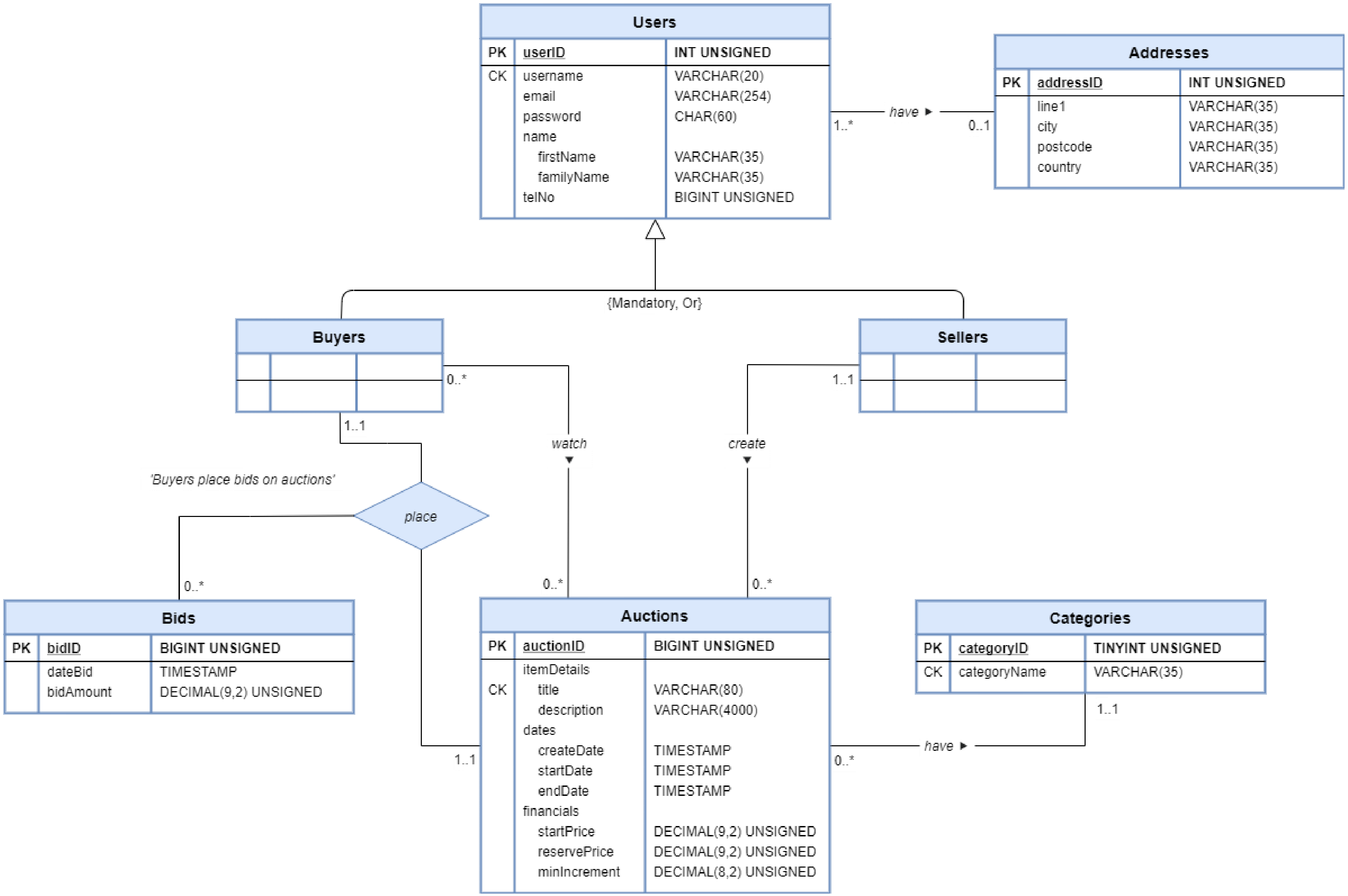
Database design wireframe

1 – Conceptual design (building conceptual data model)

- 1.1 - *Identify entity types*
  - Data dictionary documenting entities (or 'entity dictionary')
    - Highlight requirements table
- 1.2 - *Identify relationship types*
  - List of entities and relationships e.g. "Sellers create auctions"
    - Highlight requirements table
  - First-cut ER diagram without attributes incl. multiplicity
  - Check for fan and chasm traps
  - Augmented data dictionary (or 'relationship dictionary')
- 1.3 - *Identify and associate attributes with entity or relationship types in ERD*
  - Identify entity and relationship attributes (highlight requirements table)
    - Simple/composite
    - Single/multi-valued
    - Derived
      - Note here about derived attributes – many are 'natural' and not going to break 3NF (later), but generally not a good idea unless clear evidence of a performance advantage
  - Check only associated with one entity
  - Augmented data dictionary (or 'attribute dictionary')
- 1.4 - *Determine attribute domains*
  - Define domains of each attribute
    - Allowable set of values (e.g. M or F)
    - Sizes and formats (e.g. 1 character)
  - Add to 'attribute dictionary'
- 1.5 - *Determine candidate, primary, and alternate key attributes*
  - Identify CKs: minimal set of attributes uniquely identifying occurrence of the entity
    - PK: chosen for identification (PPK if composite)
      - Include userID so users can change username easily; possibly shorter
      - Include addressID to reduce data duplication; possibly shorter
      - Include auctionID so can change title; possibly shorter
      - Include categoryID so category name can be changed (e.g. to superset); possibly shorter
      - Include bidID so have a shorter identifier (than incl. buyers, auctions)
    - AK: other CKs
      - username

            - (auction) title
            - categoryName
    - Select based on size, probability of values being changed, shortest length or easiest to use
    - Add to data dictionary
    - At this stage, also identify strong and weak entities
        - *Strong:* can assign a primary key to the entity
        - *Weak:* cannot identify a primary key to the entity. PK of weak entity can be identified only once we've mapped the weak entity and its relationship with its 'owner' entity' to a relation through placement of a foreign key in the relation.
- 1.6 - *Consider use of enhanced modelling concepts (optional step)*
    - Specialisation/generalization (specific instance of the parent)
    - Aggregation (child can exist independently of parent)
    - Composition (child cannot exist independently of parent)
- 1.7 - *Check model for redundancy*
    - Remove redundant entities
        - Re-examine 1:1 relationships (i.e. don't have two entities representing same object)
    - Remove redundant relationships
        - i.e. don't have relationship where same information can be gained via other relationships, as developing a minimal data model. Be careful with the time dimension in assessing redundancy.
        - A cycle is a necessary but not a sufficient condition
- 1.8 - *Validate conceptual data model against user transactions*
    - Some examples of 'written' transactions (like SQL queries) and show how they are possible
- 1.9 - *Review conceptual data model with user*
    - Received feedback on week 3 draft of ER diagram in week 4



Users

PK	userID	INT UNSIGNED
CK	username	VARCHAR(20)
	email	VARCHAR(254)
	password	CHAR(60)
	name	
	firstName	VARCHAR(35)
	familyName	VARCHAR(35)
	telNo	BIGINT UNSIGNED

Addresses

PK	addressID	INT UNSIGNED
	line1	VARCHAR(35)
	city	VARCHAR(35)
	postcode	VARCHAR(35)
	country	VARCHAR(35)

Bids

PK	bidID	BIGINT UNSIGNED
	dateBid	TIMESTAMP
	bidAmount	DECIMAL(9,2) UNSIGNED

Auctions

PK	auctionID	BIGINT UNSIGNED
	itemDetails	
CK	title	VARCHAR(80)
	description	VARCHAR(4000)
	dates	
	createDate	TIMESTAMP
	startDate	TIMESTAMP
	endDate	TIMESTAMP
	financials	
	startPrice	DECIMAL(9,2) UNSIGNED
	reservePrice	DECIMAL(9,2) UNSIGNED
	minIncrement	DECIMAL(8,2) UNSIGNED

Categories

PK	categoryID	TINYINT UNSIGNED
CK	categoryName	VARCHAR(35)

2 – **Logical design** (building logical data model)

- 2.1 - *Derive relations for logical data model*
    - Superclass/subclass
        - {Mandatory, Or} so create one relation each for buyers and sellers
    - Strong entities (create a relation + flatten composite attributes)
        - Buyers
        - Sellers
        - Categories
    - Weak entities (create a relation + flatten composite attributes; PK already identified)
        - Bids
        - Auctions
        - Addresses
    - 1:* relationships ("1" side is designated parent and "*" side gets PK of parent as an FK)
        - Users (*) have addresses (1)
        - Sellers (1) *create* auctions (*)
        - Auctions (*) *have* categories (1)
    - *:* relationships (create a relation to represent the relationship incl. PKs of both entities participating as FKs and any other relevant attributes; one or both of FKs acts as PK)
        - Buyers (*) *watch* auctions (*)
    - Complex relationship types (create a relation representing the relationship incl. attributes part of the relationship, favouring PK of entity with 'many' cardinality near it)
        - Buyers (1) place bids (*) on auctions (1)
            - NB: This will just create a dedicated table as an extension of the bids table with FKs from Buyers and Auctions
            - Could replicate ER diagram as a visual 'schema', with added FKs and an extra 'Watching' table attached to 'watch' relationship (labelled 'Watching')
            - For 'Watching' table, don't add a new PK as not referenced elsewhere, so no advantage in terms of storage saving
- 2.2 - *Validate relations using normalization*
    - Check step-by-step that there are no FDs violating 3NF
        - Done – discussion around addresses needed (e.g. for the UK)
            - 4x4 for pairwise checking of quads, triples and doubles etc. for UK; if not true for UK then not true in general
            - Discussion around 'over normalization' and intention
        - Can systematically do this as in the book for each relation
- 2.3 - *Validate relations against user transactions*
    - Check actually works for given request manually
- 2.4 - *Check integrity constraints*
    - Required data – all data except Users telNo and Users addressID {FK}

    - Attribute domain constraints – already done earlier
    - Multiplicity – already done earlier;
        - 0:1 on users assumes addresses are optional in theory
        - Multiplicity constraints in general reflect understanding of requirements, not a wider range of what could be logically possible (e.g. users can stop an address from being associated from their account, which would imply a 0..* multiplicity.
    - Entity integrity – PKs identified, CKs (unique) identified
    - Referential integrity
        - Are FK nulls allowed? Yes, for addressID
        - How to ensure referential integrity? 2x3 of child, parent vs. insert, delete, update – 3 relevant DML functions; only interested in 4 cases (defined later)

- 2.5 - *Review logical data model with user*
    - Not done as this is an assessment
- 2.6 - *Merge logical data models into global model (optional step)*
    - Skipped as only creating a single user view, but in reality might want to create multiple views for different roles e.g. admin vs. data scientists
- 2.7 - *Check for future growth*
    - Discussion about extensibility – list of additional features e.g. rating and how difficult it would be to include

Schema:

Buyers (buyerID, username, email, password, firstName, familyName, telNo, *addressID*)

Sellers (sellerID, username, email, password, firstName, familyName, telNo, *addressID*)

Addresses (addressID, line1, city, postcode, country)

Bids (bidID, dateBid, bidAmount, *buyerID, auctionID*)

Watching (*buyerID*, *auctionID*)

Auctions (auctionID, title, description, createDate, startDate, endDate, startPrice, reservePrice, minIncrement, *sellerID, categoryID*)

Categories (categoryID, categoryName)

- Domains from before

- No nulls except addressID, telNo in Buyers, Sellers

- Default values:
  - Bids *buyerID*: *0 (N/A)*
  - Auctions *sellerID: 0 (N/A)*
  - Auctions *categoryID: 0 (Other)*

- Referential integrity:
  - On insertion of new record to child relation, check FK value equal to a value of existing parent tuple
  - On update of child FK value, check equal to a value of existing parent tuple
  - On update to parent PK value, cascade update to child entities' FK values
  - On deletion from parent tuple:
    - Buyers *addressID*: set null
      - *Since allowing null values anyway*
    - Sellers *addressID*: set null
      - *Since allowing null values anyway*
    - Bids *buyerID*: set default
      - *Mechanism to avoid allowing nulls generally but keep bid information for completed listing searches*
    - Bids *auctionID*: cascade
      - *If an auction is deleted, then no need for bid information*
    - Watching *buyerID* : cascade
      - If a buyer is deleted whilst watching an auction, no need for watching information
    - Watching *auctionID:* cascade
      - If an auction is deleted whilst someone is watching it, no need for watching information
    - Auctions *sellerID:* set default
      - *Mechanism to avoid allowing nulls generally but keep auction information for completed listing searchers*
    - Auctions *categoryID:* set default
      - *Allow auctions to keep a category: better to update but might lead to a strange organisation of categories*

3 – Physical design (translating logical data model for target DBMS)

- Has been done simultaneously (in part) with the above as know capabilities of MySQL
- Discussion of design representation of derived data:
    - Doesn't technically break 3NF for parents to include summarised data from child entities, but goes against spirit of normalisation by introducing redundancy and complexity
    - Might give performance advantages, especially for those which are accessed more than they are updated (e.g. numBids), and performance is critical in eCommerce applications
    - Reduce complexity and query in real-time when needed, but keep this as an option if performance is unsatisfactory – very important for eCommerce sites