connection.

In the R console type the following commands:

```
install.packages("RMySQL")
```

```
library(RMySQL)
```

```
# establish the initial connection
```

```
ntpDbCon <- dbConnect(MySQL(), user='mysqluser', password='101Aardvarks',  
dbname='ntp', host='localhost')
```

```
# list the tables
```

```
dbListTables(ntpDbCon)
```

```
# disconnect
```

```
dbDisconnect(ntpDbCon)
```

SQL with R

R provides multiple packages to interface and query the database. We will use DBI and pass SQL string statements to the remote database instance. You can query your data with DBI by using the `dbGetQuery()` function. Simply paste your SQL code into the R function as a quoted string. This method is sometimes referred to as *pass through SQL code*, and is probably the simplest way to query your data.

```
# install.packages("RMySQL")

library(RMySQL)

# establish the initial connection
ntpDbCon <- dbConnect(MySQL(), user='mysqluser',
password='101Aardvarks', dbname='ntp', host='localhost')

# select from the lesions table
dbGetQuery(ntpDbCon, 'SELECT DISTINCT strain FROM lesions')

# disconnect
dbDisconnect(ntpDbCon)
```


SQL with Python

The following example shows how to [query](#) data using a cursor created using the connection's [cursor\(\)](#) method. The data returned is formatted and printed on the console.

1. Use the `mysql.connector.connect()` method of MySQL Connector Python with required parameters to connect MySQL.
2. Use the connection object returned by a `connect()` method to create a **cursor** object to perform Database Operations.
3. The `cursor.execute()` to execute SQL queries from Python.
4. Close the Cursor object using a `cursor.close()` and MySQL database connection using `connection.close()` after your work completes.

```
import mysql.connector

cnx = mysql.connector.connect(user='mysqluser', password='101Aardvarks',
                             host='localhost',
                             database='ntp')

cursor = cnx.cursor()

query = ("SELECT morphology, organ FROM lesions LIMIT 10")

cursor.execute(query)

for (morphology, organ) in cursor:
    print("{} {}".format(morphology, organ))

cursor.close()
cnx.close()
```

We first open a connection to the MySQL server and store the [connection object](#) in the variable `cnx`. We then create a new cursor, by default a [MySQLCursor](#) object, using the connection's [cursor\(\)](#) method.

SQL Injection

SQL injection is the insertion of an SQL query disguised as input from the client to the database. SQL commands are formatted in the input to execute arbitrary SQL commands not intended by the application.

- SQL injection is a code injection technique that might destroy your database.
- SQL injection is one of the most common web hacking techniques.
- SQL injection is the placement of malicious code in SQL statements, via web page input.

Examples of SQL Injection Attacks

```
txtUserId = getRequestString("UserId");  
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

UserId:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

Preventing SQL Injection

Parameterized Queries

A parameterized query is a query in which parameters are used as placeholders and are supplied at execution time. In this type of query, the data types of the parameters are predefined, and in some cases, default values are also set. Doing so causes SQL injection queries to fail.

Stored Procedures

Stored procedures are the SQL statements defined and stored in the database, which are called from the application. Developers normally build SQL statements with parameters that are automatically parameterized. However, developers can generate dynamic SQL queries inside stored procedures that eliminate the risk.

Refrain From Administrative Privileges

Blocking connections with applications to their databases using an account having root access or administrative privilege is also effective protection. This prevents bad actors from gaining access to the whole database. Regardless, a non-administrative account server can also be risky for an application, primarily if it is used in various databases and applications.

Appendix

Importing the Pathology Lesions dataset to MySQL

First, convert the dataset to CSV format.

1. Download [Full Dataset](#) Excel file
2. Convert to comma separated format (CSV). File -> Save as -> File Format: CSV UTF-8 (Comma delimited) (.csv)

We will be using built-in MySQL functions to load the raw flat files into a table. The initial tables will have simple column definitions without any relational features.

1. Open MySQL Workbench and paste the following SQL code into the query tab.
2. Change the path to the CSV file as needed.

```
DROP SCHEMA IF EXISTS ntp;
```

```
CREATE SCHEMA ntp;
```

```
USE ntp;
```

```
DROP TABLE IF EXISTS lesions;
```

```
CREATE TABLE lesions
```

```
(
  lesions_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  chemical_name          VARCHAR(100),
  ntp_publication        VARCHAR(20),
  abstract_url           VARCHAR(110),
  tech_report_url        VARCHAR(50),
  species                VARCHAR(10),
  strain                 VARCHAR(25),
  sex                    ENUM('FEMALE', 'MALE'),
  organ                  VARCHAR(75),
  morphology             VARCHAR(175),
  dose                   VARCHAR(50),
```



```

        dose_unit          VARCHAR(10),
        route              VARCHAR(50),
        is_neoplastic       CHAR(1),
        is_trend_significant CHAR(1),
        is_pairwise_significant CHAR(1),
        number_of_animals_with_tumors SMALLINT UNSIGNED,
        number_of_animals_examined SMALLINT UNSIGNED,
        casrn              VARCHAR(12),
        casrn_url          VARCHAR(75),
        ntp_tdmse_number    VARCHAR(8),
        chemtrack_number    VARCHAR(8)
    );

/*
    Set the csv file path
*/
LOAD DATA LOCAL INFILE
    '/home/ubuntu/Statistically_Analyzed_NTP_Pathology_Lesions_with_Inci
    dence_2020_03_05.csv'
    INTO TABLE lesions
    FIELDS TERMINATED BY ',' ENCLOSED BY '"' IGNORE 1 LINES
    (chemical_name,ntp_publication,abstract_url,tech_report_url,species,
    strain,sex,organ,morphology,dose,dose_unit,route,is_neoplastic,is_tr
    end_significant,is_pairwise_significant,number_of_animals_with_tumor
    s,number_of_animals_examined,casrn,casrn_url,ntp_tdmse_number,chemtr
    ack_number);

/*
    Convert blank values to null
*/
UPDATE
    lesions
    SET
        is_trend_significant = NULL
    WHERE

```

```
is_trend_significant = "
```

```
UPDATE
```

```
  lesions
```

```
SET
```

```
  is_pairwise_significant = NULL
```

```
WHERE
```

```
  is_pairwise_significant = "
```

MySQL Data Types

Type	Size	Description
CHAR(Length)	Length bytes	A fixed-length field from 0 to 255 characters long.
VARCHAR(Length)	String length + 1 bytes	A fixed-length field from 0 to 255 characters long.
TINYTEXT	String length + 1 bytes	A string with a maximum length of 255 characters.
TEXT	String length + 2 bytes	A string with a maximum length of 65,535 characters.
MEDIUMTEXT	String length + 3 bytes	A string with a maximum length of 16,777,215 characters.
LONGTEXT	String length + 4 bytes	A string with a maximum length of 4,294,967,295 characters.
TINYINT	1 byte	Range of -128 to 127 or 0 to 255 unsigned.
SMALLINT	2 bytes	Range of -32,768 to 32,767 or 0 to 65535 unsigned.
MEDIUMINT	3 bytes	Range of -8,388,608 to 8,388,607 or 0 to 16,777,215 unsigned.
INT	4 bytes	Range of -2,147,483,648 to 2,147,483,647 or 0 to 4,294,967,295 unsigned.
BIGINT	8 bytes	Range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 or 0 to 18,446,744,073,709,551,615 unsigned.

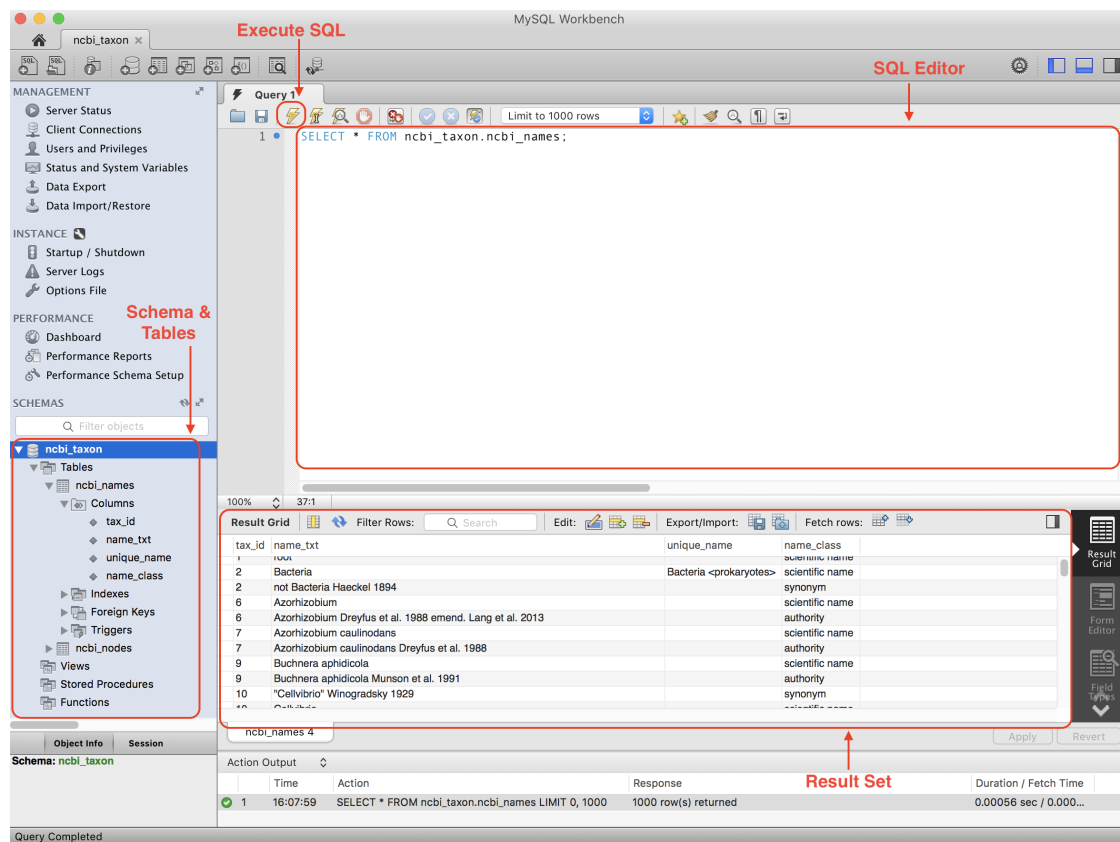
Type	Size	Description
FLOAT	4 bytes	A small number with a floating decimal point.
DOUBLE[Length, Decimals]	8 bytes	A large number with a floating decimal point.
DECIMAL[Length, Decimals]	Length + 1 or Length + 2 bytes	A DOUBLE stored as a string, allowing for a fixed decimal point.
DATE	3 bytes	In the format of YYYY-MM-DD.
DATETIME	8 bytes	In the format of YYYY-MM-DD HH:MM:SS.
TIMESTAMP	4 bytes	In the format of YYYYMMDDHHMMSS; acceptable range ends in the year 2037.
TIME	3 bytes	In the format of HH:MM:SS
ENUM	1 or 2 bytes	Short for enumeration, which means that each column can have one of several possible values.
SET	1, 2, 3, 4, or 8 bytes	Like ENUM except that each column can have more than one of several possible values.

MySQL Workbench

MySQL Workbench is the endorsed integrated development environment for the MySQL server. It features a graphical SQL client along with design and administrative tools specifically for MySQL.

Client and server

MySQL Workbench and the MySQL database are actually two independently running applications. They may not (usually are not) running on the same machine. The client makes a connection to the server and then sends it commands, such as SQL. If the client sends a query, then it will await a reply then display or process the results. Modern RDBMS servers are designed to handle several client connections simultaneously.



Installing MySQL to Your Local Computer

Description

The following are general instructions for installing a local MySQL server and the SQL client tool to your desktop or laptop. Oracle publishes two versions of MySQL, a commercial enterprise edition, and the open source community edition. We will be using the community edition.

These instructions are based on the installation procedures for MySQL server 5.7 and MySQL Workbench 6.3.

Instructions for MySQL Community Server

MySQL server is the backend service which maintains the databases you create.

1. Navigate to the MySQL Community server download page.
2. Select the operating system appropriate for your desktop or laptop.
3. Most operating systems will have several installation package options. If you're unsure which package to choose, the first option is usually a safe choice.
4. Initiate the installer or follow the installation procedure for the downloaded package.
5. **Root account password:** Assigning a root MySQL password is required during the install. Record the password you choose; you will need it later.
6. For other prompts during the installation, choose the default option.
- 7.

Instructions for MySQL Workbench

MySQL Workbench is a graphical client tool which allows users to issue SQL statements to the backend server and also features administration tools with an easy to use interface.

1. Navigate to the MySQL Workbench download page.
2. Select the operating system appropriate for your desktop or laptop.
3. The installation procedure is self-guiding. If prompted during the procedure for a choice and you're unsure

which option to choose, select the default value.

Useful Online Resources

The [official reference manual for MySQL](#) is available online.

The [Instant SQL Formatter](#) is useful for formatting and cleaning up long SQL statements.

SQL naming conventions: [How I Write SQL: Naming Conventions](#)

NIEHS Library

Resources from the NIEHS Library

Online video series from SAGE Research Methods

[Data Access and Analytics with MySQL](#) (2018)

Print books

[PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database](#) (2017)

eBooks

[SQL for Data Scientists: A Beginner's Guide for Building Datasets for Analysis](#) (2021)

[SQL for Data Science: Data Cleaning, Wrangling and Analytics with Relational Databases](#) (2020)

[Data Analysis Using SQL and Excel](#) (2016)

Need more resources? Contact the
library at library@niehs.nih.gov!