Database design theory, Part I

Functional dependencies

Introduction
As we saw in the last segment, designing a good database is a non-trivial matter. The E/R model gives a useful “rapid prototyping” tool, but provides little guidance on “the best way” of doing things; we developed some best practices, based on observed bad behaviors we want to avoid, but it was an ad-hoc exercise. This next segment introduces a formal theory of database design that codifies best practices and allows us to reliably design database schemas that exhibit certain desirable properties.

The overall goal of database design theory is to capture as much of our model’s structure as possible—particularly constraints—in the database schema itself. Doing so allows the database engine to enforce those constraints automatically and simplifies the application logic built on top of it. A “normalized” database schema has two main benefits:

1. Minimal redundancy. We already touched on redundancy in the E/R model, but database design theory gives a formal way to identify and eliminate data redundancy in a database schema.¹

2. Constraint capture. Certain types of constraints can be expressed implicitly by the structure of a relational model, and we will exploit this to relieve the application of enforcing them.

As a very simple example, consider the following relation:

<table>
<thead>
<tr>
<th>Student Name</th>
<th>Student Email</th>
<th>Course</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xiao</td>
<td>xiao@gmail</td>
<td>CSCC43</td>
<td>Johnson</td>
</tr>
<tr>
<td>Xiao</td>
<td>xiao@gmail</td>
<td>CSCD08</td>
<td>Bretscher</td>
</tr>
<tr>
<td>Jaspreet</td>
<td>jaspreet@utsc</td>
<td>CSCC43</td>
<td>Johnson</td>
</tr>
</tbody>
</table>

The relation contains significant redundancy; a student might take multiple courses, an instructor might teach multiple courses, etc. This redundancy is undesirable because it allows certain “anomalies” that place additional burden on the database application:

- **Update anomaly:** if Xiao’s email address changes, the various copies of Xiao’s record in the system could get out of sync unless the application remembers to change all of them.
- **Deletion anomaly:** what if we delete the last student taking a particular course? How do we continue to represent the course in the system?
- **Insertion anomaly:** similar to deletion, how can we insert a new student into the system without forcing him or her to already be taking at least one course?

¹ As before, the system cannot provide much help in eliminating structural redundancy, as doing so requires domain knowledge.
The solution to the above dilemma is to split the relation into several, smaller relations. In particular, student information and instructor information should probably be stored separately. Once we start splitting relations, we start to encounter the matter of constraints: whether a student has one or many email addresses, whether an instructor can teach multiple courses in a term, etc. We will explore both redundancy and constraints in more detail below, after some definitions.

Functional dependencies
Database design theory centers on the concept of a functional dependency. Functional dependencies (FD’s for short) are a generalization of the key concept we’ve used throughout this course. As part of the formal definition of an FD, we will also formalize various types of keys that can arise during database design.

FDs as assertions
Each FD is of the form X -> Y, where X and Y are subsets of attributes from some relation R. X-> Y is an assertion about values in R: any and all tuples taking the same set of values in X must also take the same set of values in Y. In other words, the values of attributes in Y somehow depend on the values that attributes in X take. We might also express this concept as a function, with X as its argument(s) and Y as return value(s): \( f(x_1, \ldots, x_n) \rightarrow y_1, \ldots, y_m \). Although the function notation is convenient, it has one flaw: \( f \) may or may not be computable. Consider a cryptographic hash, for example: with extremely high probability, a given hash value uniquely identifies the input data the hash was computed from. However, hashes—and cryptographic hashes in particular—are carefully designed to make it impossible to extract the original text from a hash value. In other words, we could state the FD hash -> plaintext as well as plaintext -> hash, but only one of those functional dependencies has a computable “function.”

When working with functional dependencies, we use upper-case letters to represent sets of attributes: X -> Y, and lower-case letters to represent individual attributes: \{x,y,z\} -> \{a,b,c\}. As a notational convenience, we usually drop the braces and commas when referring to sets of attributes whose names are all a single character: xyz -> abc.

FDs as keys
If you recall our working definition of a key (set of attributes that uniquely identify a tuple), it should be clear that a key is a kind of functional dependency: \( K \rightarrow R \) (where K is the set of key attributes and R is the set of all attributes in the relation). Note, however, that a key is not the only possible type of FD. You could have a “partial key” that identifies only some of the attributes in a tuple. In our example from the first section, consider the relation containing student and course information: we can identify the FD email -> name (given an email address, we can identify the student it belongs to), but that FD is certainly not a key, because it does not tell us what courses a student has taken or who taught those courses. In particular, a student could take multiple courses, from multiple instructors, so it’s impossible to infer a unique course or instructor from a single student email address.
Even once we have a set of attributes that uniquely identifies our tuple, the question remains whether that key is the only one possible for the relation. Suppose we had a more traditional student relation, which contained student UTorID, name, email address, and major. The attribute pair {UTorID, name} certainly qualifies as a key (it uniquely identifies every student in the database), but it is not the simplest key possible: UTorID is a key by itself, so it doesn’t matter whether we include name in the key or not. To distinguish these two cases, we define a “superkey” as any attribute set that uniquely identifies a tuple. In other words, what we have called a “key” so far is really a superkey. It’s easy to make more superkeys, by adding more attributes from the relation. If UTorID is a superkey, then so are {UTorID, name}, {UTorID, major}, and any other superset of {UTorID}. A relation can thus have a large number of superkeys: a relation has a combinatorial number of attribute subsets, and any of them that contains a superkey is also a superkey. To distinguish “true” keys (such as UTorID) from less interesting superkeys (such as {UTorID, name} and {UTorID, major}), we define a “candidate key” as any superkey that contains no superkeys in its proper subsets. Thus, {UTorID, major} is not a candidate key because one of its proper subsets (namely {UTorID}) is also a superkey. We can’t simplify {UTorID} further (its only proper subset is the empty set), so it must be a candidate key.

We call these minimal keys “candidates” because in general a relation can contain two or more keys that fit the definition of a candidate key. In our student relation, for example, {UTorID} is one candidate key, but so is {email}: an email address uniquely identifies a student, making it a superkey, and the superkey cannot be simplified further, making it a candidate key. In the end, we must select a single candidate key as our “primary key.” The database engine will use the primary key to identify tuples from the relation, and which candidate key we choose to use for this purpose is a design decision. In case of {UTorID} vs. {email}, we would probably choose {UTorID}, because the latter is created specifically to be a unique identifier and students cannot change it the way they might change an email address.

Finally, we give a special name to attributes that are part of a candidate key: “prime attributes.” If you remove a prime attribute from a superkey, the resulting attribute set is no longer a superkey; removing non-prime attributes from a superkey moves it closer to candidate key status. A candidate key is thus any superkey that contains only prime attributes. Prime attributes will become important later, when we start manipulating relations.

**FDs as domain knowledge**

Every year, students misunderstand an extremely important aspect of functional dependencies:

*Functional dependencies come from the domain.*

_Database engines cannot infer them from the available data._

In other words, functional dependencies are something the database designer knows (or assumes, or wants to enforce artificially) about the domain the data will come from. The database engine will enforce any and all functional dependencies it is made aware of (whether they make sense or not) and

---

2 A superkey is to a candidate key as a superset is to a set.

3 Recalling our discussion of keys from the E/R segment, we prefer immutable integer keys when possible.
cannot safely enforce any functional dependency (even the most obvious ones from a human’s perspective) without being told explicitly to do so. Consider the following relation:

<table>
<thead>
<tr>
<th>ID</th>
<th>Email</th>
<th>City</th>
<th>Country</th>
<th>Surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1983</td>
<td><a href="mailto:tom@gmail.com">tom@gmail.com</a></td>
<td>Toronto</td>
<td>Canada</td>
<td>Fairgrieve</td>
</tr>
<tr>
<td>8624</td>
<td><a href="mailto:mar@bell.com">mar@bell.com</a></td>
<td>London</td>
<td>Canada</td>
<td>Samways</td>
</tr>
<tr>
<td>9141</td>
<td><a href="mailto:scotty@gmail.com">scotty@gmail.com</a></td>
<td>Winnipeg</td>
<td>Canada</td>
<td>Samways</td>
</tr>
<tr>
<td>1204</td>
<td><a href="mailto:birds@gmail.com">birds@gmail.com</a></td>
<td>Aachen</td>
<td>Germany</td>
<td>Lakemeyer</td>
</tr>
</tbody>
</table>

Suppose we tried to mechanically infer functional dependencies from the above dataset. By looking at the (combinations of) values that uniquely identify rows in the above relation, a computer might conclude that the following FDs exist:

- ID, email, and city are all candidate keys
- surname -> country

If we were to add an additional tuple to the relation, however, some of those FDs could be invalidated. The arrival of (1538, happypiranha@rogers.co.uk, London, UK, Fairgrieve) would prove city is not a key and that country->surname does not hold. On the other hand, the arrival of (1983, tom@gmail.com, Toroto, Canada, Fairgrieve) would also seem to invalidate several FDs (including ID and email as keys). With our domain knowledge, such as the nature of email addresses and the types of cities in Canada, we humans can quickly identify that the first insertion is valid, and that the second is most likely a typo that should be rejected.

**A geometric view of FDs**

As an alternative way to visualize functional dependencies, consider the following Venn diagram:

The data domain is $x,y,z$, all possible integer coordinates in three dimensional space. Within this domain, we can identify any number of subdomains that constrain the values that $x,y$ and $z$ may take, and these can be grouped into families by the functional dependency they imply. Two important observations can be made:

---

4 London, UK is a valid location and it is unreasonable to assume that you can infer countries from surnames.
5 ID should be unique, and there is no city named “Toroto” in Canada (but there is a “Toronto”)
1. Subdomains can overlap. Some values of the domain (such as 1,1,2) are valid members of both subdomains defined in the figure. Others (such as 1,1,-2) fit only one of the subdomains, while still others (such as 1,-1,-2) belong to neither of the subdomains. Sometimes the overlap is complete, in that one functional dependency completely contains another, more restricted one.
2. More than one relationship can be captured by a given FD. For example, the figure gives specific examples of functions that follow z -> xy and xy -> z, but others are certainly possible. The functions $z = x \cdot y$, $z = y - x$, and $z = y \cdot \exp(x)$ all fall under the FD $z \rightarrow xy$, for example.

Note that many of the above functions are not invertible: we cannot tell from a given $z$ what $x$ and $y$ are, even when $xy \rightarrow z$. Similarly, $z \rightarrow xy$ does not make it easy to infer $z$ given $x$ and $y$ (we don’t know if $z = x$ or $z = -x$, in the example from the figure).

**Manipulating functional dependencies**

Functional dependencies have several properties that will prove useful as we manipulate database designs. We will cover these next. Note that, when discussing FDs, we often refer to their left- and right-hand sides as LHS and RHS, respectively.

**Trivial FDs**

First of all, all FDs should be simplified to remove redundancy. The trivial FD $x \rightarrow x$ is true for all $x$, but it is completely uninteresting. By the same reasoning, we can always remove from the RHS of an FD any attributes appearing in its LHS:

- $xy \rightarrow xab$ becomes $xy \rightarrow ab$
- $abcd \rightarrow abcdefg$ becomes $abcd \rightarrow efg$
- etc.

While it may seem obvious that such FDs are uninteresting, they are easy to generate as we manipulate FDs mechanically, and so the redundancy must be removed to “clean up” the results of schema manipulation.

**Splitting FDs**

Any FD can be simplified so that its RHS contains only a single column, by replicating the FD once for each attribute from the RHS. For example, the FD on the left is equivalent to the three on the right:

- $xyz \rightarrow a$
- $xyz \rightarrow abc$
- $xyz \rightarrow b$
- $xyz \rightarrow c$

Note that the converse is NOT true: you cannot “simplify” an FD by splitting its LHS! If we stretch a little, we might claim that the FD $\{\text{name, surname}\} \rightarrow \text{phone}$ in a phone book, but clearly neither name -> phone nor surname -> phone is a valid FD.
Combining FDs

Just as FDs can be split, they can also be combined, which can be especially convenient when we want to identify candidate keys. It’s easy to see how the three FDs from the previous example can be converted back into a single FD with larger RHS, because all have the same LHS. Further, “similar” LHS can be combined as well, when desired. For example:

\[xy \rightarrow ab \quad \text{and} \quad yz \rightarrow cd\]

can be combined into

\[xyz \rightarrow abcd\]

We simply take the union of the FDs: both original LHS are still present on the new LHS, and both original RHS are also present on the new RHS. Note that combining FDs can result in redundancy. For example:

\[xy \rightarrow ab \quad \text{and} \quad bc \rightarrow nm\]

can be combined into

\[xybc \rightarrow abnm\]

which in turn simplifies to

\[xybc \rightarrow anm\]

Further, note that splitting the above FD can introduce additional redundancy, as inferred from other FDs we know. For example:

\[xybc \rightarrow a \quad \text{and} \quad xybc \rightarrow nm\]

are trivial FDs because we already have

\[xy \rightarrow a \quad \text{and} \quad bc \rightarrow nm\]

Transitive property of FDs

Given two FDs \(x \rightarrow y\) and \(y \rightarrow z\), we can safely infer that \(x \rightarrow z\). Such chains of dependencies can be arbitrarily long, and the path to transitive dependence can be branchy. For example, given the following FDs:

\[x \rightarrow y \quad y \rightarrow z \quad z \rightarrow a \quad x \rightarrow w \quad wy \rightarrow b \quad ab \rightarrow c\]

We can infer \(x \rightarrow c\). Note that transitivity can form cycles: \(x \rightarrow y \quad y \rightarrow z \quad z \rightarrow x\) is not just allowed, but can arise in real life. In fact, cycles are one of the main reasons we care about transitivity, because it impacts key selection and other design decisions. For the previous example, \(x, y,\) and \(z\) are all candidate keys by transitivity. Even though none of them is directly identified as a key by the FD set we started with, we can easily show that \(x \rightarrow yz \quad y \rightarrow xz\) and \(z \rightarrow xy\) all hold.

Given an attribute set \(A\) and FD set \(F\), we can compute the transitive closure of \(F\) over \(A\) (denoted by \(A^+\)) as follows:

\[A^+ = A, \quad F' = F\]

\[\text{While } \exists X \in F' \text{ s.t. LHS}(X) \subseteq A^+:\]

\[A^+ = A^+ \cup \text{RHS}(X)\]

\[F' = F' - X\]

\[\text{At end: } A \rightarrow B \quad \forall B \in A^+\]
**Transitive closure of attribute sets**

Just as we can compute the transitive closure of an FD set, we can also compute the closure, $A^+$, of an attribute set. In this case, $A^+$ is the set of attributes entailed by $A$. In other words $A^+$ is the answer to the following question: “given $A$, what else do we know about the tuple?” We can compute $A^+$ by starting with $A$, then repeatedly adding the RHS of any FDs whose LHS is already in $A^+$. The newly-added attributes may in turn allow us to add yet more RHS of other FDs, and the process repeats until no attributes can be added. For example, given $R(a,b,c,d,e,f)$ and FDs $ab->c$, $ac->d$, $c->e$, and $ade->f$, we can compute $\{a,b\}^+$ as follows:

![Diagram showing the computation of the transitive closure for $\{a,b\}$]

In other words, $ab$ is a superkey, because it entails every other attribute in $R$. Further, it is a candidate key because removing either $a$ or $b$ from the set leaves some attributes unaccounted for.

Several transformations of database schemas start with the transitive closure of either FD sets or attribute sets.

**Minimal basis of an FD set**

A final transformation involves simplifying sets of FDs: the previous rules have shown that, given a set of FDs, we can manufacture a near-arbitrary number of additional FDs. Further, we have seen that the FDs given may not make relevant information obvious, requiring use of transitivity and closure to expose the information we need. Given some arbitrary FD set $F$, we usually desire to work with a “minimal basis” of FD—the subset $F'$ of $F$ for which all other FDs in $F$ can be derived, and from which removing an FD would cause a loss of information. Constructing a minimal basis is straightforward (see pseudocode below), but has two main problems:

1. The basis may not be unique. Many FD sets allow multiple minimal bases, and which one we end up with depends on the order in which we examine FDs from $F$ during the simplification process. Further, it turns out that the quality of our final database schema can depend on which minimal basis we use. This is unfortunate but unavoidable.

2. Computing a minimal basis is time-consuming, with an exponential search space that is amenable to dynamic programming approaches.\footnote{This makes computation of minimal bases a popular source of homework and exam questions of arbitrary “difficulty” (where “difficult” means “time consuming”). I will test you on your ability to manipulate minimal bases, but am not interested in wasting your time with significant busywork.}

The method for computing a minimal basis is as follows:

1. Split all RHS into singletons

2. $\forall X \in F'$, test whether $J = (F'-X)^+$ is still equivalent to $F^+$

$\Rightarrow$ Might make $F'$ too small (inaccurate), if so, undo.
3. \( \forall i \in \text{LHS}(X) \ \forall X \in F', \text{ let } \text{LHS}(X')=\text{LHS}(X)-i \)
   Test whether \((F'-X+X')^+\) is still equivalent to \(F'\)

=> Might make \(F'\) too big (redundant), if so, undo.

4. Repeat (2) and (3) until neither makes progress

To give a concrete example, suppose we had \(R(a, b, c, d)\) and \(F = \{a->ac, b->abc, d->abc\}\). The minimal basis of \(F\) might be computed as follows:

1\(^{st}\) Step
   \(H = \{A->A, A->C, B->A, B->B, B->C, D->A, D->B, D->C\}\)

2\(^{nd}\) Step
   – \(A->A, B->B\): can be removed as trivial
   – \(A->C\): can’t be removed, as there is no other LHS with \(A\)
   – \(B->A\): can’t be removed, because for \(J=H-\{B->A\}\) is \(B^+=BC\)
   – \(B->C\): can be removed, because for \(J=H-\{B->C\}\) is \(B^+=ABC\)
   – \(D->A\): can be removed, because for \(J=H-\{D->A\}\) is \(D^+=DBA\)
   – \(D->B\): can’t be removed, because for \(J=H-\{D->B\}\) is \(D^+=DC\)
   – \(D->C\): can be removed, because for \(J=H-\{D->C\}\) is \(D^+=DBAC\)
   Step outcome => \(H = \{A->C, B->A, D->B\}\)

3\(^{rd}\) Step
   – \(H\) doesn’t change as all LHS in \(H\) are single attributes

4\(^{th}\) Step
   – \(H\) doesn’t change

Therefore, we conclude that the minimal basis is \(\{A->C, B->A, D->B\}\).