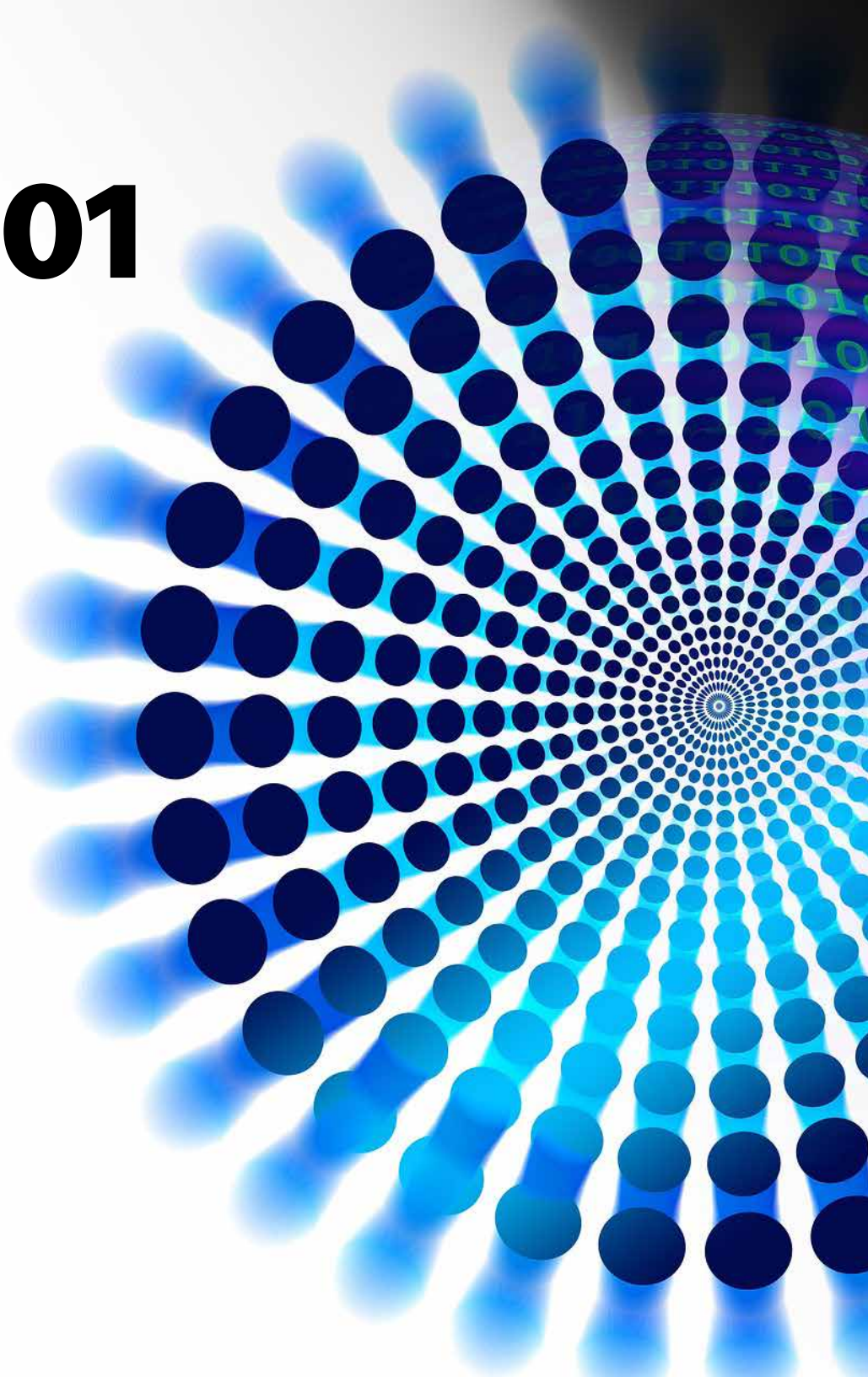
Databases 101

Nicholas Schmidt, CIPP/US



iapp

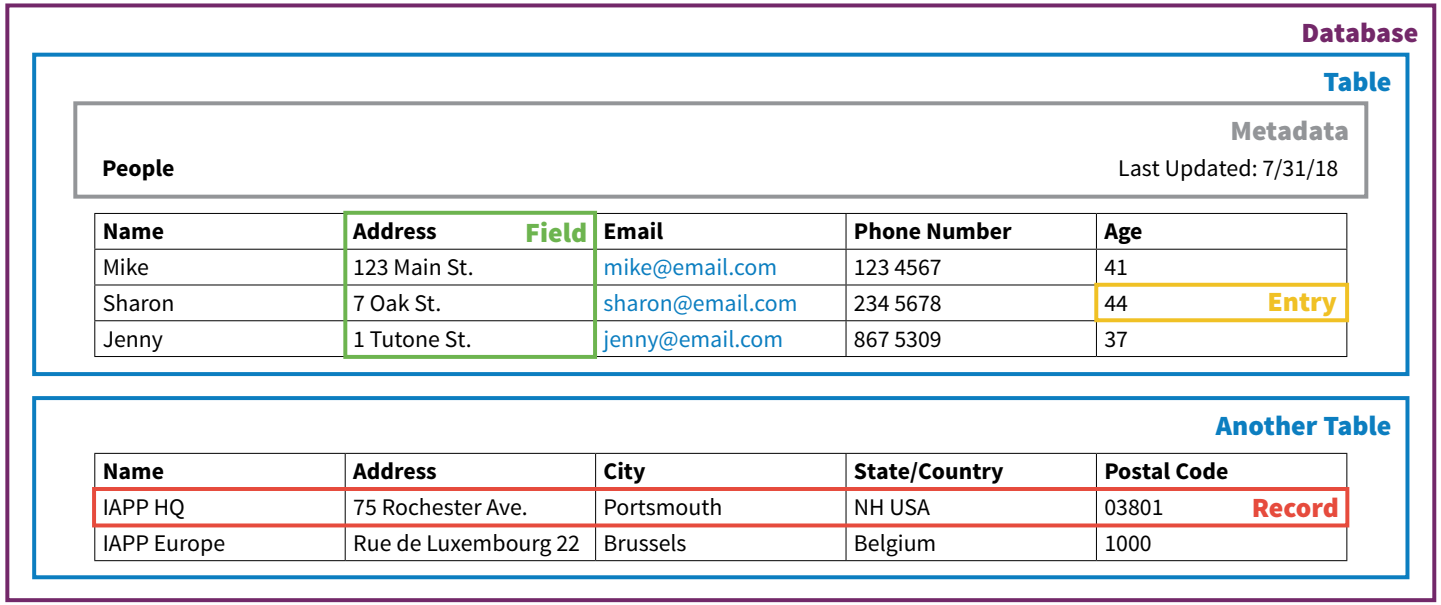
Databases 101

There are few technologies more taken for granted than databases. Since at least the 1960s, databases have been a fundamental part of most major technology products. It is almost inconceivable that you would create any sort of modern, complex program without using a database somewhere.

That doesn't mean, however, that everyone knows how they work, or why. As this first part in a series of white papers on technology for privacy pros, we're going to look at the very basics of the art of database design, with an eye towards helping the busy privacy professional understand at a glance what data an organization is storing and how, and provide some suggestions for ensuring the data is stored consistently and accurately.

Common database terminology

First, it may be helpful to look at some common database terms and what they mean:



Name	Address	Email	Phone Number	Age
Mike	123 Main St.	mike@email.com	123 4567	41
Sharon	7 Oak St.	sharon@email.com	234 5678	44
Jenny	1 Tutone St.	jenny@email.com	867 5309	37

Name	Address	City	State/Country	Postal Code
IAPP HQ	75 Rochester Ave.	Portsmouth	NH USA	03801
IAPP Europe	Rue de Luxembourg 22	Brussels	Belgium	1000

Example SQL Query and Output:
Query: SELECT Address FROM People WHERE Age>40;
Output:
"123 Main St."
"7 Oak St."

Database — A database is, according to the legendary textbook *Fundamentals of Database Systems*, any "collection of related data." In practice, it is a specific type of computer file that holds large amounts of data in a specific format and follows certain conventions. The most common file extension for a programmatic database is ".db." Databases created using Microsoft Access have the default extension ".accdb" or ".accdbx," depending upon the version. The easiest way to picture a database is as a Excel document with some extra features and rules.

Tables — A database table is a specific grouping of related data ("employees," "customers," etc.), a specific spreadsheet in a format that any database table can be exported to. If you are looking to review the contents of a database but cannot peruse the database itself, you will probably receive an export of the contents in spreadsheet form, with one spreadsheet per table.

Fields — A field is a specific category of data within a table ("employee name," "employee address," "customer credit card number," etc.), like a column in a spreadsheet. All entries in a field must have the same format. Broadly, fields can contain either numbers, nominals (i.e. letters and symbols), or booleans (true/false).

Records — A record is a specific group of entries for each field in a table, like a row in a spreadsheet.

Entries — An entry is a specific value for a specific record in a specific field, essentially a specific cell in the spreadsheet. A blank entry is recorded as "NULL," which means that there is no data recorded for the entry in memory. Whether an entry is NULL or filled can also convey information by implication.

Metadata — Metadata is data about data. Common metadata entries include update dates, original creation dates, data size, table/database/field/file names and (for files) extensions. Metadata is crucial to record or discover because it can tell you about when and by whom the data was created or altered.

SQL — Structured Query Language is the standard protocol for database access; most databases allow SQL input. SQL has inspired a number of other query languages, such as C#'s LINQ.

The Fundamental linking structure of a database

Every database table must have a primary key, a field or combination of fields that uniquely identifies a record. No two records in the same table can have the same primary key. Any database implementation that allows the insertion of records with the same primary key is broken and should never be used because of the risk of secondary "shadow" records. Usernames and timestamps are two common primary keys. A third common primary key is a GUID, or "globally unique identifier," which is a base-16 number so long it is guaranteed by probability to be unique. GUIDs are excellent primary keys because they are entirely managed by the computer system and unrelated to the actual information contained in the record. Thus, a record can be completely copied while still maintaining unique primary keys. Occasionally, more than one field in a database is expected to be unique to that record; for example, a database table could contain both an employee's ID number and social security number. In such a case, the unique fields not used in the primary key are referred to as secondary or candidate keys.

1e9b1095-4e52-478f-97d5-67248f5f3a76

An example GUID created by an online generator

If a record in one table must link to another record in the same table or another table, this is accomplished using a foreign key. A foreign key is a field that contains the primary keys of other records. For example, if your database contains a table of employees and a table of universities, and you want to record where each of your employees went to university, you can add a foreign key field to the employee table and populate each entry in that field with the primary key of that employee's alma mater in the university table. The link created between the employee and university tables is called a relation or link.

Every relation has a cardinality ratio. The most common cardinality ratios are listed below:

Relation Name	Description	Example
1:1 (One-to-One)	One record may only link to one record	Spouse-to-Spouse
1:N (One-to-Many)	One record may link to many records	Mother-to-Children
N:1 (Many-to-One)	Many records may link to one record	Children-to-Mother
N:M (Many-to-Many)	Many records may link to many records	Siblings-to-Siblings

Understanding the contained relations and cardinality ratios is crucial to understanding the structure of a database. Where necessary, usually in relationships involving at least one side that is "many," pivot tables can link specific records together.

One common use of relations is weak entity types. A weak entity type is a table with records identified by their relationship to a record in another table. In other words, a foreign key pointing to another table is all or part of the weak entity type's primary key. The relation between the two is the identifying relationship and the identifying table is called the owner entity type. When an owner entity type can connect to multiple weak entity types, the weak entity type will include a partial key that will distinguish specific weak entity types linked to the same owner. Consider the following example from an income tax tracking database, with a 1:N identifying relationship:

Taxpayer Table (Owner Entity Type)	Dependent Table (Weak Entity Type)
Taxpayer ID Number (Primary Key)	Taxpayer (Foreign Key pointing to Taxpayer ID #; Primary Key P. 1)
Name	Name (Unique Partial Key, Primary Key P. 2)
Age	Age
Address	Occupation
Occupation	Income
Income	
Amount Owed	
Last Pay Date	

Because dependents aren't paying their own taxes, they don't have a taxpayer ID and are instead identified by their relation to a taxpayer with an ID. The name field in the dependent table is a partial key. It distinguishes dependents from one another, allowing a taxpayer to claim more than one dependent. For example, "John," a taxpayer, could claim

two dependents, “Bob” and “Betty,” who are distinguished from each other by their names.

Deleting from a database

There are two different ways to delete entries from a database. The first of which is to replace the data in those entries with NULL, essentially removing the data from the database. This is called a hard delete. The alternate option is to use another field to mark a record or entry as “deleted” while still retaining the information; or remove the information to a separate “deleted” table. When the database is read, “deleted” entries can be excluded from retrieval by a properly written query. This is called a soft or logical delete.

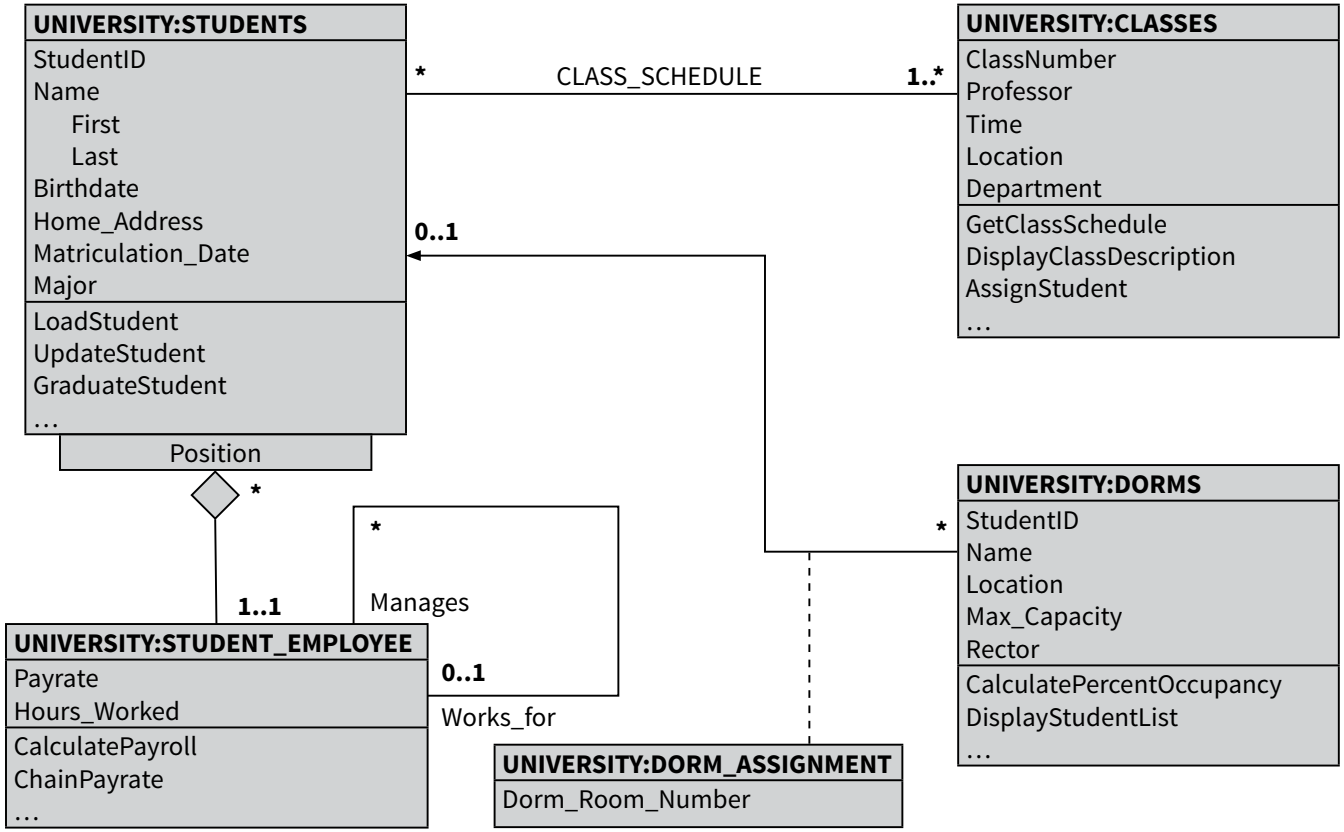
Both approaches have serious advantages and disadvantages. A soft delete allows for the fixing of mistakes and keeps the “deleted” data available for audit or troubleshooting purposes. It also permits other records related to the removed one to continue functioning as normal, as the necessary data is still present in the database. In a hard delete, any records which relate to a removed record will have to be altered to remove the relation; with now-incomplete weak entity types being deleted as well. This is called a cascading delete. Hard deletes, however, are more compatible with the right to be forgotten, since no information is retained. Additionally, hard deletes are less prone to error, since the deleted information is gone and cannot be accidentally mixed with not-“deleted” information by a mistake in a query.

File deletion in most operating systems is soft. For example, when a file is “deleted” in Windows, it is just moved from its previous folder to a special folder called the recycle bin. Even though the file doesn’t appear in its original location, that location is still retained by the operating system in its database of files and the file can still be retrieved by the operating system. The file is only hard deleted when the recycle bin is emptied. When this happens, the space allocated to the file on the hard disk is made available to other files and will eventually be overwritten. After this point, the un-overwritten parts of the file can only be retrieved by special forensic programs.

Mapping databases

The structure of databases can be depicted using a wide variety of notations and diagrams. The most commonly used in industry is UML or “Unified Modeling Language.” It is a single, unified format used to represent everything from databases to class structures in programs to hardware interactions and is managed by the International Organization for Standardization. When depicting a database, UML does not usually identify which fields are part of the primary key. UML database diagrams often simultaneously depict a class structure (used in programming) that sits on top of the database and interacts with it. On the next page is one example.

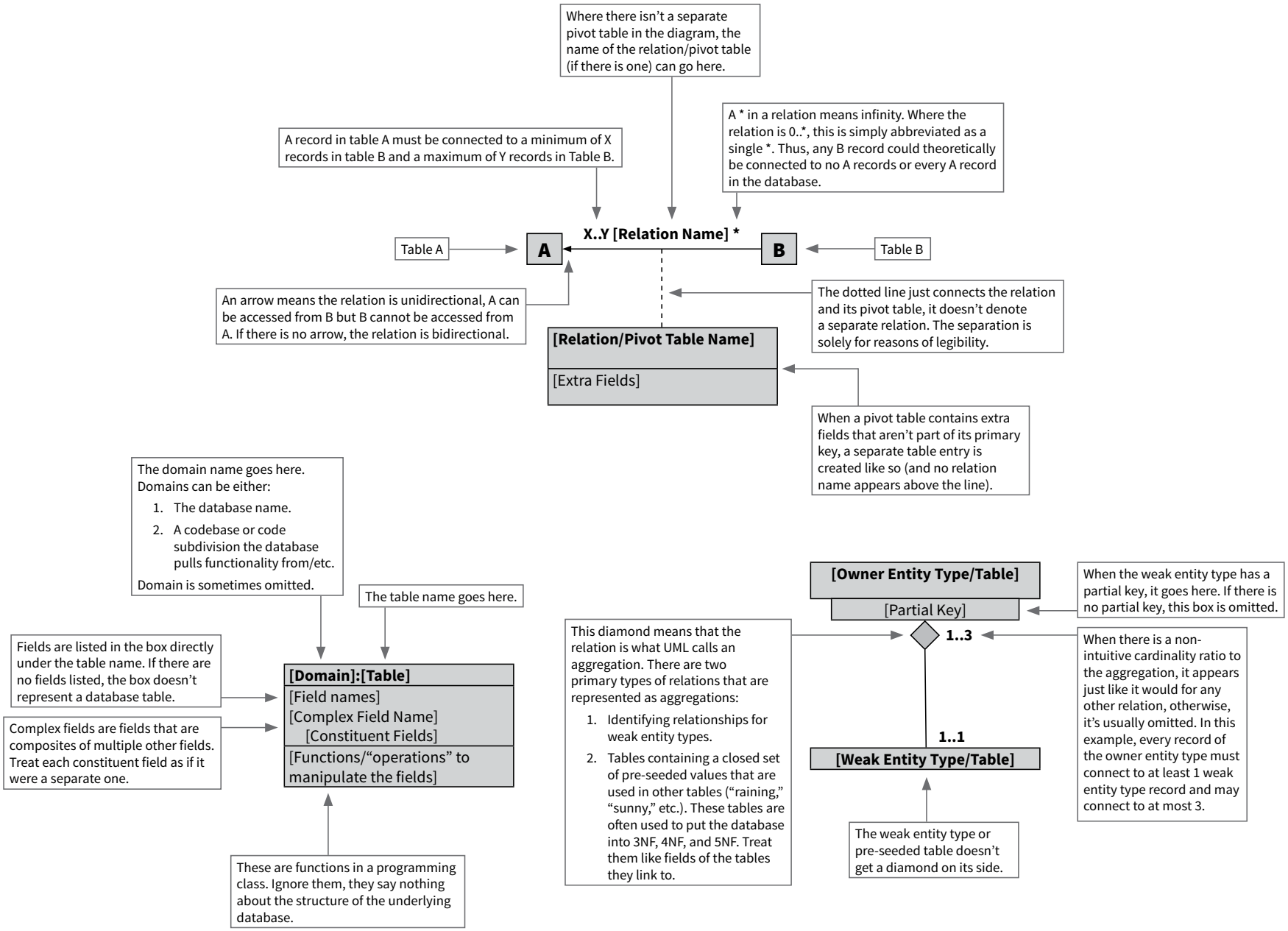
This diagram shows a database called “UNIVERSITY” with four major tables and several pivot tables. The first table, STUDENTS, stores each student’s basic information. The STUDENTS table is the owner entity type of the STUDENT_EMPLOYEE table, which contains some extra information about a student’s job with the university in a 1:N relationship; a student may have more than one job or no job. There is an unnamed pivot table connecting the two and the position field of STUDENT_EMPLOYEE differentiates a student’s different jobs from each other. Also included in that table



is the 1:N relationship between supervised and supervisor students, with another pivot table. The third table, CLASSES, relates to students on a N:M relationship, as multiple students can belong to multiple classes or none.

Every class must have at least one student and a user can retrieve a student's class schedule from the STUDENTS table or get a class roster of students from the CLASSES table. The table is linked to the STUDENTS table by way of a pivot table called CLASS_SCHEDULE. The STUDENTS table relates to the last major table, DORMS, on an N:1 relationship. The database has been set up so that a user can retrieve the list of students in a dorm via the DORMS table, but can't find out what dorm a student lives in via the STUDENTS table. The pivot table between the two tables, DORM_ASSIGNMENT, also records a student's dorm room number. A student may only be assigned to one dorm but a dorm can have theoretically infinite students.

On the next page are explanations of the three structures most common to UML depictions of databases and necessary to fully interpret the above chart:


Where there isn't a separate pivot table in the diagram, the name of the relation/pivot table (if there is one) can go here.
A record in table A must be connected to a minimum of X records in table B and a maximum of Y records in Table B.
A * in a relation means infinity. Where the relation is 0..*, this is simply abbreviated as a single *. Thus, any B record could theoretically be connected to no A records or every A record in the database.
Table A
A
X..Y [Relation Name] *
B
Table B
An arrow means the relation is unidirectional, A can be accessed from B but B cannot be accessed from A. If there is no arrow, the relation is bidirectional.
The dotted line just connects the relation and its pivot table, it doesn't denote a separate relation. The separation is solely for reasons of legibility.
[Relation/Pivot Table Name]
[Extra Fields]
When a pivot table contains extra fields that aren't part of its primary key, a separate table entry is created like so (and no relation name appears above the line).
The domain name goes here. Domains can be either:
1. The database name.
2. A codebase or code subdivision the database pulls functionality from/etc.
Domain is sometimes omitted.
The table name goes here.
Fields are listed in the box directly under the table name. If there are no fields listed, the box doesn't represent a database table.
Complex fields are fields that are composites of multiple other fields. Treat each constituent field as if it were a separate one.
[Domain]:[Table]
[Field names]
[Complex Field Name]
[Constituent Fields]
[Functions/"operations" to manipulate the fields]
These are functions in a programming class. Ignore them, they say nothing about the structure of the underlying database.
[Owner Entity Type/Table]
[Partial Key]
When the weak entity type has a partial key, it goes here. If there is no partial key, this box is omitted.
This diamond means that the relation is what UML calls an aggregation. There are two primary types of relations that are represented as aggregations:
1. Identifying relationships for weak entity types.
2. Tables containing a closed set of pre-seeded values that are used in other tables ("raining," "sunny," etc.). These tables are often used to put the database into 3NF, 4NF, and 5NF. Treat them like fields of the tables they link to.
1..3
When there is a non-intuitive cardinality ratio to the aggregation, it appears just like it would for any other relation, otherwise, it's usually omitted. In this example, every record of the owner entity type must connect to at least 1 weak entity type record and may connect to at most 3.
1..1
[Weak Entity Type/Table]
The weak entity type or pre-seeded table doesn't get a diamond on its side.

Defeating enemy #1: Inconsistency

The primary enemy of every database is inconsistency, or the presence of contradictory information within the same database. Inconsistency is also a primary risk factor for liability under the EU's General Data Protection Regulation, Chapter III, which grants rights, including rectification and erasure, to data subjects. Human error and incomplete database updates, which can occur due to many factors, including poor coding or a bad connection, are the primary generators of database inconsistencies and will happen to some extent in every database.

The best way to limit the risk of database inconsistencies is to use databases that utilize the normal forms of database design to limit redundancy. to illustrate, let's fix the simple, already inconsistent, one-table database of American sports teams at the bottom of the page. Using what you've learned about databases so far, see if you can spot all four potential sources of inconsistency.

Here are all four:

1. The Giants are recorded as having a winning record, even though they won less than a fifth of their games. This direct inconsistency between fields in the same record can occur because the Winning_Record field duplicates information already available in the Wins and Losses fields.

2. Baseball and football are fundamentally different sports. Baseball teams don't score touchdowns and football teams don't score home runs. However, the SF Giants, a baseball team, are recorded as having scored two touchdowns. This is another inconsistency between fields, this time caused by the mixture of records using different fields into the same table.

3. Someone made a typo: Boston is both incorrectly placed in Mississippi (MS) and correctly placed in Massachusetts (MA). If you query the database to find out what state Boston is in, you will get two answers and if you search for all Massachusetts teams, the Patriots won't appear. This is an example of inconsistency between records.

Table: TEAMS

Name	City	State	Sport	Wins	Losses	Winning_Record	Touchdowns	Home Runs
Cardinals	St. Louis	MO	Baseball	22	16	Yes	NULL	196
Giants	New York	NY	Football	3	13	Yes	28	NULL
Cardinals	Phoenix	AZ	Football	8	8	No	29	NULL
Red Sox	Boston	MA	Baseball	28	13	Yes	NULL	168
Giants	San Francisco	CA	Baseball	21	21	No	2	128
Patriots	Boston	MS	Football	13	3	Yes	49	NULL

4. There is no unique field that makes sense as a primary key. We have teams with the same name, teams from the same cities, and teams in the same sports. Our only option to use this database in its current state is to designate a composite key of multiple fields (like Name and City), but this may not be sustainable in the long-term. If we are expecting to add university and high school teams to our database, it is conceivable that we will have records identical in every field left of wins and thus appearing identical to the system. This could (if our database program is working properly) cause the exclusion of certain teams or (if it isn't) create inconsistent, apparently duplicate records. Don't feel bad if you didn't see this one, primary key issues can be difficult to spot if you aren't looking for them.

If we put the database into the First Normal Form (abbreviated: 1NF), we will eliminate three of our four possible inconsistencies. To put a database into the first normal form, we must:

1. Group like records together and different records separately. In this case, we have two very different types of records in the same table, baseball teams and football teams; these two classes each use unique fields, a prime indicator of difference. We should divide our TEAMS table into two different tables by sport and distribute the records and fields accordingly.

2. Eliminate redundant fields. We have two redundant fields in this database. Since our tables are now divided by sport, we don't need the Sport field anymore. The information it contains is entirely given by the record's presence in a specific table. Additionally, Winning_Record contains information that can be easily derived by comparing the Wins and Losses fields against each other. Both fields should be eliminated.

3. Give or designate a unique primary key for every record. As mentioned before, no existing field has a value which is guaranteed to be unique. Our best option is to use a GUID, in this case however, we will use simple numbers for legibility.

Here's how our database looks after these transformations:

Table: BASEBALL_TEAMS

GUID (PK)	Name	City	State	Wins	Losses	Home Runs
1	Cardinals	St. Louis	MO	22	16	196
2	Red Sox	Boston	MA	28	13	168
3	Giants	San Francisco	CA	21	21	128

Table: FOOTBALL_TEAMS

GUID (PK)	Name	City	State	Wins	Losses	Touchdowns
1	Giants	New York	NY	3	13	28
2	Cardinals	Phoenix	AZ	8	8	29
3	Patriots	Boston	MS	13	3	49

You will notice that the first two inconsistencies are eliminated. It is no longer possible to assign touchdowns to a baseball team or home runs to a football team, or to designate a team with more losses than wins as having a winning record. Winning records can be determined programmatically and doesn't need to be stored directly in the database. Similarly, when using GUIDs as a primary key, we can now add seemingly duplicate records; like a Boston, MA, children's baseball team named the Red Sox. Note that in a professional database, such an addition would be prohibited or more information would be collected (like level of play) to preserve meaningful distinctions for the user.

To eliminate our last inconsistency, the location of Boston, we will need to put the database into the Second Normal Form (abbreviated: 2NF). To do this, we must identify sets of values that apply to multiple records and create a separate table to hold them, linking back to the original records using a foreign key. In our table, the values in the City and State fields apply to every other record and can be common to multiple records (i.e. the same city can have multiple sports team). Therefore, we should move the City and State fields to their own table, LOCATION, and link it into the other tables via a foreign key called LOCATION_GUID. Here is how our database looks now:

Table: BASEBALL_TEAMS

GUID (PK)	**Name**	**Location_GUID**	**Wins**	**Losses**	**Home Runs**
1	Cardinals	1	22	16	196
2	Red Sox	2	28	13	168
3	Giants	3	21	21	128

Table: FOOTBALL_TEAMS

GUID (PK)	**Name**	**Location_GUID**	**Wins**	**Losses**	**Touchdowns**
1	Giants	4	3	13	28
2	Cardinals	5	8	8	29
3	Patriots	2	13	3	49

Table: LOCATION

GUID (PK)	**City**	**State**
1	St. Louis	MO
2	Boston	MA
3	San Francisco	CA
4	New York	NY
5	Phoenix	AZ

The last inconsistency is now eliminated and Boston is only listed once, in Massachusetts. If there is another location typo, or Mississippi made it into LOCATION instead of Massachusetts, the problem can be fixed by updating only one entry, rather than having to query through the entire database to fix multiple entries with the same mistake. This will greatly reduce the risk of inconsistency between rows.

The Third Normal Form (3NF), Fourth Normal Form (4NF), and Fifth Normal Form (5NF) are more complicated. They can generally be summarized thusly: When creating a table that records nominal values that can repeat, create a separate table to hold the nominal values, or possible combinations of nominal values, related to that occurrence and link to that table using a foreign key rather than insert the nominal values directly in the table. For example, a table in the second normal form that lists the winners of the Tour de France by year needs to worry about the Third, Fourth, and Fifth normal forms if it contains superfluous information like the winner's date of birth or country, but a table of Tour de France competitors would not, since there would only be one entry per competitor, regardless of how often they won.

Understanding and evaluating a database

When confronted with a new database, it is important to grasp the structure of the database before understanding any of its individual contents. Therefore, the best thing to begin with is to ask for a UML Diagram. A UML diagram can always be created. If a database is extremely complicated, the UML diagram can be broken up into several diagrams of specific relations. Next, you want to look at the structure of the database and see if it allows for inconsistency. Does it follow the normal forms? Pay special attention to the primary keys of each table and consider the population the database is intended to catalog: Is it conceivable that there will be other potential records that have the same primary key? After this, review the data itself. It is possible to write SQL statements that output the data in different ways, including with data from different tables mapped to each other. Just because data is stored in a specific way doesn't mean it must be reported in that way.

Test your knowledge

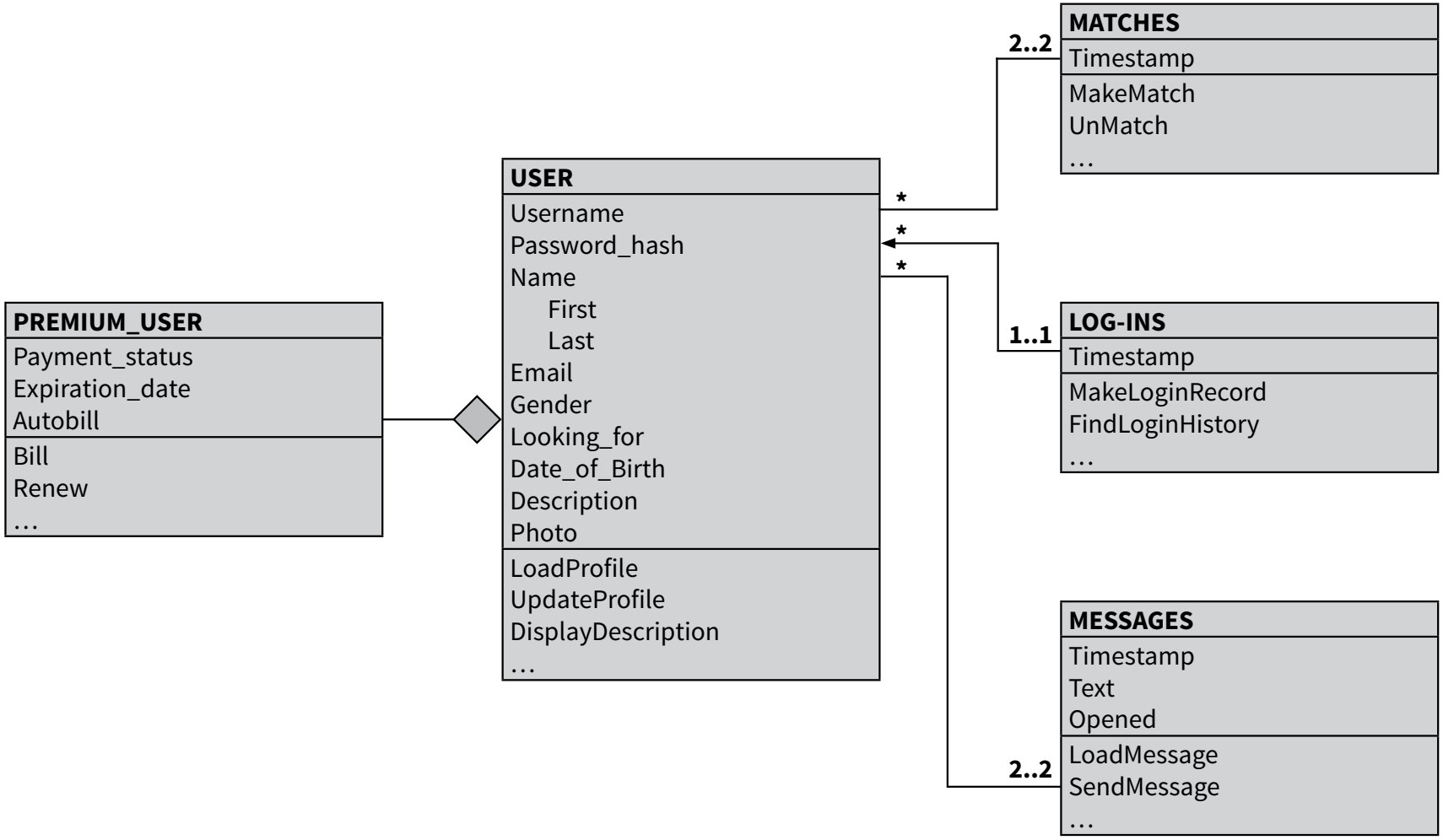
You are the data privacy officer for MatchmakerMatchmaker.com, a dating website for matchmakers, and are regularly consulted on database changes as part of the company's privacy impact assessment process. A UML diagram showing the current structure of the site's database is s on the following page. Applying what you've learned, should you approve each proposed change from a consistency perspective? Why?

1. Adding a phone number field to USER, to be used for two-factor authentication around password recovery. The recorded phone number will be expected to be unique for each user.

2. Adding an age field in years to USER, because it will make dating profiles load faster by reducing the amount of calculations required.

3. Adding an email field to LOG-INS, which records the user's email address when they log in (log-ins to MatchmakerMatchmaker use email address not username).

4. Adding a Boolean to the USER table to record when a user has unsubscribed from email notifications.


MATCHES
Timestamp
MakeMatch
UnMatch
…
2..2
USER
Username
Password_hash
Name
First
Last
Email
Gender
Looking_for
Date_of_Birth
Description
Photo
LoadProfile
UpdateProfile
DisplayDescription
…
*
*
*
PREMIUM_USER
Payment_status
Expiration_date
Autobill
Bill
Renew
…
LOG-INS
Timestamp
MakeLoginRecord
FindLoginHistory
…
1..1
MESSAGES
Timestamp
Text
Opened
LoadMessage
SendMessage
…
2..2

Answers:

1. Yes. Phone number introduces information that is not contained anywhere else in the database. Because phone number is expected to be unique, it will become a candidate key for the USER table.

2. No. The age field breaks the first normal form because it is redundant with Date_of_Birth. The normal forms are good rules of thumb, not requirements, and should be departed from if a good case can be made. Although introducing a redundant field into a table to decrease loading times when it contains information that is time-consuming to derive from the other data (like position in a long, sorted list) is one such circumstance, it is not advisable in this case. Because of how dates are stored in most databases, age can be derived from a date of birth in milliseconds; most languages would only require one line of code for the task.

3. No. This breaks one of the advanced normal forms, the third to be precise. Because a user may change their email address, the entry in USER's email can become inconsistent with that in LOG-INS's email. Because the Username field (part of the primary key) is already recorded, you have a foreign key to the USER table, where you can get the same information. If someone down the road uses the LOG-INS table to make a mailing list, there's a risk that MatchmakerMatchmaker will violate GDPR, especially if the user's data has been purged from your database but the raw log-in data has been preserved. Additionally, if MatchmakerMatchmaker isn't purging email addresses in the LOG-INS table when a user deletes their account (which would be computationally expensive), it may violate the right to be forgotten.

4. No. This is an inappropriate and legally risky implementation of soft delete. Remember that whether an entity is NULL also conveys information. A true value in the proposed field is inconsistent with the implication of a populated email field entry in the same table. This proposal is a very tempting "easy fix" for GDPR's unsubscribe requirements that, if used, creates a huge danger of future liability. If an employee assembles a mailing list from the USER table and makes the easy mistake of forgetting to exclude records where the unsubscribe boolean is true, MatchmakerMatchmaker will violate GDPR. Better implementations of unsubscribe include hard deleting the unsubscribed user from the database. If a soft delete is necessary it could be better implemented by moving unsubscribed users or their email information to a different, clearly-labeled table (like UNSUBSCRIBED). An important part of software development is considering how future users and administrators could use and maintain a system without the guidance of the original developers.