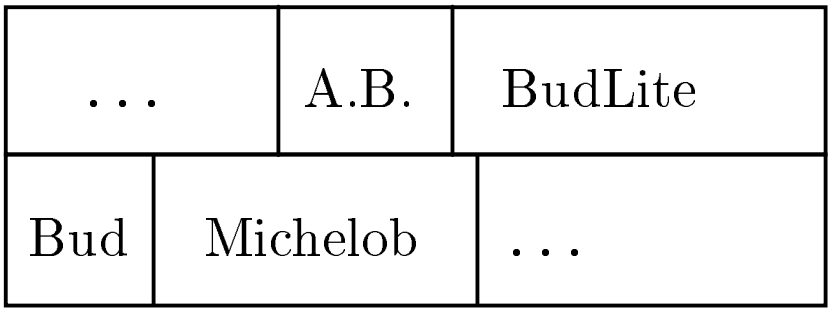
In the Begining . . .

(Historical data models: Network and Hierarchical)

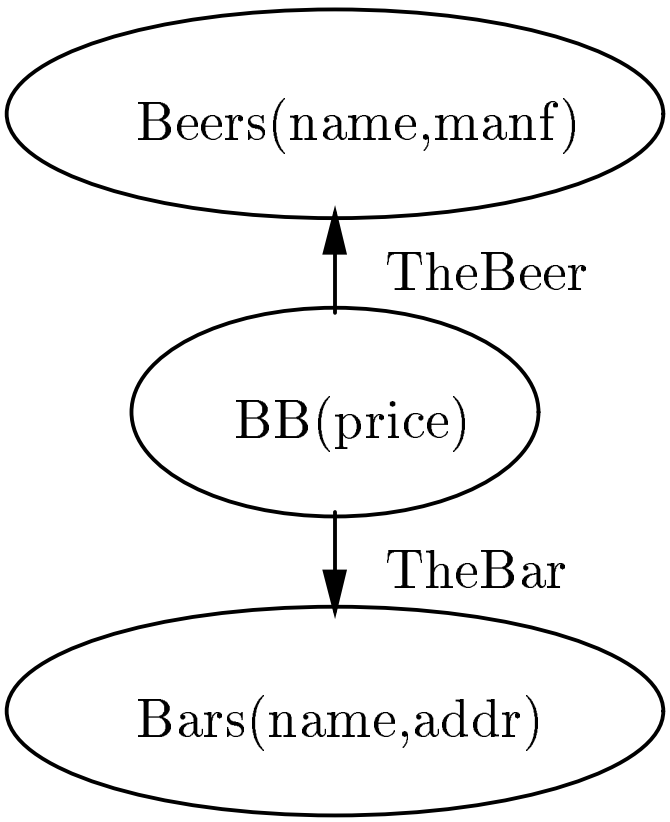
- The mid 1960's saw the first systems that used secondary storage for *querying* = retrieval by value, not by file name.
- Big difference: *secondary storage model of data* = data in blocks/pages, and major cost is retrieving or storing a *block*.
- *Unsolved problem*: Storing many-many relationships so they can be traversed efficiently in both directions.
  - ❖ Easy in RAM model: linked lists of successors, predecessors, e.g.
  - ❖ Many-one is easy in secondary-storage model, e.g., store each beer following its manufacturer, so "find the Anheuser-Busch beers" can be answered by retrieving them all on one or a few blocks.



Network Model

Essentially entity sets and binary, many-one relationships.

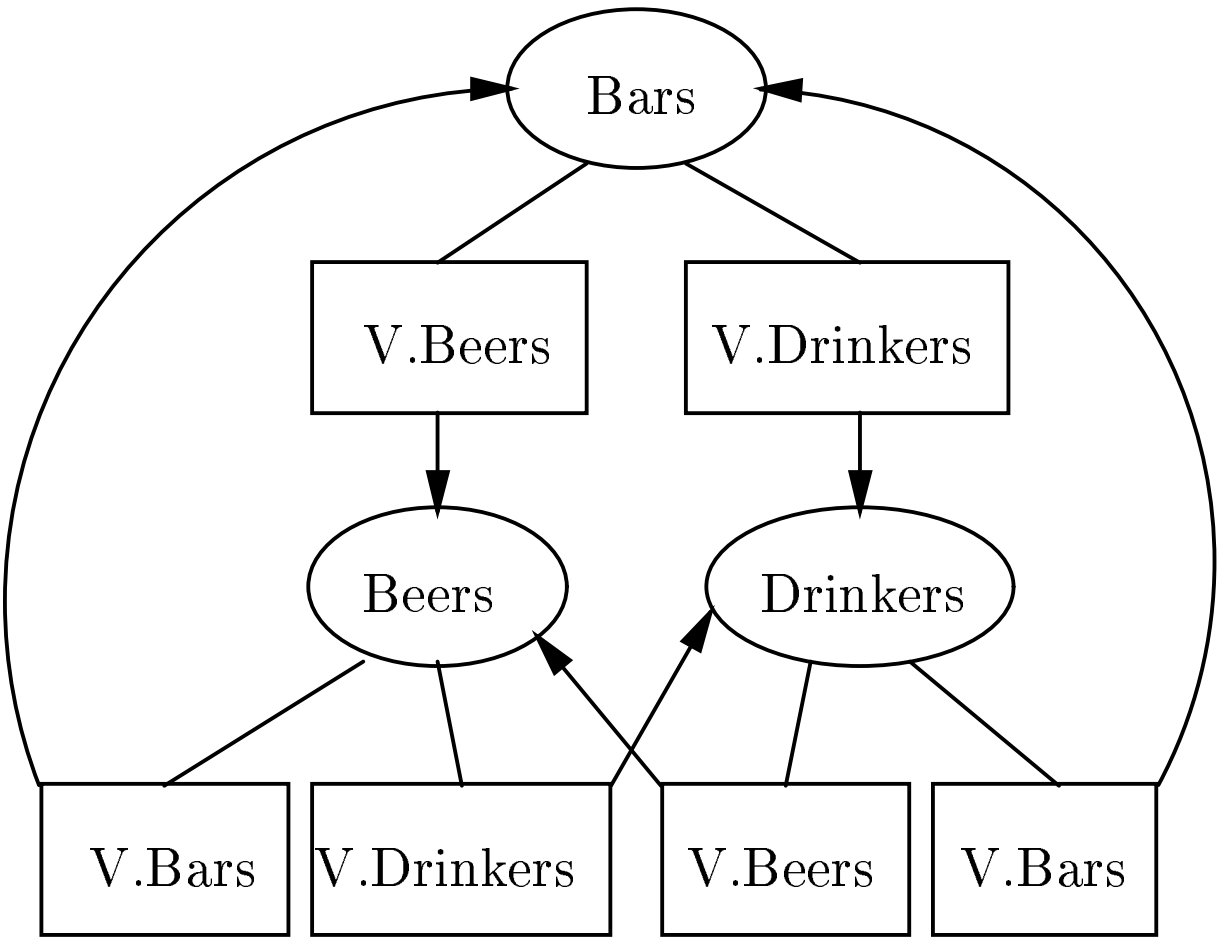
- Replace a many-many relationship by a connecting E.S. and two many-one relationships.
- Entity set → *Logical Record Type* (LRT).
- Many-one relationship → *Link*.
  - ❖ Terminology useful to this day: *owner* = one, *member* = many, e.g., a manufacturer record "owns" beer records.



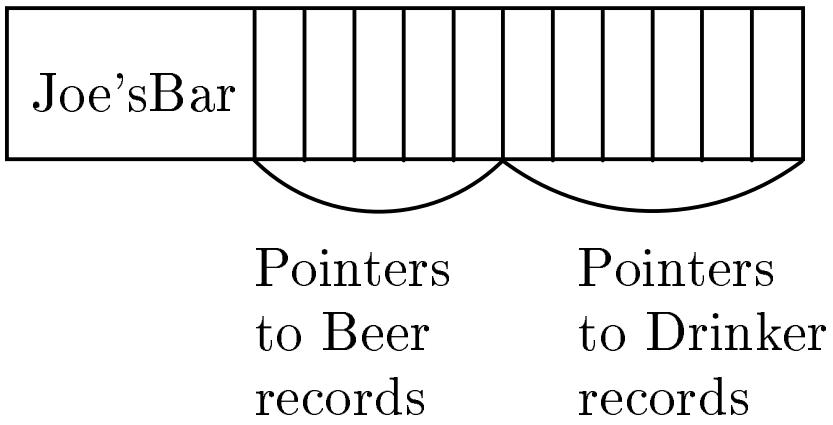
Hierarchical Model

Used in major early DBMS's, including IBM's IMS, which is still supported today.

- Network model, restricted to a forest, where owners are parents of children.
- Adds *Virtual LRT* to handle many-many relationships.
  - ❖ Think of V. LRT as representing pointers.

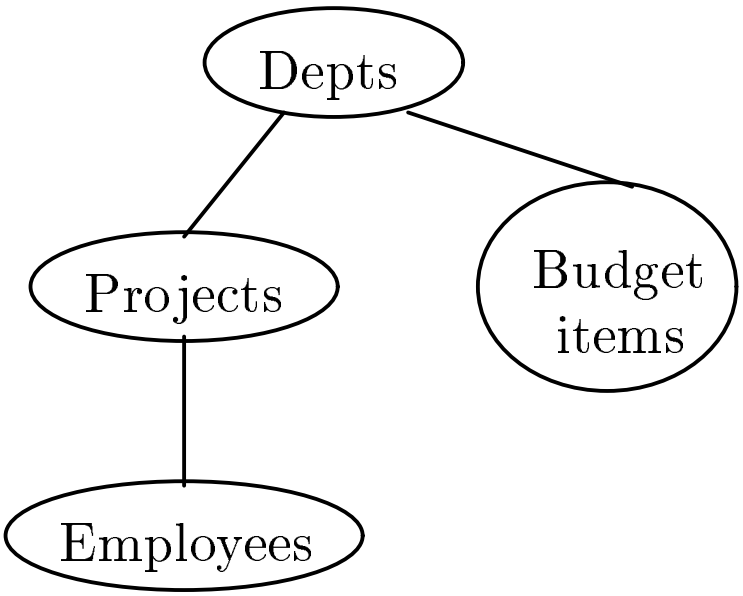


Typical Bar Record

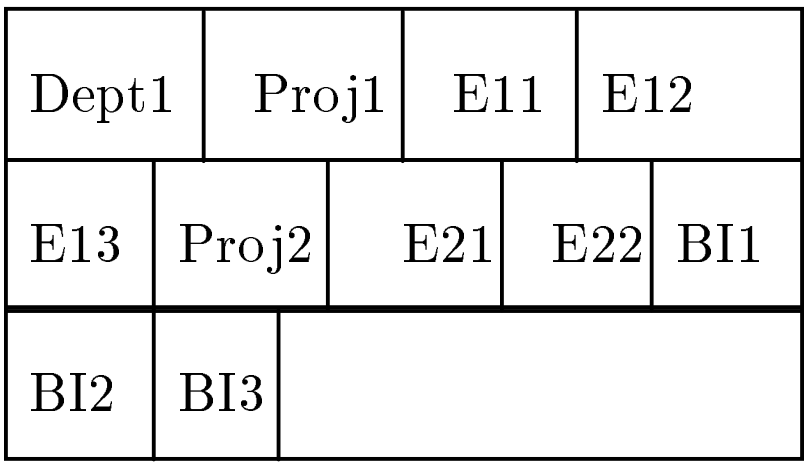


- No help in secondary storage model when going from a bar to either its beers or its drinkers.

Example Where Hierarchical Model Wins



- Typical stored records make efficient queries that go Dept → Budget Item or Project → Employees.



Relational Model

- Table = relation.
- Column headers = *attributes*.
- Row = *tuple*

name	manf
WinterBrew	Pete's
BudLite	A.B.
...	...

`Beers`

- *Relation schema* = name(attributes).
  Example: `Beers(name, manf)`.
  - ❖ Order of attributes is arbitrary, but in practice we need to assume the order given in the relation schema.
  - ❖ *Relation instance* is current set of rows for a relation schema.
- *Database schema* = collection of relation schemas.

Keys in Relations

An attribute or set of attributes K is a *key* for a relation R if we expect that in no instance of R will two different tuples agree on all the attributes of K.

- Indicate a key by underlining the key attributes.
- Example: If `name` is a key for `Beers`:

  `Beers(`<u>`name`</u>`, manf)`

Why Relations?

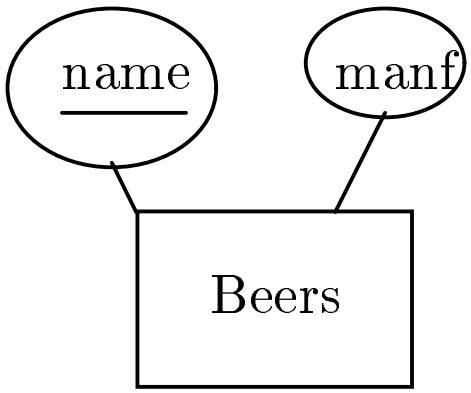
- Very simple model.
- *Often* a good match for the way we think about our data.
- Abstract model that underlies SQL, the most important language in DBMS's today.
  - ❖ But SQL uses "bags," while the abstract relational model is set-oriented.

Relational Design

- Relations are closer to real storage structures than the concepts of E/R or ODL.
  - ❖ Thus, going from E/R or ODL designs to relational often requires some additional intellectual input.

Easiest Case: Entity Set → Relation

E. S. attributes become relational attributes.



Becomes:

```
Beers(name, manf)
```

Slightly Harder: ODL Class Without Relationships

- Problem: ODL allows attribute types build from structures and collection types.
- Structure: Make one attribute for each field.
- Set: make one tuple for each member of the set.
  - ❖ More than one set attribute? Make tuples for all combinations.
- Problem: ODL class may have no key, but we should have one in the relation to represent "OID."

Example

```
interface Drinkers (key name) {
    attribute string name;
    attribute Struct Addr
        {string street, string city,
        int zip} address;
    attribute Set<string> phone;
}
```

<u>name</u>	street	city	zip	p<u>hone</u>
n_1	s_1	c_1	z_1	p_1
n_1	s_1	c_1	z_1	p_2

- Surprise: the key for the class (name) is not the key for the relation (name, phone).
  - ❖ `name` in the class determines a unique object, including a unique *set* of phones.
  - ❖ `name` in the relation does not determine a unique tuple.
  - ❖ Since tuples are not identical to objects, there is no inconsistency!

Decompose Relations?

One option is to get `phone` into a separate relation (with `name`). The database would look like:

name	street	city	zip
n_1	s_1	c_1	z_1

name	phone
n_1	p_1
n_1	p_2

- Advantages:
  1. Avoids redundancy in address components.
  2. Handles the case where someone has *no* phone.
- Disadvantage: Harder to answer queries that jump between two relations, e.g., "in what city is phone 650-725-4802?"

A Design Problem

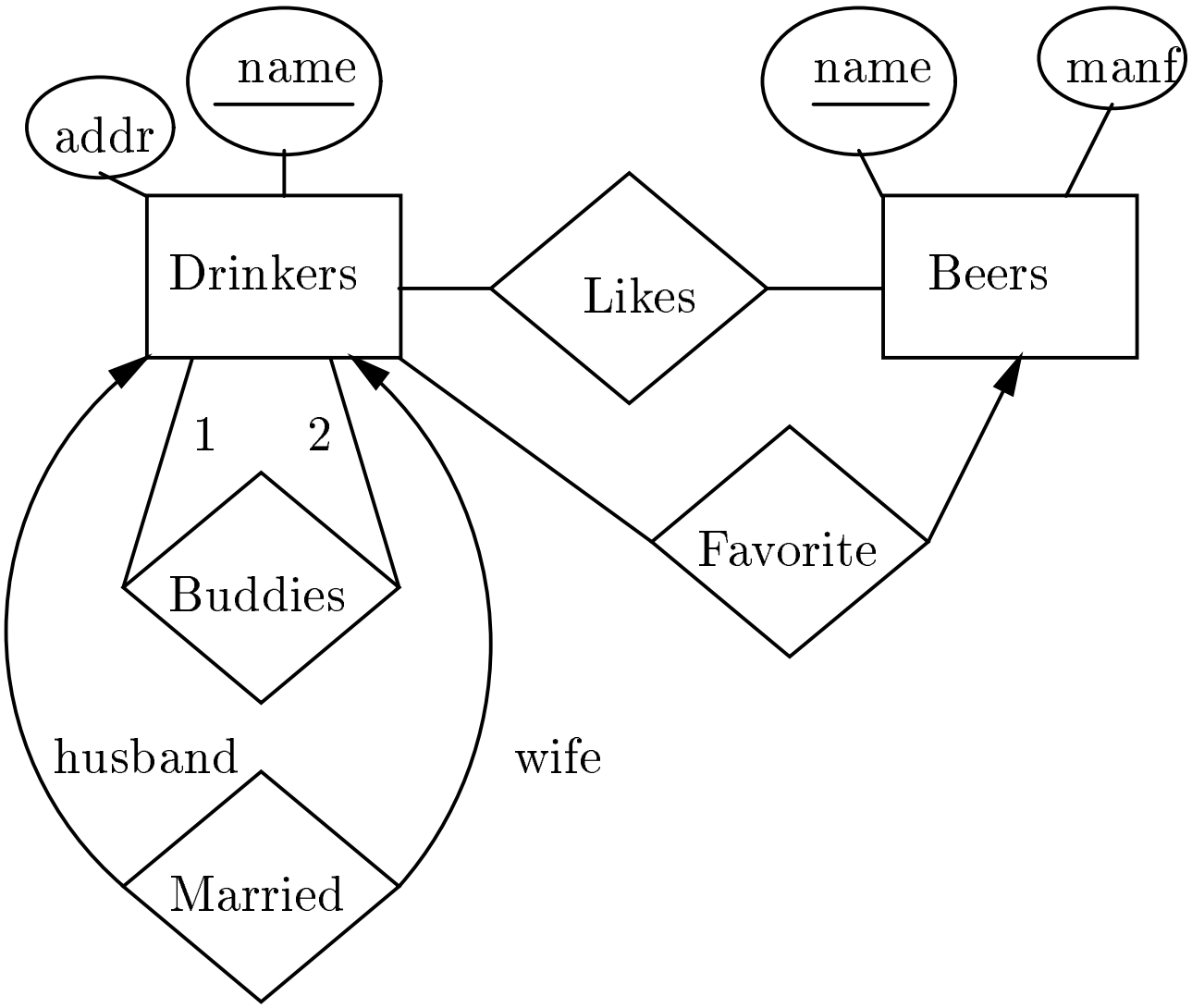
```
interface Family {
    attribute Set<string> parents;
    attribute Set<string> children;
}
```

1. What is the key?
2. How should we represent a family with two parents and three children?
3. Would you favor decomposition into several relations?

E/R Relationships → Relations

Relation has attribute for *key* attributes of each E.S. that participates in the relationship.

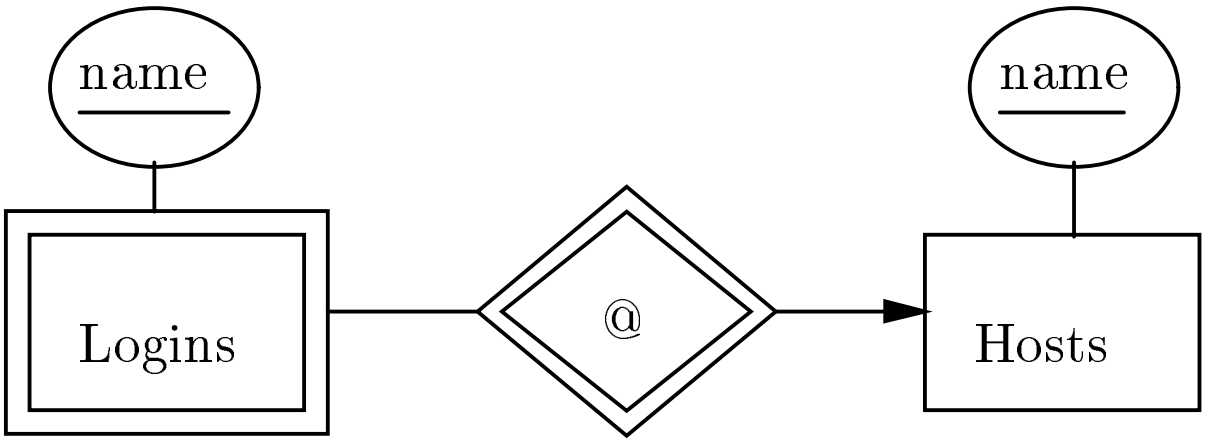
- Key of relation excludes attributes from the "one" side if relationship is many-one.
- For a one-one relationship, choose which side provides the key of the relation.
- Renaming attributes OK.
  - ❖ Essential if multiple roles for an E.S.



Likes(<u>drinker</u>, <u>beer</u>)
Favorite(<u>drinker</u>, beer)
Married(husband, <u>wife</u>)
Buddies(<u>name1</u>, <u>name2</u>)

- For one-one relation `Married`, we can choose either `husband` or `wife` as key.

Weak Entity Sets, Relationships → Relations



Hosts(hostName)
Logins(loginName, hostName)
At(loginName, hostName, hostName2)

- In `At`, `hostName` and `hostName2` must be the same host, so delete one of them.
- Then, `Logins` and `At` become the same relation; delete one of them.
- In this case, `Hosts`' schema is a subset of `Logins`' schema. Delete `Hosts`?
- General rule: Delete the relation that comes from a many-one relationship supporting a weak entity set.