CS 317/387

Structured Query Language (SQL)

What is SQL?

- SQL = Structured Query Language (often pronounced as "sequel."
- SQL is the primary mechanism for *defining*, *querying* and *modifying* the data in an RDB.
- SQL is declarative:
  - Say what you want to accomplish, without specifying how.
  - One of the main reasons for the commercial success of RDMBSs.
- SQL has many standards and implementations:
  - ANSI SQL
  - SQL-92/SQL2 (null operations, outerjoins)
  - SQL-99/SQL3 (recusion, triggers, objects)
  - Vendor-specific variations.

Why SQL?

- SQL is a very-high-level language.
  - Say "what to do" rather than "how to do it."
  - Avoid a lot of data-manipulation details needed in procedural languages like C++ or Java.
- Database management system figures out "best" way to execute query.
  - Called "query optimization."

Our Running Example

- All our SQL queries will be based on the following database schema.

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

Sells(bar, beer, price)

Frequents(drinker, bar)

Another Schema

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-type)

account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

Select-From-Where

SELECT desired attributes

FROM one or more tables

WHERE condition about tuples of the tables

Example

- Using Beers(name, manf), what beers are made by Anheuser-Busch?

```
SELECT name
FROM Beers
WHERE manf = 'Anheuser-Busch';
```

Notice SQL uses single-quotes for strings.
SQL is *case-insensitive*, except inside strings.

Result of Query

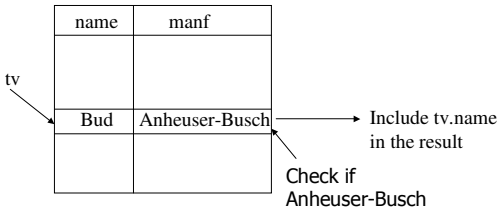
name
Bud
Bud Lite
Michelob
. . .

The answer is a relation with a single attribute - name, and tuples with the values for that attribute - in our case, name of each beer by Anheuser-Busch, such as Bud, Michelob etc. .

Meaning of Single-Relation Query

- Begin with the relation in the FROM clause.
- Apply the selection indicated by the WHERE clause.
- Apply the extended projection indicated by the SELECT clause.

Operational Semantics



Operational Semantics

- To implement this algorithm think of a *tuple variable* ranging over each tuple of the relation mentioned in FROM.
- Check if the "current" tuple satisfies the WHERE clause.
- If so, compute the attributes or expressions of the SELECT clause using the components of this tuple.

* in SELECT clauses

- When there is one relation in the FROM clause, * in the SELECT clause stands for "all attributes of this relation."
- Example using Beers(name, manf):

```
SELECT * FROM Beers
    WHERE manf = 'Anheuser-Busch';
```

Result of Query:

name	manf
Bud	Anheuser-Busch
Bud Lite	Anheuser-Busch
Michelob	Anheuser-Busch
. . .	. . .

Now, the result has each of the attributes of Beers.

Renaming Attributes

- If you want the result to have different attribute names, use "AS <new name>" to rename an attribute.
- Example based on Beers(name, manf):

```
SELECT name AS beer, manf
FROM Beers
WHERE manf = 'Anheuser-Busch'
```

Result of Query:

beer	manf
Bud	Anheuser-Busch
Bud Lite	Anheuser-Busch
Michelob	Anheuser-Busch
. . .	. . .

Expressions in SELECT Clauses

- Any expression that is syntactically correct can appear as an element of a SELECT clause.
- Example: from Sells(bar, beer, price):

```
SELECT bar, beer, price * 40 AS
priceInRupees
    FROM Sells;
```

Result of Query

bar	beer	priceInRupees
Joe's	Bud	80
Sue's	Miller	120
…	…	…

Constant Expressions

- From Likes(drinker, beer) :

```
SELECT drinker, 'likes Bud' AS
                    whoLikesBud
FROM Likes
WHERE beer = 'Bud';
```

Result of Query

drinker	whoLikesBud
Sally	likes Bud
Fred	likes Bud
…	…

Complex Conditions in WHERE Clause

- From Sells(bar, beer, price), *find the price Joe's Bar charges for Bud:*

Notice how we get a single-quote in strings.

```
SELECT price
FROM Sells
WHERE bar = 'Joe''s Bar' AND
    beer = 'Bud';
```

Patterns

- *WHERE* clauses can have conditions in which a string is compared with a pattern, to see if it matches.
- Syntax: **<Attribute> LIKE <pattern>** or **<Attribute> NOT LIKE <pattern>**
- Pattern is a quoted string with:
  - % = "any string"
  - _ = "any character"

Example

- From Drinkers(name, addr, phone) find the drinkers who fall within exchange 555:

```
SELECT name
FROM Drinkers
WHERE phone LIKE '%555-_ _ _ _';
```

The "between" operator

- SQL includes a **between** comparison operator
- E.g. Find the loan number of those loans with loan amounts between Rs90,000 and Rs100,000 (that is, ≥ Rs.90,000 and ≤ Rs. 100,000)

```
select loan-number
   from loan
   where amount between 90000 and 100000
```

NULL Values

- Tuples in SQL relations can have NULL as a value for one or more components.
- Meaning depends on context. Two common cases:
- ***Missing value*** : e.g., we know Joe's Bar has some address, but we don't know what it is.
- ***Inapplicable*** : e.g., the value of attribute *spouse* for an unmarried person.

Comparing NULL's to Values

- The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- When any value is compared with NULL, the truth value is UNKNOWN.
- But a query only produces a tuple in the answer if its truth value for the WHERE clause is TRUE (not FALSE or UNKNOWN).

Three-Valued Logic

- To understand how AND, OR, and NOT work in 3-valued logic, think of TRUE = 1, FALSE = 0, and UNKNOWN = ½.
- AND = MIN; OR = MAX, NOT(x) = 1-x.
- Example:

TRUE AND (FALSE OR NOT(UNKNOWN)) =
MIN(1, MAX(0, (1 - ½))) =
MIN(1, MAX(0, ½) = MIN(1, ½) = ½.

Surprising Example

- From the following Sells relation:

bar	beer	price
Joe's bar	Bud	NULL

```
SELECT bar
FROM Sells
WHERE price < 2.00 OR price >= 2.00;
```

UNKNOWN UNKNOWN

UNKNOWN

Reason: 2-Valued Laws != 3-Valued Laws

- Some common laws, like commutativity of AND, hold in 3-valued logic.
- But not others, e.g., the "law of the excluded middle": *p* OR NOT *p* = TRUE.
- When *p* = UNKNOWN, the left side is MAX(½, (1 – ½)) = ½ != 1.

More on Null

- The predicate **is null** can be used to check for null values.
  - E.g. Find all loan number which appear in the *loan* relation with null values for *amount*.

```
select loan-number
from loan
where amount is null
```

- The result of any arithmetic expression involving *null* is *null*
  - E.g. 5 + null returns null
- However, aggregate functions simply ignore nulls

Ordering the output

- List in alphabetic order the names of all customers having a loan in Powai branch
-

```
select distinct customer-name
    from    borrower, loan
    where borrower loan-number = loan.loan-number and
                branch-name = 'Powai'
    order by customer-name
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - E.g. **order by** *customer-name* **desc**

Multi-relation Queries

- Interesting queries often combine data from more than one relation.
- We can address several relations in one query by listing them all in the FROM clause.
- Distinguish attributes of the same name by "<relation>.<attribute>"

Example

- Using relations Likes(drinker, beer) and Frequents(drinker, bar), find the beers liked by at least one person who frequents Joe's Bar.

```
SELECT beer
FROM Likes, Frequents
WHERE bar = 'Joe''s Bar' AND
    Frequents.drinker = Likes.drinker;
```

Formal Semantics

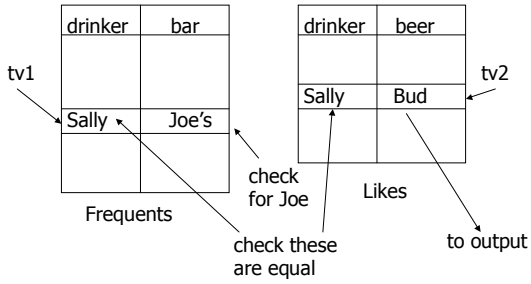
- Almost the same as for single-relation queries:

1. Start with the product of all the relations in the FROM clause.
2. Apply the selection condition from the WHERE clause.
3. Project onto the list of attributes and expressions in the SELECT clause.

Operational Semantics

- Imagine one tuple-variable for each relation in the FROM clause.
  - These tuple-variables visit each combination of tuples, one from each relation.
- If the tuple-variables are pointing to tuples that satisfy the WHERE clause, send these tuples to the SELECT clause.

Example



Explicit Tuple-Variables

- Tuple variables are defined in the **from** clause
- Case 1: Sometimes, a query needs to use two copies of the same relation.
- Distinguish copies by following the relation name by the name of a tuple-variable, in the FROM clause.
- It's always an option to rename relations this way, even when not essential.

Example

- From Beers(name, manf), find all pairs of beers by the same manufacturer.
  - Do not produce pairs like (Bud, Bud).
  - Produce pairs in alphabetic order, e.g. (Bud, Miller), not (Miller, Bud).

```
SELECT b1.name, b2.name
    FROM Beers b1, Beers b2
    WHERE b1.manf = b2.manf AND
        b1.name < b2.name;
```

Tuple Variables

- Case 2: via the use of the **as** clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

```
select customer-name, T.loan-number, S.amount
        from borrower as T, loan as S
        where  T.loan-number = S.loan-number
```

Tuple variable …

- Find the names of all branches that have greater assets than some (any ?) branch located in Delhi.

```
select distinct T.branch-name
  from branch as T, branch as S
  where T.assets > S.assets and S.branch-
city = 'Delhi'
```

Union, Intersection, and Difference

- Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
  - (subquery) UNION (subquery)
  - (subquery) INTERSECT (subquery)
  - (subquery) EXCEPT (subquery)

Set Operations

- Find all customers who have a loan, an account, or both:

```
(select customer-name from depositor)
  union
(select customer-name from borrower)
```

- Find all customers who have both a loan and an account.

```
(select customer-name from depositor)
  intersect
(select customer-name from borrower)
```

- Find all customers who have an account but no loan.

```
(select customer-name from depositor)
  except
(select customer-name from borrower)
```

Example

- From relations Likes(drinker, beer), Sells(bar, beer, price), and Frequents(drinker, bar), find the drinkers and beers such that:
  1. The drinker likes the beer, and
  2. The drinker frequents at least one bar that sells the beer.

Solution

```
(SELECT * FROM Likes)
  INTERSECT
(SELECT drinker, beer
 FROM Sells, Frequents
 WHERE Frequents.bar = Sells.bar
);
```

The drinker frequents a bar that sells the beer.

Bag Semantics

- Although the SELECT-FROM-WHERE statement uses bag semantics, the default for union, intersection, and difference is **set** semantics.
  - That is, duplicates are eliminated as the operation is applied.

Motivation: Efficiency

- When doing projection, it is easier to avoid eliminating duplicates.
  - Just work tuple-at-a-time.
- For intersection or difference, it is most efficient to sort the relations first.
  - At that point you may as well eliminate the duplicates anyway.

Controlling Duplicate Elimination

- Force the result to be a set by SELECT DISTINCT . . .
- Force the result to be a bag (i.e., don't eliminate duplicates) by ALL, as in . . . UNION ALL . . .

Example: DISTINCT

- From Sells(bar, beer, price), find all the different prices charged for beers:

```
Select DISTINCT price
FROM Sells;
```

- Notice that without DISTINCT, each price would be listed as many times as there were bar/beer pairs at that price.

Example: ALL

- Using relations Frequents(drinker, bar) and Likes(drinker, beer) - List drinkers who frequent more bars than they like beers.

```
(SELECT drinker FROM Frequents)
        EXCEPT ALL
(SELECT drinker FROM Likes);
```

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

Aggregate Functions (Cont.)

- Find the average account balance at the Powai branch.

```
select avg (balance)
        from account
        where branch-name = 'Powai'
```

- Find the number of tuples in the *customer* relation.

```
select count (*) from customer
```

- Find the number of depositors in the bank.

```
select count (distinct customer-name)
        from depositor
```

Aggregate Functions – Group By

- Partitions table (result) on given attributes; we can retrieve some aggregate value for each group
- Find the number of depositors for <u>each</u> branch.

```
select branch-name, count (distinct customer-name)
      from depositor, account
where depositor.account-number = account.account-number
      group by branch-name
```

Example: Grouping

- From Sells(bar, beer, price), find the average price for each beer:

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

Example: Grouping

- From Sells(bar, beer, price) and Frequents(drinker, bar), find for each drinker the average price of Bud at the bars they frequent:

```
SELECT drinker, AVG(price)
FROM Frequents, Sells
WHERE beer = 'Bud' AND
      Frequents.bar = Sells.bar
GROUP BY drinker;
```

Compute drinker-bar-price for Bud tuples first, then group by drinker.

Restriction on SELECT Lists With Aggregation

- If any aggregation is used, then each element of the SELECT list must be either:
  1. Aggregated, or
  2. An attribute on the GROUP BY list.

Illegal Query Example

- You might think you could find the bar that sells Bud the cheapest by:

```
SELECT bar, MIN(price)
FROM Sells
WHERE beer = 'Bud';
```

- But this query is illegal in SQL because
- Bar is not aggregated or an attribute on a group by list.

Aggregate Functions – Having Clause

- Allows us to select a partition based on some grouping or aggregate value
- Example: Find the names of all branches where the average account balance is more than 12000.

```
select branch-name, avg (balance)
     from account
     group by branch-name
     having avg (balance) > 1200
```

Example: HAVING

- From Sells(bar, beer, price) and Beers(name, manf), find the average price of those beers that are either served in at least three bars or are manufactured by Pete's.

Solution

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
HAVING COUNT(bar) >= 3 OR
    beer IN (SELECT name
             FROM Beers
             WHERE mfr = 'Pete's');
```

Beer groups with at least 3 non-NULL bars and also beer groups where the manufacturer is Pete's.

Beers manufactured by Pete's.

Requirements on HAVING Conditions

- These conditions may refer to any relation or tuple-variable in the FROM clause.
- They may refer to attributes of those relations, as long as the attribute makes sense within a group; i.e., it is either:
  1. A grouping attribute, or
  2. Aggregated.

Subqueries

- A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including FROM and WHERE clauses.
- Example: in place of a relation in the FROM clause, we can place another query, and then query its result.
  - Better use a tuple-variable to name tuples of the result.

Subqueries That Return One Tuple

- If a subquery is guaranteed to produce one tuple, then the subquery can be used as a value.
  - Usually, the tuple has one component.
  - A run-time error occurs if there is no tuple or more than one tuple.

Example

- From Sells(bar, beer, price), find the bars that serve Miller for the same price Joe charges for Bud.
- Two queries would surely work:
  1. Find the price Joe charges for Bud.
  2. Find the bars that serve Miller at that price.

Query + Subquery Solution

```
SELECT bar
FROM Sells
WHERE beer = 'Miller' AND
   price = (SELECT price
    FROM Sells
    WHERE bar = 'Joe''s Bar' AND
    beer = 'Bud');
```

The price at which Joe sells Bud

The IN Operator

- **<tuple> IN <relation>** is true if and only if the tuple is a member of the relation.
  - **<tuple> NOT IN <relation>** means the opposite.
- IN-expressions can appear in WHERE clauses.
- The <relation> is often a subquery.

Example

- From Beers(name, manf) and Likes(drinker, beer), find the name and manufacturer of each beer that Fred likes.

```
SELECT * FROM Beers
        WHERE name IN
            (SELECT beer
                 FROM Likes
             WHERE drinker = Fred');
```

The set of beers Fred likes

The Exists Operator

- **EXISTS(<relation>)** is true if and only if the <relation> is not empty.
- Example: From Beers(name, manf) , find those beers that are the unique beer by their manufacturer.

Example Query with EXISTS

```
SELECT name
FROM Beers b1
WHERE NOT EXISTS(
    SELECT *
    FROM Beers
    WHERE manf = b1.manf AND
          name <> b1.name);
```

Notice scope rule: manf refers to closest nested FROM with a relation having that attribute.

Notice the SQL "not equals" operator

Set of beers with the same manf as b1, but not the same beer

The Operator ANY

- ***x*** **= ANY(<relation>)** is a boolean condition true if *x* equals at least one tuple in the relation.
- Similarly, = can be replaced by any of the comparison operators.
- Example: ***x*** **>= ANY(<relation>)** means *x* is not the smallest tuple in the relation.
  - Note tuples must have one component only.

Set Comparison with SOME clause

- Find all branches that have greater assets than some branch located in Delhi.

```
select distinct T.branch-name
        from branch as T, branch as S
        where T.assets > S.assets and
              S.branch-city = 'Delhi'
```

- **Same query using > some clause**

```
select branch-name
      from branch
      where assets > some
            (select assets
             from branch
             where branch-city = 'Delhi')
```

Definition of some Clause

- F <comp-op> **some** r $\Leftrightarrow$ $\exists t \in r$ *s.t.* (F <comp-op> t)
  Where <comp-op> can be: $<, \leq, >, =, \neq$

(5< **some** | 0 / 5 / 6 |) = true
(read: 5 < some tuple in the relation)

(5< **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

The Operator ALL

- Similarly, ***x*** **<> ALL(<relation>)** is true if and only if for every tuple t in the relation, x is not equal to t.
  - That is, x is not a member of the relation.
- The **<>** can be replaced by any comparison operator.
- Example: ***x*** **>= ALL(<relation>)** means there is no tuple larger than x in the relation.

Example

- From Sells(bar, beer, price), find the beer(s) sold for the highest price.

```
SELECT beer  FROM Sells
        WHERE price >=
    ALL(SELECT price
        FROM Sells);
```

price from the outer Sells must not be less than any price.

Join Expressions

- SQL provides several versions of (bag) joins.
- These expressions can be stand-alone queries or used in place of relations in a FROM clause.

Products and Natural Joins

- Natural join:
  R NATURAL JOIN S;
- Product:
  R CROSS JOIN S;
- Example:
  `Likes NATURAL JOIN Serves;`
- Relations can be parenthesized subqueries, as well.

Theta Join

- R JOIN S ON <condition>
- Example: using Drinkers(name, addr) and Frequents(drinker, bar):

```
Drinkers JOIN Frequents ON
    name = drinker;
```

gives us all (*d, a, d, b*) quadruples such that drinker *d* lives at address *a* and frequents bar *b*.

Outerjoins

- R OUTER JOIN S is the core of an outerjoin expression. It is modified by:
  1. Optional NATURAL in front of OUTER.
  2. Optional ON <condition> after JOIN.
  3. Optional LEFT, RIGHT, or FULL before OUTER.
     - LEFT = pad dangling tuples of R only.
     - RIGHT = pad dangling tuples of S only.
     - FULL = pad both; this choice is the default.

More SQL

Database Modification
Defining a Database Schema
Views

Database Modifications

- A *modification* command does not return a result (as a query does), but changes the database in some way.
- Three kinds of modifications:
  1. *Insert* a tuple or tuples.
  2. *Delete* a tuple or tuples.
  3. *Update* the value(s) of an existing tuple or tuples.

Insertion

- To insert a single tuple:

```
INSERT INTO <relation>
VALUES ( <list of values> );
```

- Example: add to Likes(drinker, beer) the fact that Sally likes Bud.

```
INSERT INTO Likes
VALUES('Sally', 'Bud');
```

Specifying Attributes in INSERT

- We may add to the relation name a list of attributes.
- Two reasons to do so:

1. We forget the standard order of attributes for the relation.
2. We don't have values for all attributes, and we want the system to fill in missing components with NULL or a default value.

Example: Specifying Attributes

- Another way to add the fact that Sally likes Bud to Likes(drinker, beer):

```
INSERT INTO Likes(beer, drinker)
VALUES('Bud', 'Sally');
```

Inserting Many Tuples

- We may insert the entire result of a query into a relation, using the form:

```
INSERT INTO <relation> ( <subquery> );
```

Example: Insert a Subquery

- Using Frequents(drinker, bar), enter into the new relation PotBuddies(name) all of Sally's "potential buddies," i.e., those drinkers who frequent at least one bar that Sally also frequents.

Solution

The other drinker

Pairs of Drinker tuples where the first is for Sally, the second is for someone else, and the bars are the same.

```
INSERT INTO PotBuddies
( SELECT d2.drinker
  FROM Frequents d1, Frequents d2
  WHERE d1.drinker = 'Sally' AND
    d2.drinker <> 'Sally' AND
    d1.bar = d2.bar
);
```

Deletion

- To delete tuples satisfying a condition from some relation:

```
DELETE FROM <relation>
WHERE <condition>;
```

Example: Deletion

- Delete from Likes(drinker, beer) the fact that Sally likes Bud:

```
DELETE FROM Likes
WHERE drinker = 'Sally' AND
      beer = 'Bud';
```

Example: Delete all Tuples

- Make the relation **Likes** empty:

```
DELETE FROM Likes;
```

- Note no WHERE clause needed.

Example: Delete Many Tuples

- Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b
WHERE EXISTS (
    SELECT name FROM Beers
    WHERE manf = b.manf AND
          name <> b.name);
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b.

Semantics of Deletion --- (1)

- Suppose Anheuser-Busch makes only Bud and Bud Lite.
- Suppose we come to the tuple *b* for Bud first.
- The subquery is nonempty, because of the Bud Lite tuple, so we delete Bud.
- Now, when *b* is the tuple for Bud Lite, do we delete that tuple too?

Semantics of Deletion --- (2)

- Answer: we *do* delete Bud Lite as well.
- The reason is that deletion proceeds in two stages:
  1. Mark all tuples for which the WHERE condition is satisfied.
  2. Delete the marked tuples.

Updates

- To change certain attributes in certain tuples of a relation:

```
UPDATE <relation>
SET <list of attribute assignments>
WHERE <condition on tuples>;
```

Example: Update

- Change drinker Fred's phone number to 555-1212:

```
UPDATE Drinkers
SET phone = '555-1212'
WHERE name = 'Fred';
```

Example: Update Several Tuples

- Make $4 the maximum price for beer:

```
UPDATE Sells
SET price = 4.00
WHERE price > 4.00;
```

Defining a Database Schema

- A *database schema* comprises declarations for the relations ("tables") of the database.
- Several other kinds of elements also may appear in the database schema, including views, indexes, and triggers, which we'll introduce later.

Creating (Declaring) a Relation

- Simplest form is:

```
CREATE TABLE <name> (
    <list of elements>
);
```

- To delete a relation:

```
DROP TABLE <name>;
```

Elements of Table Declarations

- Most basic element: an attribute and its type.
- The most common types are:
  - `INT` or `INTEGER` (synonyms).
  - `REAL` or `FLOAT` (synonyms).
  - `CHAR(`*n*`)` = fixed-length string of *n* characters.
  - `VARCHAR(`*n*`)` = variable-length string of up to *n* characters.

Example: Create Table

```
CREATE TABLE Sells (
    bar     CHAR(20),
    beer    VARCHAR(20),
    price   REAL
);
```

Dates and Times

- DATE and TIME are types in SQL.
- The form of a date value is:
  DATE 'yyyy-mm-dd'
  - Example: `DATE '2004-09-30'` for Sept. 30, 2004.

Times as Values

- The form of a time value is:
  TIME 'hh:mm:ss'

with an optional decimal point and fractions of a second following.

- Example: `TIME '15:30:02.5'` = two and a half seconds after 3:30PM.

Declaring Keys

- An attribute or list of attributes may be declared PRIMARY KEY or UNIQUE.
- Either says the attribute(s) so declared functionally determine all the attributes of the relation schema.
- There are a few distinctions to be mentioned later.

Declaring Single-Attribute Keys

- Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- Example:

```
CREATE TABLE Beers (
     name CHAR(20) UNIQUE,
     manf CHAR(20)
);
```

Declaring Multiattribute Keys

- A key declaration can also be another element in the list of elements of a CREATE TABLE statement.
- This form is essential if the key consists of more than one attribute.
  - May be used even for one-attribute keys.

Example: Multiattribute Key

- The bar and beer together are the key for Sells:

```
CREATE TABLE Sells (
    bar     CHAR(20),
    beer    VARCHAR(20),
    price   REAL,
    PRIMARY KEY (bar, beer)
);
```

PRIMARY KEY Versus UNIQUE

- The SQL standard allows DBMS implementers to make their own distinctions between PRIMARY KEY and UNIQUE.
  - Example: some DBMS might automatically create an *index* (data structure to speed search) in response to PRIMARY KEY, but not UNIQUE.

Required Distinctions

- However, standard SQL requires these distinctions:

1. There can be only one PRIMARY KEY for a relation, but several UNIQUE attributes.
2. No attribute of a PRIMARY KEY can ever be NULL in any tuple. But attributes declared UNIQUE may have NULL's, and there may be several tuples with NULL.

Some Other Declarations for Attributes

- NOT NULL means that the value for this attribute may never be NULL.
- DEFAULT <value> says that if there is no specific value known for this attribute's component in some tuple, use the stated <value>.

Example: Default Values

```
CREATE TABLE Drinkers (
   name CHAR(30) PRIMARY KEY,
   addr CHAR(50)
        DEFAULT '123 Sesame St.',
   phone CHAR(16)
);
```

Effect of Defaults --- (1)

- Suppose we insert the fact that Sally is a drinker, but we know neither her address nor her phone.
- An INSERT with a partial list of attributes makes the insertion possible:

```
INSERT INTO Drinkers(name)
VALUES('Sally');
```

Effect of Defaults --- (2)

- But what tuple appears in Drinkers?

name	addr	phone
Sally	123 Sesame St	NULL

- If we had declared phone NOT NULL, this insertion would have been rejected.

Adding Attributes

- We may add a new attribute ("column") to a relation schema by:

  ALTER TABLE <name> ADD
  <attribute declaration>;

- Example:

```
ALTER TABLE Bars ADD
phone CHAR(16)DEFAULT 'unlisted';
```

Deleting Attributes

- Remove an attribute from a relation schema by:

```
ALTER TABLE <name>
    DROP <attribute>;
```

- Example: we don't really need the license attribute for bars:

```
ALTER TABLE Bars DROP license;
```

Views

- A *view* is a "virtual table" = a relation defined in terms of the contents of other tables and views.
- Declare by:

```
CREATE VIEW <name> AS <query>;
```

- Antonym: a relation whose value is really stored in the database is called a *base table*.

Example: View Definition

- CanDrink(drinker, beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
   SELECT drinker, beer
   FROM Frequents, Sells
   WHERE Frequents.bar = Sells.bar;
```

Example: Accessing a View

- Query a view as if it were a base table.
  - Also: a limited ability to modify views if it makes sense as a modification of one underlying base table.
- Example query:

```
SELECT beer FROM CanDrink
WHERE drinker = 'Sally';
```

What happens when a View is used?

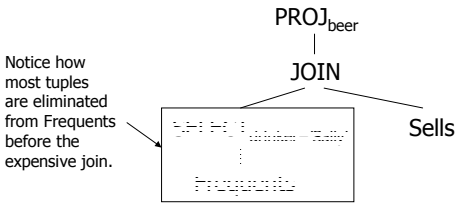
- The DBMS starts by interpreting the query as if the view were a base table.
  - Typical DBMS turns the query into something like relational algebra.
- The definitions of any views used by the query are also replaced by their algebraic equivalents, and "spliced into" the expression tree for the query.

Example: View Expansion

DMBS Optimization

- It is interesting to observe that the typical DBMS will then "optimize" the query by transforming the algebraic expression to one that can be executed faster.
- Key optimizations:
  1. Push selections down the tree.
  2. Eliminate unnecessary projections.

Example: Optimization



Constraints

Foreign Keys
Local and Global Constraints
Triggers

Constraints and Triggers

- A *constraint* is a relationship among data elements that the DBMS is required to enforce.
  - Example: key constraints.
- *Triggers* are only executed when a specified condition occurs, e.g., insertion of a tuple.
  - Easier to implement than complex constraints.

Kinds of Constraints

- Keys.
- Foreign-key, or referential-integrity.
- Value-based constraints.
  - Constrain values of a particular attribute.
- Tuple-based constraints.
  - Relationship among components.
- Assertions: any SQL boolean expression.

Foreign Keys

- Consider Relation Sells(bar, beer, price).
- We might expect that a beer value is a real beer --- something appearing in Beers.name .
- A constraint that requires a beer in Sells to be a beer in Beers is called a *foreign-key* constraint.

Expressing Foreign Keys

- Use the keyword REFERENCES, either:
  1. Within the declaration of an attribute (only for one-attribute keys).
  2. As an element of the schema:

  FOREIGN KEY (<list of attributes>)
  REFERENCES <relation> (<attributes>)
- Referenced attributes must be declared PRIMARY KEY or UNIQUE.

Example: With Attribute

```
CREATE TABLE Beers (
  name    CHAR(20) PRIMARY KEY,
  manf    CHAR(20) );

CREATE TABLE Sells (
  bar     CHAR(20),
  beer    CHAR(20) REFERENCES
     Beers(name),
  price   REAL );
```

Example: As Element

```
CREATE TABLE Beers (
  name    CHAR(20) PRIMARY KEY,
  manf    CHAR(20) );

CREATE TABLE Sells (
  bar     CHAR(20),
  beer    CHAR(20),
  price   REAL,
  FOREIGN KEY(beer) REFERENCES
     Beers(name));
```

Enforcing Foreign-Key Constraints

- If there is a foreign-key constraint from attributes of relation *R* to a key of relation *S*, two violations are possible:
  1. An insert or update to *R* introduces values not found in *S*.
  2. A deletion or update to S causes some tuples of *R* to "dangle."

Actions Taken --- (1)

- Suppose *R* = Sells, *S* = Beers.
- An insert or update to Sells that introduces a nonexistent beer ***must be rejected.***
- A deletion or update to Beers that removes a beer value found in some tuples of Sells can be handled in three ways

Actions Taken --- (2)

1. *Default* : Reject the modification.
2. *Cascade* : Make the same changes in Sells.
   - Deleted beer: delete Sells tuple.
   - Updated beer: change value in Sells.
3. *Set NULL* : Change the beer to NULL.

Example: Cascade

- Delete the Bud tuple from Beers:
  - Then delete all tuples from Sells that have beer = 'Bud'.
- Update the Bud tuple by changing 'Bud' to 'Budweiser':
  - Then change all Sells tuples with beer = 'Bud' so that beer = 'Budweiser'.

Example: Set NULL

- Delete the Bud tuple from Beers:
  - Change all tuples of Sells that have beer = 'Bud' to have beer = NULL.
- Update the Bud tuple by changing 'Bud' to 'Budweiser':
  - Same change.

Choosing a Policy

- When we declare a foreign key, we may choose policies SET NULL or CASCADE independently for deletions and updates.
- Follow the foreign-key declaration by:
  `ON [UPDATE, DELETE][SET NULL CASCADE]`
- Two such clauses may be used.
- Otherwise, the default (reject) is used.

Example

```
CREATE TABLE Sells (
  bar    CHAR(20),
  beer   CHAR(20),
  price  REAL,
  FOREIGN KEY(beer)
     REFERENCES Beers(name)
     ON DELETE SET NULL
     ON UPDATE CASCADE
);
```

Attribute-Based Checks

- Constraints on the value of a particular attribute.
- Add: CHECK(<condition>) to the declaration for the attribute.
- The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery.

Example

```
CREATE TABLE Sells (
  bar    CHAR(20),
  beer   CHAR(20) CHECK ( beer IN
         (SELECT name FROM Beers)),
  price  REAL CHECK ( price <= 5.00 )
);
```

Timing of Checks

- Attribute-based checks performed only when a value for that attribute is *inserted or updated*.
- Example: `CHECK (price <= 5.00)` checks every new price and rejects the modification (for that tuple) if the price is more than $5.
- Example: `CHECK (beer IN (SELECT name FROM Beers))` not checked if a beer is deleted from Beers (unlike foreign-keys).

Tuple-Based Checks

- CHECK (<condition>) may be added as a relation-schema element.
- The condition may refer to any attribute of the relation.
  - But any other attributes or relations require a subquery.
- Checked on insert or update only.

Example: Tuple-Based Check

- Only Joe's Bar can sell beer for more than $5:

```
CREATE TABLE Sells (
    bar     CHAR(20),
    beer    CHAR(20),
    price   REAL,
    CHECK (bar = 'Joe''s Bar' OR
                price <= 5.00)
);
```

Assertions

- These are database-schema elements, like relations or views.
- Defined by:

```
CREATE ASSERTION <name>
    CHECK ( <condition> );
```

- Condition may refer to any relation or attribute in the database schema.

Example: Assertion

- In Sells(bar, beer, price), no bar may charge an average of more than $5.

```
CREATE ASSERTION NoRipoffBars CHECK (
    NOT EXISTS (
        SELECT bar FROM Sells
        GROUP BY bar
        HAVING 5.00 < AVG(price)
    ));
```

Bars with an average price above $5

Example: Assertion

- In Drinkers(name, addr, phone) and Bars(name, addr, license), there cannot be more bars than drinkers.

```
CREATE ASSERTION FewBar CHECK (
  (SELECT COUNT(*) FROM Bars) <=
  (SELECT COUNT(*) FROM Drinkers)
);
```

Timing of Assertion Checks

- In principle, we must check every assertion after every modification to any relation of the database.
- A clever system can observe that only certain changes could cause a given assertion to be violated.
  - Example: No change to Beers can affect FewBar. Neither can an insertion to Drinkers.

Triggers: Motivation

- Assertions are powerful, but the DBMS often can't tell when they need to be checked.
- Attribute- and tuple-based checks are checked at known times, but are not powerful.
- Triggers let the user decide when to check for a powerful condition.

Event-Condition-Action Rules

- Another name for "trigger" is *ECA rule*, or *event-condition-action* rule.
- *Event* : typically a type of database modification, e.g., "insert on Sells."
- *Condition* : Any SQL boolean-valued expression.
- *Action* : Any SQL statements.

Preliminary Example: A Trigger

- Instead of using a foreign-key constraint and rejecting insertions into Sells(bar, beer, price) with unknown beers, a trigger can add that beer to Beers, with a NULL manufacturer.

Example: Trigger Definition

```
CREATE TRIGGER BeerTrig
  AFTER INSERT ON Sells
  REFERENCING NEW ROW AS NewTuple
  FOR EACH ROW
  WHEN (NewTuple.beer NOT IN
      (SELECT name FROM Beers))
  INSERT INTO Beers(name)
    VALUES(NewTuple.beer);
```

The event

The condition

The action

Options: CREATE TRIGGER

- CREATE TRIGGER <name>
- Option:

CREATE OR REPLACE TRIGGER <name>

- Useful if there is a trigger with that name and you want to modify the trigger.

Options: The Event

- AFTER can be BEFORE.
  - Also, INSTEAD OF, if the relation is a view.
    - A great way to execute view modifications: have triggers translate them to appropriate modifications on the base tables.
- INSERT can be DELETE or UPDATE.
  - And UPDATE can be UPDATE . . . ON a particular attribute.

Options: FOR EACH ROW

- Triggers are either "row-level" or "statement-level."
- FOR EACH ROW indicates row-level; its absence indicates statement-level.
- *Row level triggers* : execute once for each modified tuple.
- *Statement-level triggers* : execute once for an SQL statement, regardless of how many tuples are modified.

Options: REFERENCING

- INSERT statements imply a new tuple (for row-level) or new table (for statement-level).
  - The "table" is the set of inserted tuples.
- DELETE implies an old tuple or table.
- UPDATE implies both.
- Refer to these by

[NEW OLD][TUPLE TABLE] AS <name>

Options: The Condition

- Any boolean-valued condition is appropriate.
- It is evaluated before or after the triggering event, depending on whether BEFORE or AFTER is used in the event.
- Access the new/old tuple or set of tuples through the names declared in the REFERENCING clause.

Options: The Action

- There can be more than one SQL statement in the action.
  - Surround by BEGIN . . . END if there is more than one.
- But queries make no sense in an action, so we are really limited to modifications.

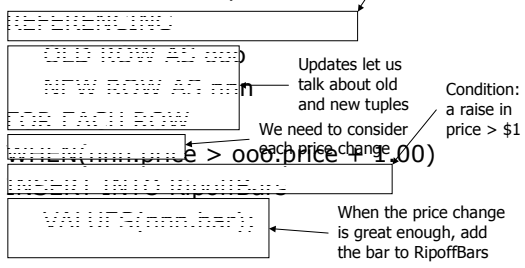
Another Example

- Using Sells(bar, beer, price) and a unary relation RipoffBars(bar) created for the purpose, maintain a list of bars that raise the price of any beer by more than $1.

The Trigger

CREATE TRIGGER PriceTrig
AFTER UPDATE OF price ON Sells

The event – only changes to prices



Triggers on Views

- Generally, it is impossible to modify a view, because it doesn't exist.
- But an INSTEAD OF trigger lets us interpret view modifications in a way that makes sense.
- Example: We'll design a view Synergy that has (drinker, beer, bar) triples such that the bar serves the beer, the drinker frequents the bar and likes the beer.

Example: The View

CREATE VIEW Synergy AS
  SELECT [illegible]

Pick one copy of each attribute

Natural join of Likes, Sells, and Frequents

Interpreting a View Insertion

- We cannot insert into Synergy --- it is a view.
- But we can use an INSTEAD OF trigger to turn a (drinker, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequents.
  - The Sells.price will have to be NULL.

The Trigger

```
CREATE TRIGGER ViewTrig
   INSTEAD OF INSERT ON Synergy
   REFERENCING NEW ROW AS n
   FOR EACH ROW
   BEGIN
        INSERT INTO LIKES VALUES(n.drinker, n.beer);
        INSERT INTO SELLS(bar, beer) VALUES(n.bar, n.beer);
        INSERT INTO FREQUENTS VALUES(n.drinker, n.bar);
   END;
```