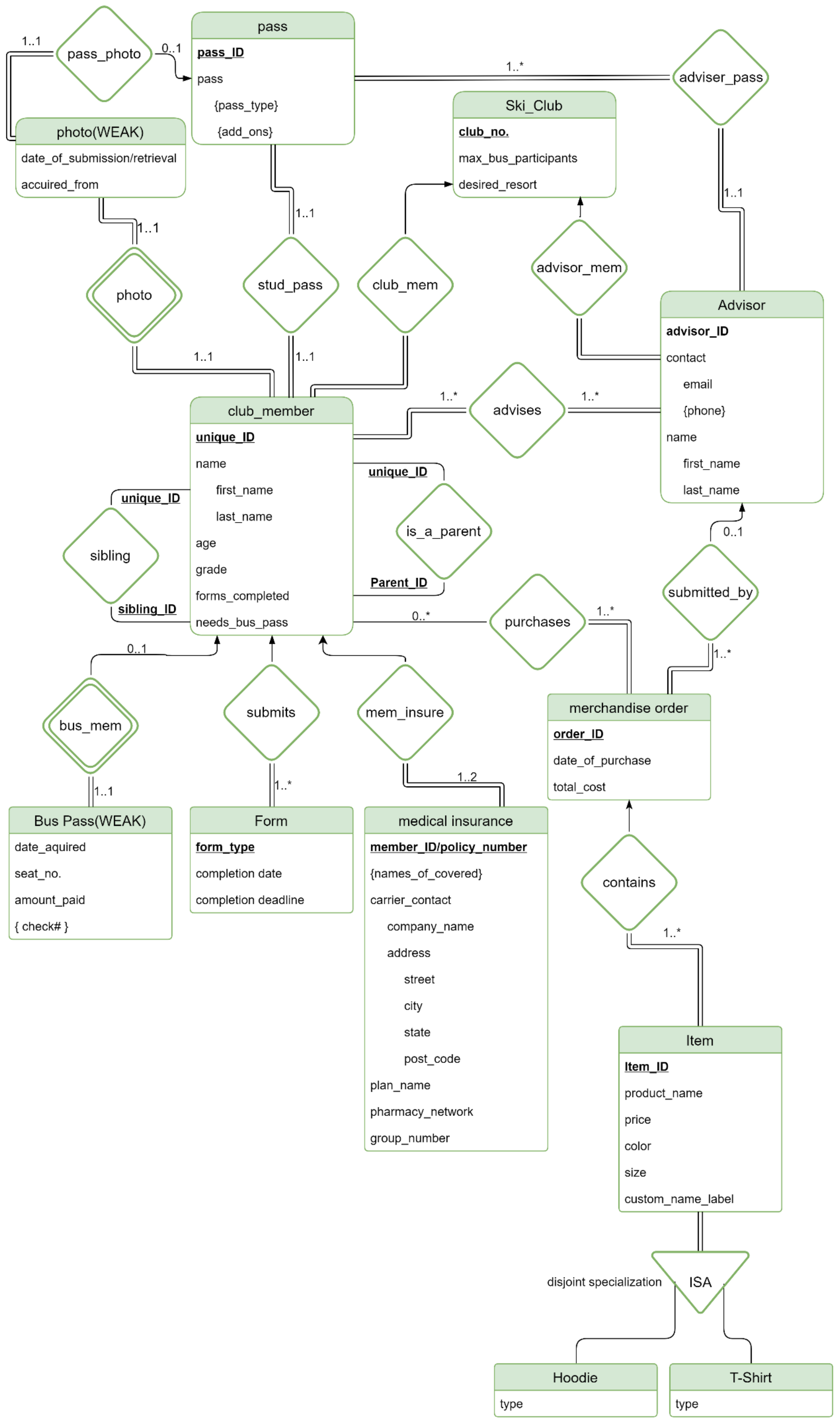
Landon Shenberger

The E-R diagram:



Explanation of the relationships and overall design:

NOTE: These design notes were included in the previous submission; however, I thought it would be important to include because the assignment specified “database in your word document write-up – Explain your choices.” In addition, during the normalization process, additional explanation of FDs, indexing (only on primary attributes), primary keys, foreign keys, referral integrity constraints and the theoretical basis for such choices is given. Finally, a complete relational schema is supplied. This document is very long, but I wanted to insure a correct design for easy implementation. The final schema at the bottom is the result of all this work.

Club_mem: connects the Ski_Club entity with the club_member entity. The ski club side denotes a one relationship, meaning there is one, not many ski clubs to many club members (hence the strait line with no arrow on the club member side). In addition, on the club member side, there is total participation, meaning every member must be involved in a ski club.

Advisor_mem: Describes a relationship similar to the club_mem relationship, except it connects an advisor with its corresponding club rather than a regular club member.

Advises: This relationship describes that an advisor must advise a club member. Both sides are total participation because a club_member requires at least one advisor and an advisor is required to advise at least one member, although they could both have more than one advisor/member.

Is_a_parent: This relationship would be used by a database to determine whether a member is a parent of a student in the club. It refers back to itself, that is, the Unique_ID of a club member is referred to by Parent_ID, denoting the parent of that Unique_ID.

Sibling: this relationship is alike the parent relationship. It refers to the club_member entity in the same manner that the parent relation refers back to club_member. It would be used to find a sibling of another for the purposes of determining the price of a bus pass.

Photo: a weak relationship connecting the club members submission of a photo to the photo submission itself. Conspicuously, the club member owns their photo, hence the weak relationship. Every photo needs a club member and every club member must have a photo on file and thus the relationship is total.

Pass_photo: Each photo is to be submitted to snow trails and used on the corresponding pass; therefore, the pass_photo relationship denotes how a photo relates to a pass (i.e. the photo is to be printed on the pass). As aforementioned, every student must have a photo, but advisor passes do not; thus, the photo has a total participation in the passes, and the passes have a partial participation because the advisors do not have or need photos. They have a max cardinality of one because no pass should have more than one photo.

Advisor_pass: Connects an advisor with their corresponding snow trails pass. The advisors are given passes with no photos, hence why not every pass has a photo on it (denoted by pass_photo). An advisor can hold more than one pass but must have at least one, hence the cardinality of "1..*" on the pass side. The advisor side has a cardinality of "1..1" because every advisor needs a pass, but there is not multiple advisors to one pass.

Bus_mem: This relationship is used to portray the fact that a club member can have a bus pass as it connects the club member entity with the bus pass entity. The bus_mem is a discriminator, meaning it is connected to a weak relationship, that is, the bus pass. This is conceptually consistent as a member would own their bus pass. The cardinality of "0..1" is present to delineate that a member does not need to participate in having a bus pass as they can drive up.

Submits: This relationship connects the member entity with the form entity. Theoretically, this makes sense because a form is submitted by a club member. It is a one-to-many total participation (on the form side) relationship: every form must be submitted by one member and there can be multiple forms per member submission.

Mem_insure: Describes the relationship between a member and their corresponding medical insurance card/information. Every medical card must have a member, a single member that is, hence the total participation if a medical card and the club member being singular. The cardinality of the medical insurance is “1..2” because a member can have multiple insurance cards, but it caps at two because realistically, nobody needs more than two medical carriers.

Purchases: This relationship describes how a club member can purchase a merchandise order. Not every club member is required to purchase a merchandise order; therefore, the cardinality between the purchases relationship on the member side is “0..1.” The customer (member in this case) can purchase multiple merchandise orders, hence the cardinality of “1..*”: there is one to multiple merchandise orders per member. In addition, the merchandise order is total, meaning every merchandise order must have a member associated with it.

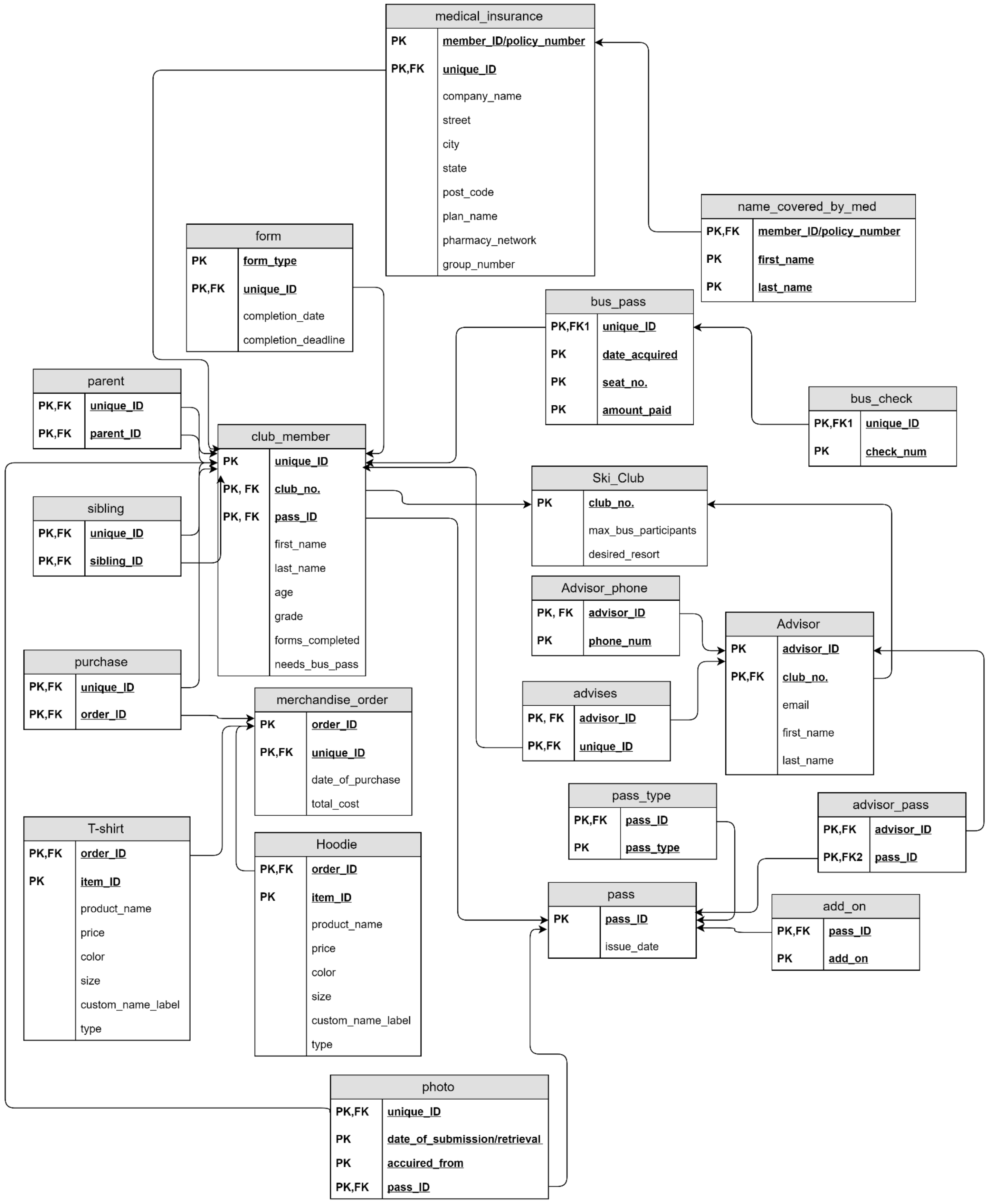
Submitted_by: Describes the need for a merchandise order to be submitted/turned-in by an advisor for processing by hometown design. An advisor can submit multiple merchandise orders and thus the cardinality of merchandise order is “1..*.” Merchandise order is total participation because every order must be submitted by an advisor. The advisor, however, is not mandatory because an advisor is not required to submit an order.

Contains: A merchandise order contains merchandise items, hence the contains relationship. This is a one-to-many relationship as there is one order containing many items (for that very reason, the cardinality on the item side is “1..*”). Every item must participate in an order or its meaning is void.

The disjoint total specialization: In this case, every item is either a shirt or a hoodie, not both, hence the total disjoint specialization. The type variables of each can be misconceived; they do not denote the same variable, rather, they denote the type of hoodie (e.g. slim-fit, zipper, mini sweatshirt etc.) and the type of T-shit (e.g. long sleeve or short sleeve).

It is easier to normalize the relations if a relational schema is present. Thus, a relational schema is derived from the E-R diagram.

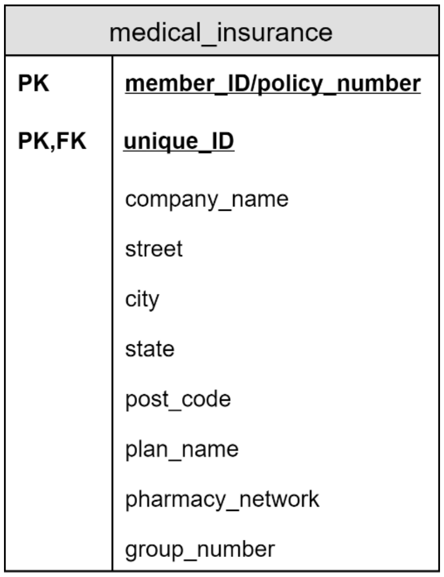
The Relational Schema Before Normalization:



Normalization

Every relation will be attempted to be presented in Boyce-Codd Normal Form (BCNF). BCNF is a stricter form of normalization than 3NF; therefore, BCNF will be employed. All valid FDs will be in blue and all new relations to be added or altered from BCNF decomposition will be in red.

medical_insurance:



medical_insurance	
PK	**member_ID/policy_number**
PK,FK	**unique_ID**
	company_name
	street
	city
	state
	post_code
	plan_name
	pharmacy_network
	group_number

member_ID/policy_number, unique_ID → company_name, street, city, state, post_code, plan_name, pharmacy_network, group_number

street, city, state, post_code → company_name

Explanation:

The FD of plan_name → pharmacy_network was considered; however, plan_name → pharmacy_network remains invalid as different plans can have different networks: "for example, CDPHP employer plans use a Premier network; CDPHP individual plans (like those through the healthcare exchange) use a Value network; and CDPHP plans for seniors use the Medicare network" (Filkins, 2019).

Considering that every geographical address is unique, an address could imply the company name. The FD is derived: street, city, state, post_code → company_name. Because we can have two company names that are the same, we cannot derive company_name → street, city, state, post_code.

Decomposition:

R_1 = (member_ID/policy_number, unique_ID, company_name, street, city, state, post_code, plan_name, pharmacy_network, group_number)

R_1 is not in BCNF because the LHS of "street, city, state, post_code → company_name" is not a candidate key.

The relations are decomposed:

R_2 = (street, city, state, post_code, company_name) and

R_3 = (member_ID/policy_number, unique_ID, street, city, state, post_code, plan_name, pharmacy_network, group_number)

R_2 and R_3 are both in BCNF according to the functional dependencies. However, we can see that "street, city, state, post_code → company_name" caused the relation to be invalid, and company_name is thus omitted in R_3 but "street, city, state, post_code" are left. "street, city, state, post_code" correspond to the company_name and thus it is simpler to define a new attribute denoting the company which generates the FD: company_ID → street, city, state, post_code, company_name.

If we reiterated through the algorithm, we would then get our final decomposition:

R = {(company_ID, street, city, state, post_code, company_name) (member_ID/policy_number, unique_ID, company_ID, plan_name, pharmacy_network, group_number)}

name_covered_by_med:

name_covered_by_med	
PK,FK	member_ID/policy_number
PK	first_name
PK	last_name

member_ID/policy_number → first_name, last_name

Decomposition:

The relation is already in BCNF because member_ID/policy_number is on the LHS of the only FD and it is a super key.

form:

form	
PK	form_type
PK,FK	unique_ID
	completion_date
	completion_deadline

form_type, unique_ID → completion_date, completion_deadline

Explanation:

We cannot have “form_type → completion_date, completion_deadline” because we need the corresponding unique_ID (which indicates the submitter of the form), nor can we have “unique_ID → completion_date, completion_deadline”.

Decomposition:

The relation is already in BCNF because “form_type, unique_ID” on the LHS of the only functional dependency is a candidate key.

bus_pass:

bus_pass	
PK,FK1	unique_ID
PK	date_aquired
PK	seat_no.
PK	amount_paid

unique_ID → date_acquired, seat_no., amount_paid

Explanation:

We cannot derive anymore functional dependencies because date_acquired, seat_no., and amount_paid cannot imply anything other than themselves.

Decomposition:

This relation is already in BCNF because unique_ID is a super key for the relation (i.e. unique_ID → date_acquired, seat_no., amount_paid).

bus_check:

bus_check	
PK,FK1	unique_ID
PK	check_num

unique_ID → check_num

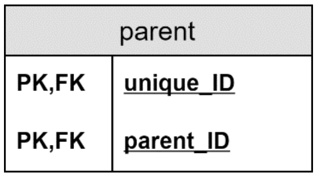
Explanation:

The ID of the club member gives us their corresponding check number.

Decomposition:

This relation is already in BCNF as the unique_ID is a super key for the relation via unique_ID → check_num.

parent:



parent	
PK,FK	unique_ID
PK,FK	parent_ID

No functional dependencies (see explanation)

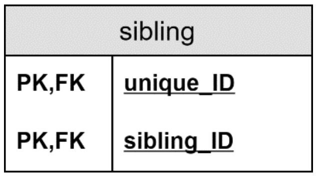
Explanation:

This relation was created because there is a many-to-many relationship between the parent and the child of that parent. Therefore, unique_ID → parent_ID is invalid as a child can have multiple parents, and parent_ID → unique_ID is invalid as a parent can have multiple children.

Decomposition:

Already in BCNF because the relation merely encompasses key attributes.

sibling:



sibling	
PK,FK	unique_ID
PK,FK	sibling_ID

No functional dependencies (see explanation)

Explanation:

This relation has no functional dependencies as a sibling can have multiple siblings. Thus, no functional dependencies hold in this relation for the same reason they do not hold in the parent relation.

Decomposition:

Already in BCNF because the relation merely encompasses key attributes.

club_member:

club_member	
PK	**unique_ID**
PK, FK	**club_no.**
PK, FK	**pass_ID**
	first_name
	last_name
	age
	grade
	forms_completed
	needs_bus_pass

unique_ID, club_no., pass_ID → first_name, last_name, age, grade, forms_completed, needs_bus_pass

Explanation:

The primary key of this relation implies all the other tuples as it should. There are no other functional dependencies capable of being derived from this relation.

Decomposition:

All the functional dependencies within this relation contain keys on the LHS, meaning it is already in BCNF.

Ski_Club:

Ski_Club	
PK	**club_no.**
	max_bus_participants
	desired_resort

club_no. → max_bus_participants, desired_resort

Explanation:

The club number is the unique identifier of a club and thus implies all other attributes that a club may hold (i.e. max_bus_participants, desired_resort). No club number will appear twice in with different values for max_bus_participants, desired_resort; the FD of "**club_no.** → max_bus_participants, desired_resort" holds.

Decomposition:

The LHS of "**club_no.** → max_bus_participants, desired_resort" is a key and thus the relation is in BCNF.

purchase:

purchase	
PK,FK	unique_ID
PK,FK	order_ID

order_ID → unique_ID

Explanation:

One club member can order multiple times and thus unique_ID → order_ID does not hold; moreover, an order has to be attached to a single club member, so order_ID → unique_ID does hold.

Decomposition:

Again, the relation is already in BCNF as there are only two attributes which are keys.

merchandise_order:

merchandise_order	
PK	order_ID
PK,FK	unique_ID
	date_of_purchase
	total_cost

order_ID, unique_ID → date_of_purchase, total_cost

order_ID → date_of_purchase, total_cost, **unique_ID**

Explanation:

"Order_ID, unique_ID → date_of_purchase, total_cost" holds because an order ID alone can determine everything else in the relation. Hence, "order_ID → date_of_purchase, total_cost, unique_ID" also holds. Notice we can also retrieve unique_ID from the order_ID because there is one order per club member.

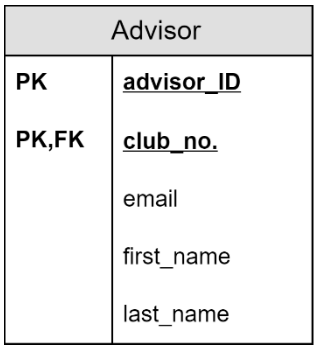
Decomposition:

In this case, the FDs are not in minimal form. Iterating through the algorithm, the first thing to note is {order_ID, unique_ID}$^+$ gives us unique_ID through "order_ID → date_of_purchase, total_cost, unique_ID." As a result, unique_ID on the LHS of "order_ID, unique_ID → date_of_purchase, total_cost" is removed.

The minimal cover is:

order_ID → date_of_purchase, total_cost, unique_ID which denotes the table is already in BCNF as order_ID is a candidate key.

Advisor:



Advisor	
PK	**advisor_ID**
PK,FK	**club_no.**
	email
	first_name
	last_name

advisor_ID → club_no., email, first_name, last_name

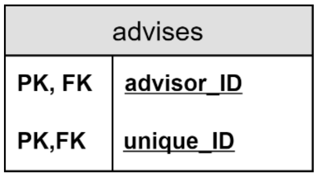
Explanation:

The advisor's ID can gives us every other attribute because and ID is unique to the advisor. A club cannot imply an advisor for obvious reasons: there are many different advisors to one club.

Decomposition:

This table is already in BCNF as the only FD that exists is "advisor_ID → club_no., email, first_name, last_name"

advises:



advises	
PK, FK	**advisor_ID**
PK,FK	**unique_ID**

No functional dependencies (see explanation)

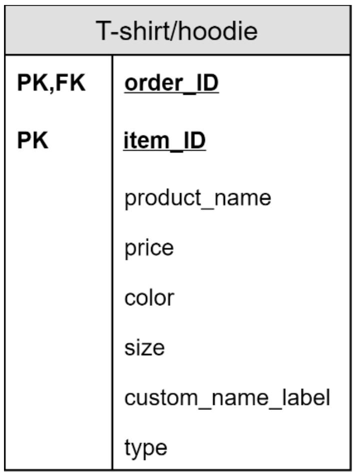
Explanation:

The advises relation represents a many-to-many relationship. As it only has two attributes referencing the advisor relation, and it is a many-to-many relationship, there are no functional dependencies.

Decomposition:

Already in BCNF because the relation merely encompasses key attributes.

T-shirt/hoodie:



T-shirt/hoodie	
PK,FK	order_ID
PK	item_ID
	product_name
	price
	color
	size
	custom_name_label
	type

order_ID, item_ID →product_name, price, color, size, custom_name_label, type

item_ID → product_name, price, color, size, custom_name_label, type

Explanation:

The hoodie and the T-shirt relations are represented by one in the decomposition because the attributes are the same. However, it is important to refresh why there are two relations denoting a hoodie and a T-shirt. Recall that there exists an IS-A relationship between hoodie/T-shirt and a merchandise order as a merchandise can contain either T-shirts and hoodies (and multiple of each). The type variable differs from the T-shirt and hoodie relation. The item relation was not added because the relationship between them was disjoint. Now, as to the functional dependencies, it is safe to say the item id can identify everything in the relation, hence why we have "item_ID → product_name, price, color, size, custom_name_label, type." Notice that we cannot get order_ID from item_ID because the same item can be a part of different orders. In addition, we cannot get the corresponding items from the order_ID because the order is likely to have multiple items; nonetheless, we can obtain the item from the order_ID in conjunction with the item_ID, hence "order_ID, item_ID → product_name, price, color, size, custom_name_label, type."

Decomposition:

Compute the canonical cover:

Minimizing the LHS of the FDs we see that order_ID is extraneous. The minimum cover remains:

item_ID → product_name, price, color, size, custom_name_label, type

The only candidate key is {order_ID, item_ID} as they are not on the LHS of any FD in the canonical cover.

Starting with the entire relation:

R_1 = (item_ID, product_name, price, color, size, custom_name_label, type)

R_1 violates BCNF as FD “item_ID $\rightarrow$ product_name, price, color, size, custom_name_label, type” is not trivial (as order_ID is not implied) and item_ID is not a super key. The relation is split:

R_2 = (order_ID, item_ID) R_3 = (item_ID, product_name, price, color, size, custom_name_label, type)

R_2 itself is a canidate key and “item_ID $\rightarrow$ product_name, price, color, size, custom_name_label, type” now satisfies R_3. Thus, the new relation is R = (R_2, R_3).

R_3 will represent a new relation that denotes orders, and order_ID will be removed from the original R_1.

R = {(order_ID, item_ID) (item_ID, product_name, price, color, size, custom_name_label, type)}

Advisor_phone:

Advisor_phone	
PK, FK	**advisor_ID**
PK	**phone_num**

No functional dependencies (see explanation)

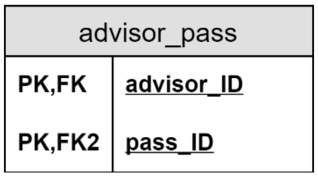
Explanation:

Ths relation denotes a multivalued attribute. An advisor cannot always imply a specific phone number as they can have multiple, and a phone number cannot imply a specific advisor; therefore, there are no functional dependencies.

Decomposition

Already in BCNF because the relation merely encompasses key attributes.

advisor_pass:



advisor_pass	
PK,FK	advisor_ID
PK,FK2	pass_ID

pass_ID → advisor_ID

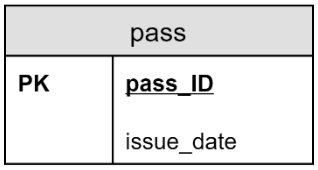
Explanation:

A pass is unique, and thus can imply an advisor. However, because an advisor can have multiple passes, an advisor cannot imply a specific pass.

Decomposition:

The relation is already in BCNF. In “pass_ID → advisor_ID”, pass_ID is a super key.

pass:



pass	
PK	pass_ID
	issue_date

pass_ID → issue_date

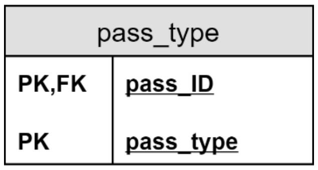
Explanation:

A specific pass can imply the date in which it was issued; however, multiple passes can be issued on one day, so “issue_date → pass_ID” is invalid.

Decomposition:

As the pass_ID is a super key and “pass_ID → issue_date” is the only functional dependency, the relation is already in BCNF.

pass_type:



pass_type	
PK,FK	pass_ID
PK	pass_type

None (see explanation)

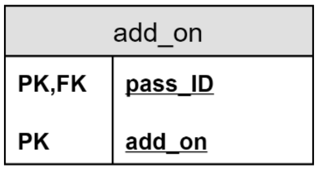
Explanation:

Pass_type was derived from a multivalued attribute in pass. A pass (pass_ID) can have multiple types (e.g. rentals & every day pass, no rentals & weekly pass). A pass could appear multiple times in this relation with different types; therefore, there are no functional dependencies.

Decomposition:

This relation is already in BCNF as it composed of just keys and there is no functional dependencies.

add_on:



add_on	
PK,FK	pass_ID
PK	add_on

None (see explanation)

Explanation:

Add_on was derived from a multivalued attribute in pass. A pass (pass_ID) can have multiple add-ons. Thus, a pass could appear multiple times in this relation, denoting a different add_on each time.

Decomposition:

This relation is already in BCNF as it composed of just keys and there is no functional dependencies.

photo:

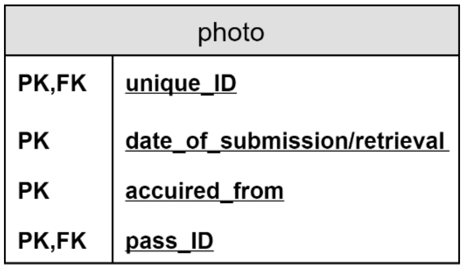


photo	
PK,FK	unique_ID
PK	date_of_submission/retrieval
PK	accuired_from
PK,FK	pass_ID

unique_ID → date_of_submission/retrieval, accuired_from, pass_ID

pass_ID → unique_ID, date_of_submission/retrieval, accuired_from

Explanation:

Both the unique_ID of the club_member denoting this pass or the pass_ID can infer everything in the relation as they are unique and there is only one photo to a club member.

Decomposition

R_1 = (unique_ID, date_of_submission/retrieval, accuired_from, pass_ID)

Test "unique_ID → date_of_submission/retrieval, accuired_from, pass_ID": unique_ID is a super key and is on the LHS.

Test "pass_ID → unique_ID, date_of_submission/retrieval, accuired_from": pass_ID is a super key and is on the LHS

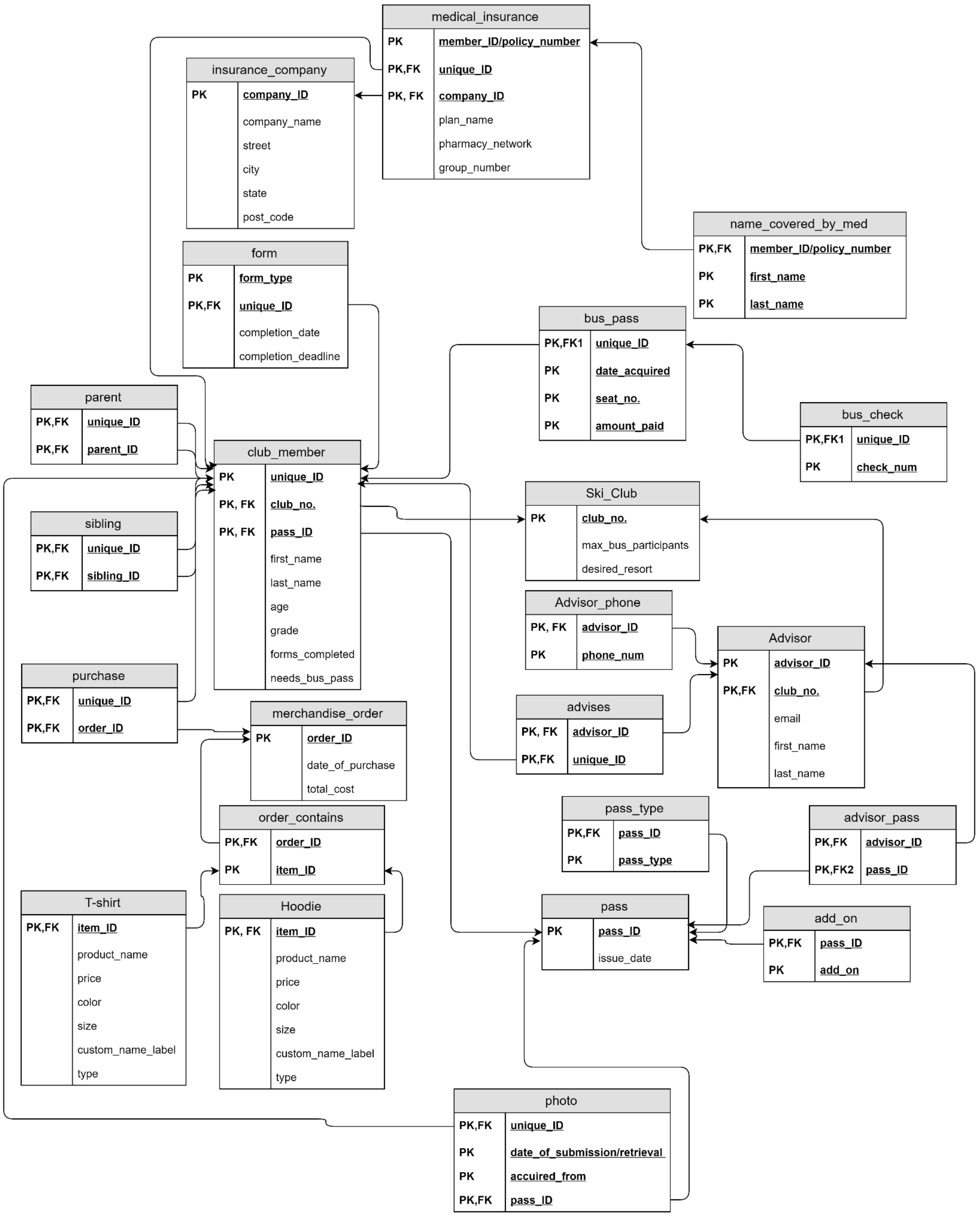
R_1 is in BCNF

Changes

To get the entire relational schema into BCNF, the following must be done:

- Add a company relation from medical insurance
    - R = {(company_ID, street, city, state, post_code, company_name) (member_ID/policy_number, unique_ID, company_ID, plan_name, pharmacy_network, group_number)}
- Add an order_contains relation from T-Shirt/Hoodie
    - R = {(order_ID, item_ID) (item_ID, product_name, price, color, size, custom_name_label, type)}

The Relational Schema After Normalization (PKs and FKs included):



Simplifying the database:

The design is now complete; however, it seems as if some of the relations are besides the point of the original problem domain and the database itself is very large (and would take immense amounts of time to implement). Thus, I have simplified the database by removing some tables that are beside the point of the database. The explanation for each removal is given here:

Parent: The parent relation helps denote a parent of a corresponding child. However, there is much more practical way of doing this. It may be important to know is a club member is a student or a parent; hence, we do not create a separate relation, rather, we can just add a Boolean attributed named is_student.

Insurance_Company: The entire purpose of storing medical information is to have it in the case of an injury. The medical_insurance relation contains enough information to identify an insurer in the case of an accident. Thus, insurance_company is redundant.

Name_covered_by_med: This relation denotes the names covered under a specific medical insurance plan. This is valuable information. Nonetheless, the entire purpose of storing medical information is to have it in the case of an injury, and because medical_insurance already corresponds to the club member via FK of unique_ID, this relation is redundant.

Photo: The photo, whether one is present or not, is on the pass. Therefore, it is redundant to store information on the photo as snow trails handles and makes the pass. This is indeed permissible because even if the student does not submit a photo, their school photo is used.

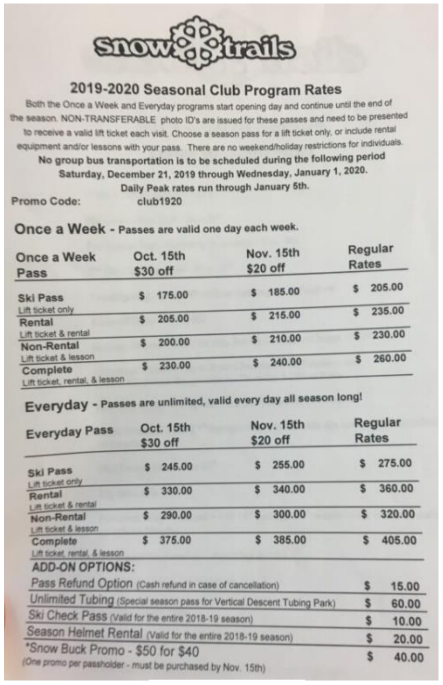
Advisor_phone: The advisors phone number is represented by a multivalued attribute in this relation because the assumption of an advisor having multiple phone numbers is made. However, the point in having a phone number in the first place is to be able to contact an advisor. It is safe to say that, especially in 2019, an advisor has one preferred phone number in which they want to be primarily contacted by.

T-shirt/hoodie: The only thing keeping these relations from becoming one item is the type variable. It would be simpler if the definition of an extra attribute was added and thereby the relations can be combined into one item relation. The variable item_ID, instead, will be used to denote all the types of items and is unique.

Bus_check: The check# is multivalued because the assumption of multiple checks is made. However, the bus pass must be paid in full if a check is submitted; this undermines the original specification. Since check# cannot be multivalued anymore, we remove the bus_check relation and add check# to the bus_pass relation.

Advises: The advisers do not advise specific individuals, rather, after talking to the administer of the club, the advisers are collectively in change of the students. Since they are collectively in charge of the members, we cannot assign specific individuals to specific advisors.

Pass_type: After further analysis of figure 1, there are a total of 5 add-ons. Multiple add-ons are able to be added to the pass, and thus the add_ons relation remains as these attributes are actually multivalued. The pass_type is also multivalued. The types are once a week or everyday (easier defined as pass_days_validity attribute), and the other four choices are the type. Thus, we add pass_days_validity to the pass relation, and move type to the same relation, removing pass_type.


snow trails

2019-2020 Seasonal Club Program Rates

Both the Once a Week and Everyday programs start opening day and continue until the end of the season. NON-TRANSFERABLE photo ID's are issued for these passes and need to be presented to receive a valid lift ticket each visit. Choose a season pass for a lift ticket only, or include rental equipment and/or lessons with your pass. There are no weekend/holiday restrictions for individuals.

No group bus transportation is to be scheduled during the following period Saturday, December 21, 2019 through Wednesday, January 1, 2020.

Daily Peak rates run through January 5th.

Promo Code: club1920

Once a Week - Passes are valid one day each week.

Once a Week Pass	Oct. 15th $30 off	Nov. 15th $20 off	Regular Rates
Ski Pass Lift ticket only	$ 175.00	$ 185.00	$ 205.00
Rental Lift ticket & rental	$ 205.00	$ 215.00	$ 235.00
Non-Rental Lift ticket & lesson	$ 200.00	$ 210.00	$ 230.00
Complete Lift ticket, rental, & lesson	$ 230.00	$ 240.00	$ 260.00

Everyday - Passes are unlimited, valid every day all season long!

Everyday Pass	Oct. 15th $30 off	Nov. 15th $20 off	Regular Rates
Ski Pass Lift ticket only	$ 245.00	$ 255.00	$ 275.00
Rental Lift ticket & rental	$ 330.00	$ 340.00	$ 360.00
Non-Rental Lift ticket & lesson	$ 290.00	$ 300.00	$ 320.00
Complete Lift ticket, rental, & lesson	$ 375.00	$ 385.00	$ 405.00

ADD-ON OPTIONS:	
Pass Refund Option (Cash refund in case of cancellation)	$ 15.00
Unlimited Tubing (Special season pass for Vertical Descent Tubing Park)	$ 60.00
Ski Check Pass (Valid for the entire 2018-19 season)	$ 10.00
Season Helmet Rental (Valid for the entire 2018-19 season)	$ 20.00
*Snow Buck Promo - $50 for $40 (One promo per passholder - must be purchased by Nov. 15th)	$ 40.00

Figure 1

Add_ons: although the add_ons relation is completely correct, the school does not need to know this information. The school only needs to know if the member has obtained a pass, not the specifics of it. For that reason, add_ons is removed (it is not stored on the spreadsheet I was handed).

Advisor_pass: This relation denotes a many-to-many relationship in which an advisor holds a pass. However, as found in the CDM submission, every advisor is supplied with a generic pass: they need not supply the school with information on their passes because Snow Trails holds the records. If they are in the advisor relation, they have a pass, the school does not need to store this information (it was not stored on the spreadsheet I was supplied with).

Merchandise_order, purchase, order_contains: These relations are simplified and then combined into one relation called member_order. A member submits an order, and thus it is easier to combine the purchase with the merchandise_order. A member can place many orders, hence why the new relation has unique_ID FK in it. The order ID still helps uniquely identify it, and the other attributes give us the specifics as to what is in the order itself.

Sibling: The whole point of the sibling relation is to determine siblings and thereby change the price of the bus pass. However, the school keeps siblings on record in which they determine the bus pass price, so we just need to include the amount paid in the bus_pass relation.

The schema is still in BCNF because each relation only has functional dependencies that are superkeys on the LHS:

Club_member:

unique_ID, club_no.→ first_name, last_name, age, grade, forms_completed, needs_bus_pass, is_student

unique_ID → club_no., first_name, last_name, age, grade, forms_completed, needs_bus_pass, is_student

Form:

form_type, unique_ID → completion_date

Cannot have unique_ID → completion_date because a member could send in more than one form in one day.

Medical_insurance:

member_ID/policy_number, unique_ID → plan_name, pharmacy_network, group_number

Bus_pass:

unique_ID**, b_pass_ID** → check#, amount_paid

b_pass_ID → unique_ID , check#, amount_paid

Ski_club:

club_no. → max_bus_participants, desired_resort

Advisor:

advisor_ID → club_no., email, first_name, last_name

advisor_ID, club_no. → email, first_name, last_name

Pass:

Pass_ID → unique_ID, pass_days_validity, type

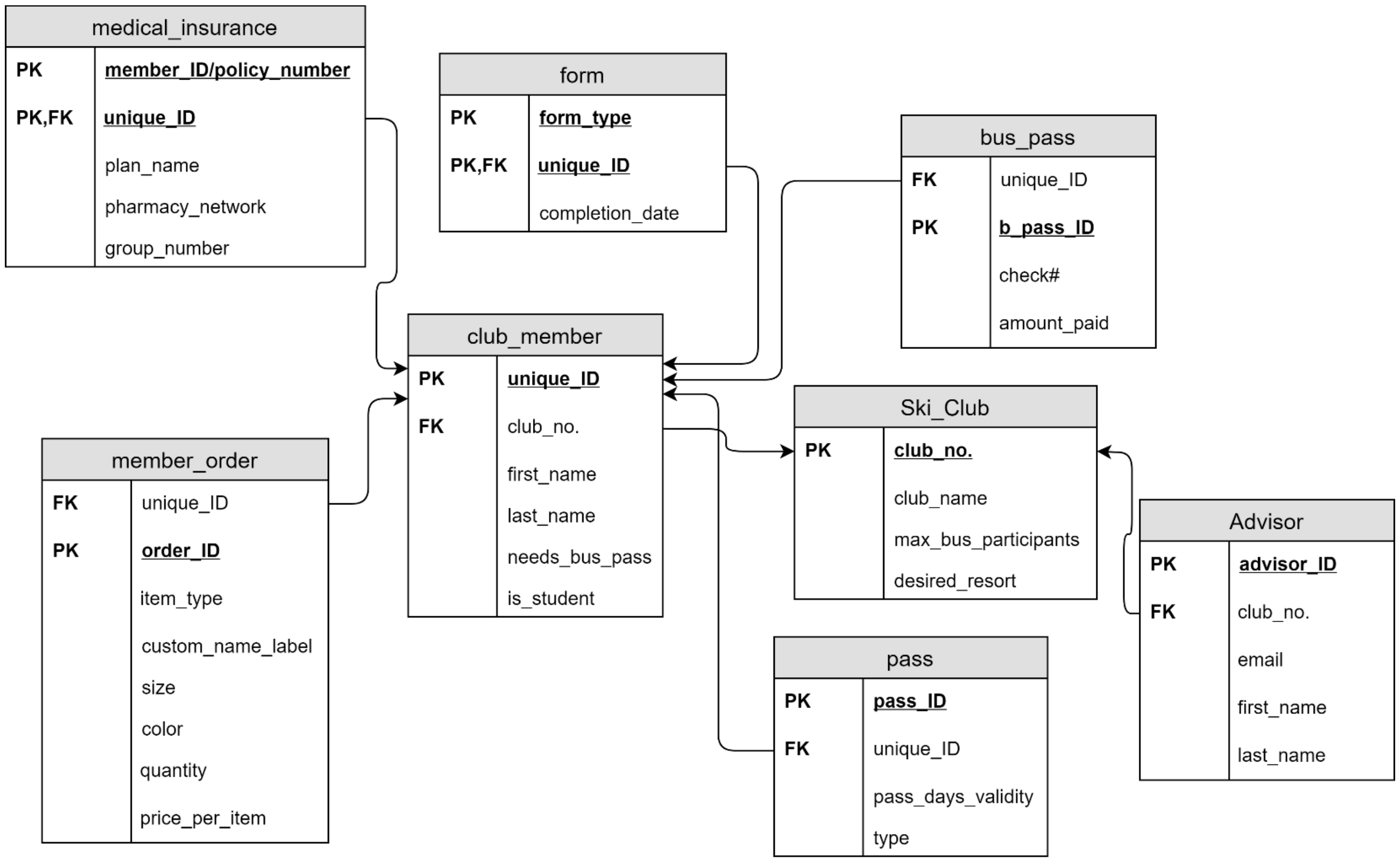
Pass_ID, unique_ID → pass_days_validity, type

Member_order:

unique_ID**, order_ID** → item_type, custom_name_label, size, color, quantity, price_per_item

order_ID → unique_ID, item_type, custom_name_label, size, color, quantity, price_per_item

The Final Schema



Referential integrity constraints:

Club_member:

The club member contains a foreign key denoted as club_no. club_member is the referencing relation and the club_no. FK in club member is referencing the Ski_Club relation.

On delete: SET NULL because the member may simply be switching clubs and we do not want to delete all their data just because they are not a part of a specific club anymore.

On update: CASCADE because the club that a member is a part of must be updated if the club identifier changes in the ski club relation.

Form:

The form relation has a foreign key that references the club member relation. The form relation is used to denote the completion of a form by a specific member, hence why we use the members unique_ID to reference the club member relation.

On delete: CASCADE because the completion stamp of a form for that club ember should no longer exist as the club member does not exist. In addition, SET NULL cannot be implemented because the unique_ID in form is necessary for uniquely identifying a tuple and thus cannot be NULL.

On update: CASCADE because if the unique ID of the member is changed (which is highly unlikely with a sound design) the child ID in form must also be updated.

Medical_insurance:

The medical insurance relation contains a foreign key denoting the club member associated with a particular medical card that references the club member relation.

On delete: CASCADE because we no longer need the medical information of that member because they no longer exist in the database.

On update: CASCADE because if the unique ID of the member is changed (which is highly unlikely with a sound design) the child ID in medical insurance must also be updated so the medical insurance does not correspond with the wrong member.

Bus_pass:

The bus pass relation has a foreign key (unique_ID) which references the club member to denote who owns the bus pass.

On delete: SET NULL: It is imperative that the database keeps the purchase of the pass and the pass on record because it was purchased. The member can refund their purchase and thus it is important to keep the bus pass on file even if the corresponding member was deleted. They still purchased the pass and it still exists if a club member is removed.

On update: CASCADE because if the unique ID of the member is changed (which is highly unlikely with a sound design) the child ID in the bus pass must update also to reflect the correct member ownership of the pass.

Ski_club:

Does not reference anything. Although it is referenced which is explained in the advisor and club member section.

Advisor:

The advisor has a foreign key which references the club they are an advisor of. The foreign key club_no references the primary key in the ski club relation.

On delete: SET NULL because the advisor may simply be switching clubs and we do not want to delete all their data just because they are not a part of a specific club anymore.

On update: CASCADE so the advisor reflects the correct club.

Pass:

The pass relation contains a foreign key which references the club member relations primary key to denote the owner of the pass.

On delete: SET NULL because the pass still exists. It was purchased and cannot be refunded so it should still exist even though it would not reference a club member.

On update: CASCADE so the id change in the club member reflects the change and corresponds with the correct club member.

Member_order:

The member order relation contains a foreign key which references the unique ID in the club member relation so that an order can correspond with the club member that placed it.

On delete: SET NULL. Again, this is a purchased item and even if the club member were to be removed from the database the purchase of the order still remains and should be processed as it is non-refundable.

On update: CASCADE so the order reflects any changes to the key of the club member and corresponds to the right club member.

Indexing

Postgresql creates indices for primary key attributes automatically. In the schema design, there are several foreign keys that are not declared primary. It is a good idea to declare indices for non-key foreign key attributes because joins utilizing these non-key foreign keys are very common. For instance, acquiring the orders associated with a specific member would require a join of club_member ⋈ member_order, and member_order if referencing club_member with a non-key attribute. Creating indices for these can significantly speed up the join queries.

For the reasons stated above, indices are created on these attributes:

- unique_ID in member_order
- club_no. in club_member
- unique_ID in pass
- club_no. in advisor
- unique_ID in bus_pass

Considering the functional requirements of the database, we will often need to know whether a member who needs a bus pass has a bus pass purchased. To do this, we will need to search through needs_bus_pass often. Thus, definition of one more index will succor database efficiency:

- need_bus_pass in club_member

References

Filkins, D. (2019, March 5). Understanding Your Health Insurance ID Card. Retrieved from https://blog.cdphp.com/how-to/understanding-health-insurance-id-card/