Chapter 6. The Database Language SQL
The Database Language SQL

SQL: “Structured Query Language.”

SQL includes:
- Queries.
- Modifying the database.
- Declaring a database schema.

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.

The DBMS figures out the “best” way to execute queries.
SELECT desired attributes
FROM one or more tables
WHERE condition about tuples of the tables
Example

Given the relation

\[ \text{Movies} \ (\text{title, year, length, genre, studioName, producerC#}) \]

we want to know the titles of all movies produced by Disney Studios in 1990. In SQL, we say

```
SELECT title
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

An example of the query's result is

<table>
<thead>
<tr>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrettyWoman</td>
</tr>
<tr>
<td>Ants</td>
</tr>
</tbody>
</table>
Projection in SQL

Example

Given the relation

\[ \text{Movies} (\text{title}, \text{year}, \text{length}, \text{genre}, \text{studioName}, \text{producerC\#}), \]

we want to know everything about the movies produced by Disney Studios in 1990. In SQL, we say

```sql
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

An example of the query’s result is

<table>
<thead>
<tr>
<th>title</th>
<th>year</th>
<th>length</th>
<th>genre</th>
<th>studioName</th>
<th>producerC#</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrettyWoman</td>
<td>1990</td>
<td>119</td>
<td>romance</td>
<td>Disney</td>
<td>999</td>
</tr>
<tr>
<td>Ants</td>
<td>1990</td>
<td>90</td>
<td>cartoon</td>
<td>Disney</td>
<td>999</td>
</tr>
</tbody>
</table>
Given the relation

\[ \text{Movies} \left( \text{title, year, length, genre, studioName, producerC#} \right) \],

we want to know the titles and lengths of all movies produced by Disney Studios in 1990. In SQL, we say

\[
\begin{align*}
\text{SELECT} & \quad \text{title, length} \\
\text{FROM} & \quad \text{Movies} \\
\text{WHERE} & \quad \text{studioName = 'Disney' AND year = 1990;}
\end{align*}
\]

An example of the query’s result is

<table>
<thead>
<tr>
<th>title</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrettyWoman</td>
<td>119</td>
</tr>
<tr>
<td>Ants</td>
<td>90</td>
</tr>
</tbody>
</table>
We can modify the previous example to produce a relation with attributes **name** and **duration** in place of **title** and **length** as follows:

```
SELECT title AS name, length AS duration
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

An example of the query's result is

<table>
<thead>
<tr>
<th>name</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrettyWoman</td>
<td>119</td>
</tr>
<tr>
<td>Ants</td>
<td>90</td>
</tr>
</tbody>
</table>
Another option in the SELECT clause is to use an expression in place of an attribute.

Example

Suppose we want outputs as in the previous example, but with the length in hours. We can replace this statement:

```
SELECT title AS name, length*0.016667 AS lengthInHours
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

An example of the query’s result is

<table>
<thead>
<tr>
<th>name</th>
<th>lengthInHours</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrettyWoman</td>
<td>1.98334</td>
</tr>
<tr>
<td>Ants</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Projection in SQL

We can even allow a constant as an expression in the SELECT clause.

Example

The following query

```
SELECT title, length*0.016667 AS length, 'hrs.' AS inHours
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

produces tuples such as

<table>
<thead>
<tr>
<th>title</th>
<th>length</th>
<th>inHours</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrettyWoman</td>
<td>1.98334</td>
<td>hrs.</td>
</tr>
<tr>
<td>Ants</td>
<td>1.5</td>
<td>hrs.</td>
</tr>
</tbody>
</table>
The simple SQL queries we have seen have the form:

\[
\text{SELECT } L \\
\text{FROM } R \\
\text{WHERE } C
\]

The meaning of this expression is the same as that of the relational algebra expression

\[\pi_L(\sigma_C(R)).\]
The selection operator of relational algebra, and much more, is available through the **WHERE** clause of SQL. The expression that may follow **WHERE** include conditional expressions like those found in common languages such as C or Java.

We may build expressions by comparing values using the six common comparison operators:

\[=, \neq, <, >, \leq, \geq\].

The last four operators are `s` in C, but `\neq` is the SQL symbol for “not equal to”, and “`=” in SQL is equality.
The values that may be compared include constants and attributes of the relations mentioned after **FROM**. We may also apply the usual arithmetic operators, +, *, and so on, to numeric values before we compare them. Four instance,

\[(\text{year} - 1930) \times (\text{year} - 1930) < 100\]

is true for those years within 9 of 1930. We may apply the concatenation `||` to strings; for example,

`'mini' || 'bar'`

has value `'minibar.'`
The result of a comparison is a boolean value. Boolean values may be combined by the logical operators

\[ AND, \quad OR, \quad NOT. \]

Example

Consider the query

```
SELECT title
FROM Movies
WHERE (year > 1970 OR length < 90) AND studioName='MGM';
```
Two strings are equal if they are the same sequence of characters. Strings are compared with the lexicographic order. If one string is a proper prefix of another string, then the first string is less than the second string.

Example

'fodder' < 'foo'
'bar' < 'bargain'
The **WHERE** clause can have conditions in which a string is compared with a pattern, to see if it matches.

General Form:

\[
\text{attribute} \ \text{LIKE} \ \text{pattern}
\]

or

\[
\text{attribute} \ \text{NOT \ LIKE} \ \text{pattern}
\]

**Pattern** is a quoted string with

- `%` = “any string”;
- `_` = “any character”.

Pattern Matching in SQL
Example

From the relation

\[ \text{Consumers} \ (\text{name}, \ \text{address}, \ \text{phone}), \]

we want to find the consumers with the phone exchange number 555. We say:

```
SELECT name
FROM Consumers
WHERE phone LIKE '%%%555 - _ _ _ _';
```
Implementations of SQL generally support dates and times as special data types.

Examples of the format:

- DATE '1948-05-14'
- TIME '15:00:02.5'
- TIMESTAMP '1948-05-14 15:00:02.5' combines date and time.
Null Values and Comparisons Involving NULL

SQL allows attributes to have a special value NULL, which is called the **null value**. Null values may appear because the value is unknown, inapplicable or withheld.

In **WHERE** clauses, we must be prepared for the possibility that a component of some tuple we are examining will be NULL. There are two important rules to remember when we operate upon a NULL value:

- When we operate on a NULL and any value, including another NULL, using an arithmetic operator like $\times$ or $+$, the result is NULL.

- When we compare a NULL value and any value, including another NULL, using a comparison operator like $=$ or $>$, the result is UNKNOWN. The value UNKNOWN is another truth-value, like TRUE and FALSE. We shall discuss how to deal with it shortly.
We can check if an attribute \( x \) has the value NULL with the expression

\[ x \text{ IS NULL.} \]

This expression has the value TRUE if \( x \) has the value NULL, and has the value FALSE otherwise. Similarly,

\[ x \text{ IS NOT NULL} \]

has the value TRUE unless the value of \( x \) is NULL.
The logic of conditions in SQL is really **3-valued logic**: TRUE, FALSE, and 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.
- UNKNOWN = \( \frac{1}{2} \).
- AND = min.
- OR = max.
- NOT\( (x) = 1 - x \).

Example

\[
\begin{align*}
\text{TRUE AND (FALSE OR NOT(UNKNOWN))} &= \min(1, \max(0, (1 - \frac{1}{2}))) \\
&= \min(1, \max(0, \frac{1}{2})) \\
&= \min(1, \frac{1}{2}) \\
&= \frac{1}{2} \\
&= \text{UNKNOWN}
\end{align*}
\]
Ordering the Output

We may ask the tuples produced by a query be presented in sorted order. The order may be based on the value of any attribute, with ties broken by the value of a second attribute, remaining ties broken by a third, and so on, as in the sorting operation \( \tau \).

To get output in sorted order, we may add to the select-from-where statement a clause:

```
ORDER BY < list of attributes >
```

The order is by default ascending, but we can get the output highest-first by appending the keyword `DESC` (for “descending”) to an attribute.
The **ORDER BY** clause follows the **WHERE** clause and any other clauses (which we will learn later). The ordering is performed on the result of the FROM, WHERE and other clauses, just before we apply the SELECT clause. The tuples of the result are then sorted by the attributes in the list of the ORDER BY clause, and then passed to the SELECT clause for processing in the normal manner.

**Example**

```sql
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```
An additional option in ordering is that the list following ORDER BY can include expressions, just as the SELECT clause can.

**Example**

We can order the tuples of a relation $R(A, B)$ by the sum of the two components of the tuples, highest first, with:

```sql
SELECT * 
FROM R 
ORDER BY A + B DESC;
```
Introduction to SQL

Select-From-Where Statements
Subqueries
Grouping and Aggregation
Select-From-Where Statements

SELECT desired attributes
FROM one or more tables
WHERE condition about tuples of the tables
All our SQL queries will be based on the following database schema.

- Underline indicates key attributes.

- **Candies**(name, manf)
- **Stores**(name, addr, license)
- **Consumers**(name, addr, phone)
- **Likes**(consumer, candy)
- **Sells**(store, candy, price)
- **Frequents**(consumer, store)
Example

Using **Candies(name, manf)**, what candies are made by Hershey?

```sql
SELECT name
FROM Candies
WHERE manf = 'Hershey';
```

Notice SQL uses single-quotes for strings. SQL is *case-insensitive*, except inside strings.
The answer is a relation with a single attribute, name, and tuples with the name of each candy by Hershey, such as Twizzler.
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

- To implement this algorithm think of a *tuple variable* (tv) 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.
Operational Semantics

<table>
<thead>
<tr>
<th>name</th>
<th>manf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twizzler</td>
<td>Hershey</td>
</tr>
</tbody>
</table>

Include tv.name in the result

Check if Hershey
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 $\text{Likes}(\text{consumer}, \text{candy})$ and $\text{Frequents}(\text{consumer}, \text{store})$, find the candies liked by at least one person who frequents 7-11.

SELECT candy
FROM Likes, Frequents
WHERE store = '7-11' AND
    Frequents.consumer = Likes.consumer;
Almost the same as for single-relation queries:
- Start with the product of all the relations in the FROM clause.
- Apply the selection condition from the WHERE clause.
- Project onto the list of attributes and expressions in the SELECT clause.
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 Candies(name, manf), find all pairs of candies by the same manufacturer.*
- Do not produce pairs like (Twizzler, Twizzler).
- Produce pairs in alphabetic order, e.g. (Kitkat, Twizzler), not (Twizzler, Kitkat).

```
SELECT c1.name, c2.name
FROM Candies c1, Candies c2
WHERE c1.manf = c2.manf AND c1.name < c2.name;
```
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.
  - Can use a tuple-variable to name tuples of the result.
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 $\text{Sells}(\text{store}, \text{candy}, \text{price})$, find the stores that sell Kitkats for the same price 7-11 charges for Twizzlers.

- Two queries would surely work:
  - Find the price 7-11 charges for Twizzlers.
  - Find the stores that sell Kitkats at that price.
Query + Subquery Solution

```sql
SELECT store
FROM Sells
WHERE candy = 'Kitkat' AND
  price = (SELECT price
            FROM Sells
            WHERE store= '7-11'
            AND candy = 'Twizzler');
```

The price at which 7-11 sells Twizzlers
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 Candies(name, manf) and Likes(consumer, candy), find the name and manufacturer of each candy that Fred likes.

```
SELECT *
FROM Candies
WHERE name IN (SELECT candy
FROM Likes
WHERE consumer = 'Fred');
```

The set of candies Fred likes
The Exists Operator

- EXISTS( <relation> ) is true if and only if the <relation> is not empty.
- Example: From Candies(name, manf), find those candies that are the unique candy by their manufacturer.
Example Query with EXISTS

```
SELECT name
FROM Candies c1
WHERE NOT EXISTS(
    SELECT *
    FROM Candies
    WHERE manf = c1.manf AND
    name <> c1.name);
```

Set of candies with the same manf as c1, but not the same candy.

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

- $x = \text{ANY}( <\text{relation}> )$ is a boolean condition that is true if $x$ equals at least one tuple in the relation.
- Similarly, $=$ can be replaced by any of the comparison operators.
- Example: $x > \text{ANY}( <\text{relation}> )$ means $x$ is not the smallest tuple in the relation.
  - Note tuples must have one component only.
Similarly, $x <> \text{ALL}(<\text{relation}>)$ 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: $x \geq \text{ALL}(<\text{relation}>)$ means there is no tuple larger than $x$ in the relation.
Example

- From `Sells(store, candy, price)`, find the candies sold for the highest price.

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

  price from the outer Sells must not be less than any price.
Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:

- $(\text{subquery}) \text{ UNION } (\text{subquery})$
- $(\text{subquery}) \text{ INTERSECT } (\text{subquery})$
- $(\text{subquery}) \text{ EXCEPT } (\text{subquery})$
Example

- From relations \texttt{Likes(consumer, candy)}, \texttt{Sells(store, candy, price)}, and \texttt{Frequents(consumer, store)}, find the consumers and candies such that:
  - The consumer likes the candy, and
  - The consumer frequents at least one store that sells the candy.
The consumer frequents a store that sells the candy.
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

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 \textbf{Sells}([store, candy, price]), find all the different prices charged for candies:
  
  \begin{verbatim}
  SELECT DISTINCT price 
  FROM Sells;
  \end{verbatim}

- Notice that without DISTINCT, each price would be listed as many times as there were store/candy pairs at that price.
Example: ALL

- Using relations `Frequents(consumer, store)` and `Likes(consumer, candy)`:
  
  ```
  (SELECT consumer FROM Frequent)
  EXCEPT ALL
  (SELECT consumer FROM Likes);
  ```

- Lists consumers who frequent more stores than they like candies, and does so as many times as the difference of those counts.
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:
  \[ R \text{ NATURAL JOIN } S; \]
- Product:
  \[ R \text{ CROSS JOIN } S; \]
- Example:
  \[ \text{Likes NATURAL JOIN Sells;} \]
- Relations can be parenthesized subqueries, as well.

Theta Join

- R JOIN S ON <condition>
- Example: using Consumers(name, addr) and Frequents(consumer, store):

  Consumers JOIN Frequents ON
  name = consumer;

  gives us all (c, a, c, s) quadruples such that consumer c lives at address a and frequents store s.
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.
Aggregations

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.
- Also, COUNT(*) counts the number of tuples.
Example: Aggregation

- From `Sells(store, candy, price)`, find the average price of Twizzlers:

```
SELECT AVG(price)
FROM Sells
WHERE candy = 'Twizzler';
```
Eliminating Duplicates in an Aggregation

- Use DISTINCT inside an aggregation.
- Example: find the number of different prices charged for Twizzlers:

  ```sql
  SELECT COUNT(DISTINCT price)
  FROM Sells
  WHERE candy = 'Twizzler';
  ```
NULL’s Ignored in Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.
- But if there are no non-NULL values in a column, then the result of the aggregation is NULL.
Example: Effect of NULL’s

```sql
SELECT count(*)
FROM Sells
WHERE candy = 'Twizzler';
```

The number of stores that sell Twizzlers.

```sql
SELECT count(price)
FROM Sells
WHERE candy = 'Twizzler';
```

The number of stores that sell Twizzlers at a known price.
Grouping

- We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.
- The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group.
Example: Grouping

- From `Sells(store, candy, price)`, find the average price for each candy:

```
SELECT candy, AVG(price)
FROM Sells
GROUP BY candy;
```
From \textit{Sells}((store, candy, price)) and \textit{Frequents} (consumer, store), find for each consumer the average price of Twizzlers at the stores they frequent:

\begin{verbatim}
SELECT consumer, AVG(price) 
FROM Frequents, Sells 
WHERE candy = 'Twizzler' AND 
  Frequents.store = Sells.store 
GROUP BY consumer;
\end{verbatim}
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.
HAVING Clauses

- HAVING <condition> may follow a GROUP BY clause.
- If so, the condition applies to each group, and groups not satisfying the condition are eliminated.
Example: HAVING

- From $\text{Sells}(\text{store}, \text{candy}, \text{price})$ and $\text{Candies}(\text{name}, \text{manf})$, find the average price of those candies that are either sold in at least three stores or are manufactured by Nestle.
Solution

```
SELECT candy, AVG(price)
FROM Sells
GROUP BY candy
HAVING COUNT(store) >= 3 OR candy IN (SELECT name FROM Candies WHERE manf = 'Nestle');
```

Candy groups with at least 3 non-NULL stores and also candy groups where the manufacturer is Nestle.

Candies manufactured by Nestle.
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:
  - A grouping attribute, or
  - Aggregated.
More on 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(consumer, candy)` the fact that Sally likes Twizzlers.

```
INSERT INTO Likes
VALUES('Sally', 'Twizzler');
```
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.
Another way to add the fact that Sally likes Twizzlers to Likes(consumer, candy):

```sql
INSERT INTO Likes(candy, consumer)
VALUES ('Twizzler', 'Sally');
```
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(consumer, store)`, enter into the new relation `CoShoppers(name)` all of Sally’s “co-shoppers,” i.e., those consumers who frequent at least one store that Sally also frequents.
INSERT INTO CoShoppers
(SELECT c2.consumer
FROM Frequents c1, Frequents c2
WHERE c1.consumer = 'Sally' AND c2.consumer <> 'Sally' AND c1.store = c2.store
);

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

The other consumer (shopper)
To delete tuples satisfying a condition from some relation:

```
DELETE FROM <relation> 
WHERE <condition>;
```
Example: Deletion

Delete from \texttt{Likes(consumer, candy)} the fact that Sally likes Twizzlers:

\begin{verbatim}
DELETE FROM Likes
WHERE consumer = 'Sally' AND candy = 'Twizzler';
\end{verbatim}
Example: Delete all Tuples

- Make the relation Likes empty:
  
  ```sql
  DELETE FROM Likes;
  ```

- Note no WHERE clause needed.
Example: Delete Many Tuples

- Delete from **Candies**(name, manf) all candies for which there is another candy by the same manufacturer.

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

Candies with the same manufacturer and a different name from the name of the candy represented by tuple c.
Suppose Hershey makes only Twizzlers and Kitkats.
Suppose we come to the tuple \( c \) for Twizzler first.
The subquery is nonempty, because of the Kitkat tuple, so we delete Twizzler.
Now, when \( c \) is the tuple for Kitkat, do we delete that tuple too?
Answer: we *do* delete Kitkat as well.

The reason is that deletion proceeds in two stages:

- Mark all tuples for which the WHERE condition is satisfied.
- Delete the marked tuples.
Updates

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

```sql
UPDATE <relation>
SET <list of attribute assignments>
WHERE <condition on tuples>;
```
Example: Update

- Change consumer Fred’s phone number to 555-1212:
  
  UPDATE Consumers
  SET phone = '555-1212'
  WHERE name = 'Fred';
Example: Update Several Tuples

Make $4 the maximum price for candy:

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

```sql
CREATE TABLE Sells (  
    store    CHAR(20),  
    candy    VARCHAR(20),  
    price    REAL  
);
```
DATE and TIME are types in SQL.
The form of a date value is:

```
DATE 'yyyy-mm-dd'
```

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:

```sql
CREATE TABLE Candies (  
    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.
The store and candy together are the key for Sells:

```sql
CREATE TABLE Sells (  
  store CHAR(20),  
  candy VARCHAR(20),  
  price REAL,  
  PRIMARY KEY (store, candy)  
);
```
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.
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.
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>.
CREATE TABLE Consumers (  
  name CHAR(30) PRIMARY KEY,  
  addr CHAR(50)  
    DEFAULT '123 Sesame St.',  
  phone CHAR(16)  
);
Suppose we insert the fact that Sally is a consumer, but we know neither her address nor her phone.

An INSERT with a partial list of attributes makes the insertion possible:

```
INSERT INTO Consumers(name)
VALUES('Sally');
```
Effect of Defaults --- (2)

- But what tuple appears in Consumers?

<table>
<thead>
<tr>
<th>name</th>
<th>addr</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sally</td>
<td>123 Sesame St</td>
<td>NULL</td>
</tr>
</tbody>
</table>

- 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:

  ```sql
  ALTER TABLE <name> ADD <attribute declaration>;
  ```

- **Example:**

  ```sql
  ALTER TABLE Stores ADD phone CHAR(16) DEFAULT 'unlisted';
  ```
Deleting Attributes

- Remove an attribute from a relation schema by:
  ```sql
  ALTER TABLE <name>
  DROP <attribute>;
  ```
- Example: we don’t really need the license attribute for stores:
  ```sql
  ALTER TABLE Stores DROP license;
  ```
Views

- **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

- **CanEat(consumer, candy)** is a view “containing” the consumer-candy pairs such that the consumer frequents at least one store that sells the candy:

  CREATE VIEW CanEat AS
  SELECT consumer, candy
  FROM Frequents, Sells
  WHERE Frequents.store = Sells.store;
Example: Accessing a View

- Query a view as if it were a base table.
- Example query:

```sql
SELECT candy FROM CanEat
WHERE consumer = '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

```
PROJ_{candy}

SELECT_{consumer='Sally'}

CanEat

PROJ_{consumer, candy}

JOIN

Frequents  Sells
```
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:

- Push selections down the tree.
- Eliminate unnecessary projections.
Example: Optimization

Notice how most tuples are eliminated from Frequents before the expensive join.

\[
\text{SELECT}_{\text{consumer} = 'Sally'} \\
\text{Frequents}
\]

\[
\text{JOIN}
\]

\[
\text{PROJ}_{\text{candy}}
\]

\[
\text{Sells}
\]