Subqueries

- Used mainly in `FROM` clauses and with quantifiers `EXISTS` and `FORALL`.

Example: Subquery in `FROM`

Find the manufacturers of the beers served at Joe's.

```
SELECT DISTINCT b.manf
FROM (
        SELECT s.beer
        FROM Sells s
        WHERE s.bar.name = "Joe's Bar"
    ) b
```

Quantifiers

- Boolean-valued expressions for use in `WHERE`-clauses.

  `FOR ALL` x `IN` <collection> `:`
  <condition>

  `EXISTS` x `IN` <collection> `:`
  <condition>

- The expression has value `TRUE` if the condition is true for all (resp. at least one) elements of the collection.

Example

Find all bars that sell some beer for more than \$5.

```
SELECT b.name
FROM Bars b
WHERE EXISTS s IN b.beersSold :
    s.price > 5.00
```

Problem

How would you find the bars that *only* sold beers for more than \$5?

Example

Find the bars such that the only beers they sell for more than $5 are manufactured by Pete's.

```
SELECT b.name
FROM Bars b
WHERE FOR ALL be IN (
            SELECT s.beer
            FROM b.beersSold s
            WHERE s.price > 5.00
        ) :
    be.manf = "Pete's"
```

Extraction of Collection Elements

a) A collection with a single member: Extract the member with `ELEMENT`.

Example

Find the price Joe charges for Bud and put the result in a variable p.

```
p = ELEMENT(
    SELECT s.price
    FROM Sells s
    WHERE s.bar.name = "Joe's Bar"
        AND s.beer.name = "Bud"
)
```

b) Extracting all elements of a collection, one at a time:

1. Turn the collection into a list.
2. Extract elements of a list with <list name>[i].

Example

Print Joe's menu, in order of price, with beers of the same price listed alphabetically.

```
L =
    SELECT s.beer.name, s.price
    FROM Sells s
    WHERE s.bar.name = "Joe's Bar"
    ORDER BY s.price, s.beer.name;

printf("Beer\tPrice\n\n");
for(i=1; i<=COUNT(L); i++)
    printf("%s\t%f\n",
        L[i].name,
        L[i].price
    );
```

Aggregation

The five operators avg, min, max, sum, count apply to any collection, as long as the operators make sense for the element type.

Example

Find the average price of beer at Joe's.

```
x = AVG(
    SELECT s.price
    FROM Sells s
    WHERE s.bar.name = "Joe's Bar"
);
```

- Note coersion: result of `SELECT` is technically a bag of 1-field structs, which is identified with the bag of the values of that field.

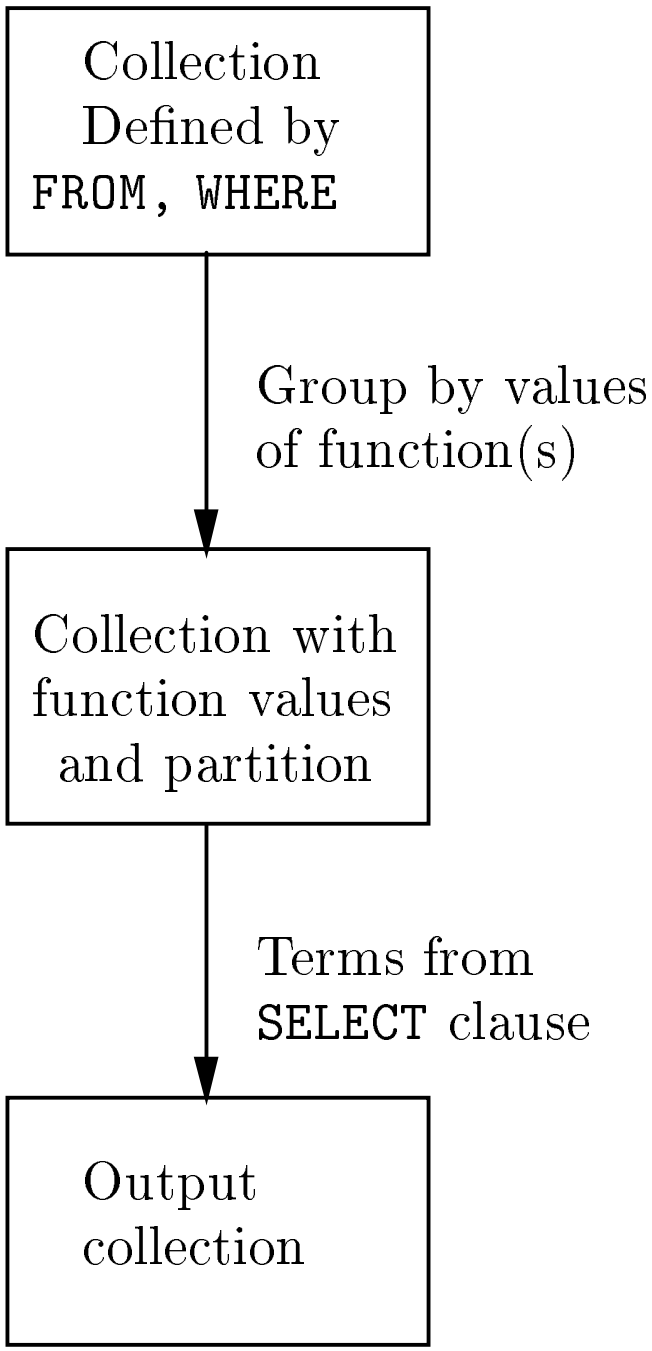
Grouping

Recall SQL grouping, for example:

```
SELECT bar, AVG(price)
FROM Sells
GROUP BY bar;
```

- Is the `bar` value the "name" of the group, or the common value for the `bar` component of all tuples in the group?
- In SQL it doesn't matter, but in OQL, you can create groups from the values of any function(s), not just attributes.
  - ❖ Thus, groups are identified by common values, not "name."
  - ❖ Example: group by first letter of bar names (method needed).

Outline of OQL Group-By



Example

Find the average price of beer at each bar.

```
SELECT barName, avgPrice: AVG(
        SELECT p.s.price
        FROM partition p
    )
FROM Sells s
GROUP BY barName: s.bar.name
```

1. Initial collection = `Sells`.

   - But technically, it is a bag of structs of the form

     `Struct(s:` $s1$`)`

     Where $s1$ is a `Sell` object. Note, the lone field is named `s`; in general, there are fields for all of the "typical objects" in the `FROM` clause.

2. Intermediate collection:

   - One function: `s.bar.name` maps `Sell` objects s to the value of the name of the bar referred to by s.
   - Collection is a set of structs of type:

```
Struct{barName: string,
    partition: Set<
        Struct{s: Sell}
    >
}
```

     For example:

```
Struct(barName = "Joe's Bar",
    partition = {s_1, ..., s_n})
```

     where $s_1, \ldots, s_n$ are all the structs with one field, named `s`, whose value is one of the `Sell` objects that represent Joe's Bar selling some beer.

3. Output collection: consists of beer-average price pairs, one for each struct in the intermediate collection.

   - Type of structures in the output:

```
Struct{barName: string,
    avgPrice: real}
```

   - Note that in the subquery of the `SELECT` clause:

```
SELECT barName, avgPrice: AVG(
        SELECT p.s.price
        FROM partition p
    )
```

     We let p range over all structs in `partition`. Each of these structs contains a single field named `s` and has a `Sell` object as its value. Thus, `p.s.price` extracts the price from one of the `Sell` objects.

   - Typical output struct:

```
Struct(barName = "Joe's Bar",
    avgPrice = 2.83)
```

Another, Less Typical Example

Find, for each beer, the number of bars that charge a "low" price (≤ 2.00) and a "high" price (≥ 4.00) for that beer.

- Strategy: group by three things:
  1. The beer name,
  2. A boolean function that is true iff the price is low.
  3. A boolean function that is true iff the price is high.

The Query

```
SELECT beerName, low, high,
    count: COUNT(partition)
FROM Beers b, b.soldBy s
GROUP BY beerName: b.name,
    low: s.price <= 2.00,
    high: s.price >= 4.00
```

1. Initial collection: Pairs (b, s), where b is a `Beer` object, and s is a `Sell` object representing the sale of that beer at some bar.

   - Type of collection members:

     ```
     Struct{b: Beer, s: Sell}
     ```

2. Intermediate collection: Quadruples consisting of a beer name, booleans telling whether this group is for high, low, or neither prices for that beer, and the partition for that group.

   ❖ The partition is a set of structs of the type:

   `Struct{b: Beer, s: Sell}`

   A typical value:

   `Struct(b: "Bud"` object,
   `s:` a `Sell` object involving Bud)

- Type of quadruples in the intermediate collection:

```
Struct{
    beerName: string,
    low: boolean,
    high: boolean,
    partition: Set<Struct{
        b: Beer,
        s: Sell
    }>
}
```

Typical structs in intermediate collection:

beerName	low	high	partition
Bud	TRUE	FALSE	S_{low}
Bud	FALSE	TRUE	S_{high}
Bud	FALSE	FALSE	S_{mid}
...	...	...	...

where S_{low} S_{high}, and S_{mid} are the sets of beer-sells pairs (b, s) where the beer is Bud and s has, respectively, a low ($\leq$ 2.00), high ($\geq$ 4.00) and medium (between 2.00 and 4.00) price.

- Note the partition with `low` = `high` = `TRUE` must be empty and will not appear.

3. Output collection: The first three components of each group's struct are copied to the output, and the last (`partition`) is counted. The result:

beerName	low	high	count
Bud	TRUE	FALSE	27
Bud	FALSE	TRUE	14
Bud	FALSE	FALSE	36
...	...	...	...