Chapter 3

The Relational Model

1. The first database systems were based on the network and hierarchical models. These are covered briefly in appendices in the text. The relational model was first proposed by E.F. Codd in 1970 and the first such systems (notably INGRES and System/R) was developed in 1970s. The relational model is now the dominant model for commercial data processing applications.

2. Note: **Attribute Name Abbreviations**

   The text uses fairly long attribute names which are abbreviated in the notes as follows.
   
   - `customer-name` becomes `cname`
   - `customer-city` becomes `ccity`
   - `branch-city` becomes `bcity`
   - `branch-name` becomes `bname`
   - `account-number` becomes `account#`
   - `loan-number` becomes `loan#`
   - `banker-name` becomes `banker`

3.1 Structure of Relational Database

1. A relational database consists of a collection of tables, each having a unique name.

   A row in a table represents a relationship among a set of values.

   Thus a table represents a collection of relationships.

2. There is a direct correspondence between the concept of a table and the mathematical concept of a relation. A substantial theory has been developed for relational databases.

3.1.1 Basic Structure

1. Figure 3.1 shows the deposit and customer tables for our banking example.
   
   - It has four attributes.
   - For each attribute there is a permitted set of values, called the domain of that attribute.
Let $D_1$ denote the domain of $bname$, and $D_2$, $D_3$ and $D_4$ the remaining attributes’ domains respectively. Then, any row of $deposit$ consists of a four-tuple $(v_1, v_2, v_3, v_4)$ where

$$v_1 \in D_1, v_2 \in D_2, v_3 \in D_3, v_4 \in D_4$$

In general, $deposit$ contains a subset of the set of all possible rows.

That is, $deposit$ is a subset of

$$D_1 \times D_2 \times D_3 \times D_4, \quad \text{or, abbreviated to,} \quad \times_{i=1}^4 D_i$$

In general, a table of $n$ columns must be a subset of

$$\times_{i=1}^n D_i \quad (\text{all possible rows})$$

2. Mathematicians define a relation to be a subset of a Cartesian product of a list of domains. You can see the correspondence with our tables.

We will use the terms relation and tuple in place of table and row from now on.

3. Some more formalities:

- let the tuple variable $t$ refer to a tuple of the relation $r$.
- We say $t \in r$ to denote that the tuple $t$ is in relation $r$.
- Then $t[bname] = t[1]$ = the value of $t$ on the $bname$ attribute.

4. We’ll also require that the domains of all attributes be indivisible units.

- A domain is atomic if its elements are indivisible units.
- For example, the set of integers is an atomic domain.
- The set of all sets of integers is not.
- Why? Integers do not have subparts, but sets do — the integers comprising them.
- We could consider integers non-atomic if we thought of them as ordered lists of digits.
3.1.1 Structure of Relational Database

3.1.2 Database Schema

1. We distinguish between a database scheme (logical design) and a database instance (data in the database at a point in time).

2. A relation scheme is a list of attributes and their corresponding domains.

3. The text uses the following conventions:
   - italics for all names
   - lowercase names for relations and attributes
   - names beginning with an uppercase for relation schemes

   These notes will do the same.

   For example, the relation scheme for the deposit relation:
   
   $$\text{Deposit-scheme} = (\text{bname}, \text{account\#}, \text{cname}, \text{balance})$$

   We may state that deposit is a relation on scheme Deposit-scheme by writing deposit(Deposit-scheme).

   If we wish to specify domains, we can write:
   
   $$\text{bname: string, account\#: integer, cname: string, balance: integer}.$$

   Note that customers are identified by name. In the real world, this would not be allowed, as two or more customers might share the same name.

   Figure 3.2 shows the E-R diagram for a banking enterprise.

4. The relation schemes for the banking example used throughout the text are:
   
   $$\text{Branch-scheme} = (\text{bname, assets, bcity})$$
   $$\text{Customer-scheme} = (\text{cname, street, ccity})$$
   $$\text{Deposit-scheme} = (\text{bname, account\#, cname, balance})$$
   $$\text{Borrow-scheme} = (\text{bname, loan\#, cname, amount})$$

   Note: some attributes appear in several relation schemes (e.g., bname, cname). This is legal, and provides a way of relating tuples of distinct relations.

5. Why not put all attributes in one relation?
CHAPTER 3. THE RELATIONAL MODEL

Suppose we use one large relation instead of customer and deposit:

- \( \text{Account-scheme} = (\text{bname}, \text{account\#}, \text{cname}, \text{balance}, \text{street}, \text{ccity}) \)

- If a customer has several accounts, we must duplicate her or his address for each account.
- If a customer has an account but no current address, we cannot build a tuple, as we have no values for the address.
- We would have to use null values for these fields.
- Null values cause difficulties in the database.
- By using two separate relations, we can do this without using null values

3.1.3 Keys

1. The notions of superkey, candidate key and primary key all apply to the relational model.

2. For example, in Branch-scheme,
   - \{bname\} is a superkey.
   - \{bname, bcity\} is a superkey.
   - \{bname, bcity\} is not a candidate key, as the superkey \{bname\} is contained in it.
   - \{bname\} is a candidate key.
   - \{bcity\} is not a superkey, as branches may be in the same city.
   - We will use \{bname\} as our primary key.

3. The primary key for Customer-scheme is \{cname\}.

4. More formally, if we say that a subset \(K\) of \(R\) is a superkey for \(R\), we are restricting consideration to relations \(r(R)\) in which no two distinct tuples have the same values on all attributes in \(K\). In other words,
   - If \(t_1\) and \(t_2\) are in \(r\), and
   - \(t_1 \neq t_2\),
   - then \(t_1[K] \neq t_2[K]\).

3.1.4 Query Languages

1. A query language is a language in which a user requests information from a database. These are typically higher-level than programming languages.

   They may be one of:

   - **Procedural**, where the user instructs the system to perform a sequence of operations on the database. This will compute the desired information.

   - **Nonprocedural**, where the user specifies the information desired without giving a procedure for obtaining the information.

2. A complete query language also contains facilities to insert and delete tuples as well as to modify parts of existing tuples.
3.2. THE RELATIONAL ALGEBRA

<table>
<thead>
<tr>
<th>bname</th>
<th>loan#</th>
<th>cname</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>17</td>
<td>Jones</td>
<td>1000</td>
</tr>
<tr>
<td>Lougheed_Mall</td>
<td>23</td>
<td>Smith</td>
<td>2000</td>
</tr>
<tr>
<td>SFU</td>
<td>15</td>
<td>Hayes</td>
<td>1500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bname</th>
<th>assets</th>
<th>bcity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>9,000,000</td>
<td>Vancouver</td>
</tr>
<tr>
<td>Lougheed_Mall</td>
<td>21,000,000</td>
<td>Burnaby</td>
</tr>
<tr>
<td>SFU</td>
<td>17,000,000</td>
<td>Burnaby</td>
</tr>
</tbody>
</table>

Figure 3.3: The borrow and branch relations.

3.2 The Relational Algebra

1. The relational algebra is a procedural query language.
   - Six fundamental operations:
     - select (unary)
     - project (unary)
     - rename (unary)
     - cartesian product (binary)
     - union (binary)
     - set-difference (binary)
   - Several other operations, defined in terms of the fundamental operations:
     - set-intersection
     - natural join
     - division
     - assignment
   - Operations produce a new relation as a result.

3.2.1 Fundamental Operations

1. The Select Operation
   Select selects tuples that satisfy a given predicate. Select is denoted by a lowercase Greek sigma (σ), with the predicate appearing as a subscript. The argument relation is given in parentheses following the σ.
   For example, to select tuples (rows) of the borrow relation where the branch is "SFU", we would write
   \[ \sigma_{\text{bname}=\text{SFU}}(\text{borrow}) \]
   Let Figure 3.3 be the borrow and branch relations in the banking example.
   The new relation created as the result of this operation consists of one tuple: \((SFU, 15, Hayes, 1500)\).
   We allow comparisons using \(=, \neq, \leq, >, \geq\) and \(\in\) in the selection predicate.
   We also allow the logical connectives \(\lor\) (or) and \(\land\) (and). For example:
   \[ \sigma_{\text{bname}=\text{Downtown} \land \text{amount}>1200}(\text{borrow}) \]
   Suppose there is one more relation, client, shown in Figure 3.4, with the scheme
   \[ \text{Client\_scheme} = (\text{cname}, \text{banker}) \]
   we might write
   \[ \sigma_{\text{cname}=\text{banker}}(\text{client}) \]
   to find clients who have the same name as their banker.

2. The Project Operation
   Project copies its argument relation for the specified attributes only. Since a relation is a set, duplicate rows are eliminated. Projection is denoted by the Greek capital letter pi (Π). The attributes to be copied
appear as subscripts.

For example, to obtain a relation showing customers and branches, but ignoring amount and loan#, we write

$$\pi_{\text{cname, banker}}(\text{borrow})$$

We can perform these operations on the relations resulting from other operations. To get the names of customers having the same name as their bankers,

$$\pi_{\text{cname}}(\sigma_{\text{cname=banker}}(\text{client}))$$

Think of select as taking rows of a relation, and project as taking columns of a relation.

3. The Cartesian Product Operation

The cartesian product of two relations is denoted by a cross (×), written

$$r_1 \times r_2$$

for relations $$r_1$$ and $$r_2$$

The result of $$r_1 \times r_2$$ is a new relation with a tuple for each possible pairing of tuples from $$r_1$$ and $$r_2$$. In order to avoid ambiguity, the attribute names have attached to them the name of the relation from which they came. If no ambiguity will result, we drop the relation name.

The result $$\text{client} \times \text{customer}$$ is a very large relation. If $$r_1$$ has $$n_1$$ tuples, and $$r_2$$ has $$n_2$$ tuples, then $$r = r_1 \times r_2$$ will have $$n_1 n_2$$ tuples.

The resulting scheme is the concatenation of the schemes of $$r_1$$ and $$r_2$$, with relation names added as mentioned.

To find the clients of banker Johnson and the city in which they live, we need information in both client and customer relations. We can get this by writing

$$\sigma_{\text{banker=“Johnson”}}(\text{client} \times \text{customer})$$

However, the customer.cname column contains customers of bankers other than Johnson. (Why?)

We want rows where $$\text{client.cname = customer.cname}$$. So we can write

$$\sigma_{\text{client.cname=customer.cname}}(\sigma_{\text{banker=“Johnson”}}(\text{client} \times \text{customer}))$$

to get just these tuples. Finally, to get just the customer’s name and city, we need a projection:

$$\pi_{\text{client.cname, city}}(\sigma_{\text{client.cname=customer.cname}}(\sigma_{\text{banker=“Johnson”}}(\text{client} \times \text{customer})))$$

4. The Rename Operation

The rename operation solves the problems that occurs with naming when performing the cartesian product of a relation with itself. Suppose we want to find the names of all the customers who live on the same street and in the same city as Smith. We can get the street and city of Smith by writing

$$\pi_{\text{street, city}}(\sigma_{\text{cname=“Smith”}}(\text{customer}))$$

To find other customers with the same information, we need to reference the customer relation again:
3.2. THE RELATIONAL ALGEBRA

\[
\sigma_P(\text{customer} \times (\Pi_{\text{street},\text{city}} (\sigma_{\text{cname}=\text{Smith}}(\text{customer})))))
\]

where \( P \) is a selection predicate requiring \textit{street} and \textit{city} values to be equal.

\textbf{Problem:} how do we distinguish between the two street values appearing in the Cartesian product, as both come from a \textit{customer} relation?

\textbf{Solution:} use the rename operator, denoted by the Greek letter \( \rho \). We write

\[
\rho_x(r)
\]

to get the relation \( r \) under the name of \( x \).

If we use this to rename one of the two \textit{customer} relations we are using, the ambiguities will disappear.

\[
\Pi_{\text{customer},\text{cname}}(\sigma \text{cust1.street} = \text{customer.street} \land \text{cust1.city} = \text{customer.city} (\text{customer} \times (\Pi_{\text{street},\text{city}} (\sigma_{\text{cname}=\text{Smith}}(\rho_{\text{cust2}}(\text{customer}))))))
\]

5. The Union Operation

The \textit{union} operation is denoted \( \cup \) as in set theory. It returns the union (set union) of two compatible relations. For a union operation \( r \cup s \) to be legal, we require that

- \( r \) and \( s \) must have the same number of attributes.
- The domains of the corresponding attributes must be the same.

To find all customers of the SFU branch, we must find everyone who has a loan or an account or both at the branch. We need both \textit{borrow} and \textit{deposit} relations for this:

\[
\Pi_{\text{ename}}(\sigma_{\text{bname}=\text{SFU}}(\text{borrow})) \cup \Pi_{\text{ename}}(\sigma_{\text{bname}=\text{SFU}}(\text{deposit}))
\]

As in all set operations, duplicates are eliminated, giving the relation of Figure 3.5(a).

6. The Set Difference Operation

Set difference is denoted by the minus sign (\( - \)). It finds tuples that are in one relation, but not in another. Thus \( r - s \) results in a relation containing tuples that are in \( r \) but not in \( s \).

To find customers of the SFU branch who have an account there but no loan, we write

\[
\Pi_{\text{ename}}(\sigma_{\text{bname}=\text{SFU}}(\text{deposit})) - \Pi_{\text{ename}}(\sigma_{\text{bname}=\text{SFU}}(\text{borrow}))
\]

The result is shown in Figure 3.5(b).

We can do more with this operation. Suppose we want to find the largest account balance in the bank. Strategy:

- Find a relation \( r \) containing the balances not the largest.
- Compute the set difference of \( r \) and the \textit{deposit} relation.
To find $r$, we write

$$
\Pi_{\text{deposit.balance}}(\sigma_{\text{deposit.balance} < \text{balance}}(\text{deposit} \times \rho_{d}(\text{deposit})))
$$

This resulting relation contains all balances except the largest one. (See Figure 3.6(a)). Now we can finish our query by taking the set difference:

$$
\Pi_{\text{balance}(\text{deposit})} - \Pi_{\text{deposit.balance}}(\sigma_{\text{deposit.balance} < \text{balance}}(\text{deposit} \times \rho_{d}(\text{deposit})))
$$

Figure 3.6(b) shows the result.

### 3.2.2 Formal Definition of Relational Algebra

1. A basic expression consists of either
   - A relation in the database.
   - A constant relation.

2. General expressions are formed out of smaller subexpressions using
   - $\sigma_{p}(E_{1})$ select ($p$ a predicate)
   - $\Pi_{s}(E_{1})$ project ($s$ a list of attributes)
   - $\rho_{x}(E_{1})$ rename ($x$ a relation name)
   - $E_{1} \cup E_{2}$ union
   - $E_{1} - E_{2}$ set difference
   - $E_{1} \times E_{2}$ cartesian product

### 3.2.3 Additional Operations

1. Additional operations are defined in terms of the fundamental operations. They do not add power to the algebra, but are useful to simplify common queries.

2. **The Set Intersection Operation**

   Set intersection is denoted by $\cap$, and returns a relation that contains tuples that are in both of its argument relations. It does not add any power as

   $$
   r \cap s = r - (r - s)
   $$

   To find all customers having both a loan and an account at the SFU branch, we write

   $$
   \Pi_{\text{name}}(\sigma_{\text{name} = \text{SFU}}(\text{borrow})) \cap \Pi_{\text{name}}(\sigma_{\text{name} = \text{SFU}}(\text{deposit}))
   $$

3. **The Natural Join Operation**

   Often we want to simplify queries on a cartesian product. For example, to find all customers having a loan
3.2. THE RELATIONAL ALGEBRA

<table>
<thead>
<tr>
<th>cname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Burnaby</td>
</tr>
<tr>
<td>Hayes</td>
<td>Burnaby</td>
</tr>
<tr>
<td>Jones</td>
<td>Vancouver</td>
</tr>
</tbody>
</table>

Figure 3.7: Joining borrow and customer relations.

at the bank and the cities in which they live, we need borrow and customer relations:

$$\Pi_{\text{borrow.cname,city}}(\sigma_{\text{borrow.cname}=\text{customer.cname}}(\text{borrow} \times \text{customer}))$$

Our selection predicate obtains only those tuples pertaining to only one cname.

This type of operation is very common, so we have the natural join, denoted by a $\bowtie$ sign. Natural join combines a cartesian product and a selection into one operation. It performs a selection forcing equality on those attributes that appear in both relation schemes. Duplicates are removed as in all relation operations.

To illustrate, we can rewrite the previous query as

$$\Pi_{\text{cname,city}}(\text{borrow $\bowtie$ customer})$$

The resulting relation is shown in Figure 3.7.

We can now make a more formal definition of natural join.

- Consider $R$ and $S$ to be sets of attributes.
- We denote attributes appearing in both relations by $R \cap S$.
- We denote attributes in either or both relations by $R \cup S$.
- Consider two relations $r(R)$ and $s(S)$.
- The natural join of $r$ and $s$, denoted by $r \bowtie s$ is a relation on scheme $R \cup S$.
- It is a projection onto $R \cup S$ of a selection on $r \times s$ where the predicate requires $r.A = s.A$ for each attribute $A$ in $R \cap S$.

Formally,

$$r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1=s.A_1 \land r.A_2=s.A_2 \land \ldots \land r.A_n=s.A_n}(r \times s))$$

where $R \cap S = \{A_1, A_2, \ldots, A_n\}$.

To find the assets and names of all branches which have depositors living in Stamford, we need customer, deposit and branch relations:

$$\Pi_{\text{bname,assets}}(\sigma_{\text{city}=\text{"Stamford"}}(\text{customer} \bowtie \text{deposit} \bowtie \text{branch}))$$

Note that $\bowtie$ is associative.

To find all customers who have both an account and a loan at the SFU branch:

$$\Pi_{\text{cname}}(\sigma_{\text{bname}=\text{"SFU"}}(\text{borrow} \bowtie \text{deposit}))$$

This is equivalent to the set intersection version we wrote earlier. We see now that there can be several ways to write a query in the relational algebra.

If two relations $r(R)$ and $s(S)$ have no attributes in common, then $R \cap S = \emptyset$, and $r \bowtie s = r \times s$.

4. The Division Operation

Division, denoted $\div$, is suited to queries that include the phrase “for all”.

Suppose we want to find all the customers who have an account at all branches located in Brooklyn. Strategy:
think of it as three steps.

We can obtain the names of all branches located in Brooklyn by

\[ r_1 = \Pi_{branch} (\sigma_{city = "Brooklyn"}(branch)) \]

Figure 3.19 in the textbook shows the result.

We can also find all `ename`, `bname` pairs for which the customer has an account by

\[ r_2 = \Pi_{ename, bname}(deposit) \]

Figure 3.20 in the textbook shows the result.

Now we need to find all customers who appear in \( r_2 \) with **every** branch name in \( r_1 \). The divide operation provides exactly those customers:

\[ \Pi_{ename, bname}(deposit) \div \Pi_{ename}(\sigma_{city = "Brooklyn"}(branch)) \]

which is simply \( r_2 \div r_1 \).

Formally,

- Let \( r(R) \) and \( s(S) \) be relations.
- Let \( S \subseteq R \).
- The relation \( r \div s \) is a relation on scheme \( R - S \).
- A tuple \( t \) is in \( r \div s \) if for every tuple \( t_s \) in \( s \) there is a tuple \( t_r \) in \( r \) satisfying both of the following:
  \[ t_r[S] = t_s[S] \]
  \[ t_r[R - S] = t[R - S] \]

These conditions say that the \( R - S \) portion of a tuple \( t \) is in \( r \div s \) if and only if there are tuples with the \( R - S \) portion and the \( S \) portion in \( r \) for **every** value of the \( S \) portion in relation \( S \).

We will look at this explanation in class more closely.

The division operation can be defined in terms of the fundamental operations.

\[ r \div s = \Pi_{R - S}(r) - \Pi_{R - S}((\Pi_{R - S}(r) \times s) - r) \]

Read the text for a more detailed explanation.

5. **The Assignment Operation**

Sometimes it is useful to be able to write a relational algebra expression in parts using a temporary relation variable (as we did with \( r_1 \) and \( r_2 \) in the division example).

The assignment operation, denoted \(-\), works like assignment in a programming language. We could rewrite our division definition as

\[
\begin{align*}
temp1 &= \Pi_{R - S}(r) \\
temp2 &= \Pi_{R - S}(\Pi_{R - S}(r) \times s) - r) \\
result &= temp1 - temp2
\end{align*}
\]

No extra relation is added to the database, but the relation variable created can be used in subsequent expressions. Assignment to a permanent relation would constitute a modification to the database.

3.3 **The Tuple Relational Calculus**

1. The tuple relational calculus is a nonprocedural language. (The relational algebra was procedural.) We must
provide a formal description of the information desired.

2. A query in the tuple relational calculus is expressed as

\[ \{ t \mid P(t) \} \]

i.e. the set of tuples \( t \) for which predicate \( P \) is true.

3. We also use the notation

- \( t[a] \) to indicate the value of tuple \( t \) on attribute \( a \).
- \( t \in r \) to show that tuple \( t \) is in relation \( r \).

### 3.3.1 Example Queries

1. For example, to find the branch-name, loan number, customer name and amount for loans over \$1200:

\[ \{ t \mid t \in borrow \land t[amount] > 1200 \} \]

This gives us all attributes, but suppose we only want the customer names. (We would use `project` in the algebra.) We need to write an expression for a relation on scheme \((cname)\).

\[ \{ t \mid \exists s \in borrow (t[cname] = s[cname] \land s[amount] > 1200) \} \]

In English, we may read this equation as “the set of all tuples \( t \) such that there exists a tuple \( s \) in the relation \( borrow \) for which the values of \( t \) and \( s \) for the \( cname \) attribute are equal, and the value of \( s \) for the \( amount \) attribute is greater than 1200.”

The notation \( \exists t \in r(Q(t)) \) means “there exists a tuple \( t \) in relation \( r \) such that predicate \( Q(t) \) is true.”

How did we get the above expression? We needed tuples on scheme \( cname \) such that there were tuples in \( borrow \) pertaining to that customer name with \( amount \) attribute greater than 1200.

The tuples \( t \) get the scheme \( cname \) implicitly as that is the only attribute \( t \) is mentioned with.

Let’s look at a more complex example.

Find all customers having a loan from the SFU branch, and the the cities in which they live:

\[ \{ t \mid \exists s \in borrow (t[cname] = s[cname] \land s[bname] = \text{"SFU"}) \land \exists u \in customer (u[cname] = s[cname] \land u[ccity] = u[ccity]) \} \]

In English, we might read this as “the set of all \((cname,ccity)\) tuples for which \( cname \) is a borrower at the SFU branch, and \( ccity \) is the city of \( name \).”

Tuple variable \( s \) ensures that the customer is a borrower at the SFU branch. Tuple variable \( u \) is restricted to pertaining to the same customer as \( s \), and also ensures that \( ccity \) is the city of the customer.

The logical connectives \( \land \) (AND) and \( \lor \) (OR) are allowed, as well as \( \neg \) (negation).

We also use the existential quantifier \( \exists \) and the universal quantifier \( \forall \).

Some more examples:

1. Find all customers having a loan, an account, or both at the SFU branch:

\[ \{ t \mid \exists s \in borrow (t[cname] = s[cname] \land s[bname] = \text{"SFU"}) \lor \exists u \in deposit (t[cname] = u[cname] \land u[bname] = \text{"SFU"}) \} \]

Note the use of the \( \lor \) connective.

As usual, set operations remove all duplicates.
2. Find all customers who have \textbf{both} a loan and an account at the SFU branch.
Solution: simply change the $\lor$ connective in 1 to $\land$.
3. Find customers who have an account, but \textbf{not} a loan at the SFU branch.
   \[
   \{ t \mid \exists s \in \text{deposit}(t[cname] = u[cname] \land u[bname] = \text{"SFU"}) \land \neg \exists s \in \text{borrow}(t[cname] = s[cname] \land s[bname] = \text{"SFU"}) \} 
   \]
4. Find all customers who have an account at \textbf{all} branches located in Brooklyn. (We used \texttt{division} in relational algebra.)
   For this example we will use implication, denoted by a pointing finger in the text, but by $\Rightarrow$ here. The formula $P \Rightarrow Q$ means $P$ implies $Q$, or, if $P$ is true, then $Q$ must be true.
   \[
   \{ t \mid \forall u \in \text{branch} (u[bcity] = \text{"Brooklyn"}) \Rightarrow \\
   \exists s \in \text{deposit}(t[cname] = s[cname] \land u[bname] = s[bname]) \}
   \]
   In English: the set of all $cname$ tuples $t$ such that for all tuples $u$ in the branch relation, if the value of $u$ on attribute $bcity$ is Brooklyn, then the customer has an account at the branch whose name appears in the $bname$ attribute of $u$.
   Division is difficult to understand. Think it through carefully.

### 3.3.2 Formal Definitions

1. A tuple relational calculus expression is of the form
   \[
   \{ t \mid P(t) \}
   \]
   where $P$ is a \textbf{formula}. Several tuple variables may appear in a formula.

2. A tuple variable is said to be a \textbf{free variable} unless it is quantified by a $\exists$ or a $\forall$. Then it is said to be a \textbf{bound variable}.

3. A formula is built of \textbf{atoms}. An atom is one of the following forms:
   - $s \in r$, where $s$ is a tuple variable, and $r$ is a relation ($\notin r$ is not allowed).
   - $s[x] \Theta u[y]$, where $s$ and $u$ are tuple variables, and $x$ and $y$ are attributes, and $\Theta$ is a comparison operator ($<, \leq, =, \neq, >, \geq$).
   - $s[x] \Theta c$, where $c$ is a constant in the domain of attribute $x$.

4. \textbf{Formulae} are built up from atoms using the following rules:
   - An atom is a formula.
   - If $P$ is a formula, then so are $\neg P$ and $(P)$.
   - If $P_1$ and $P_2$ are formulae, then so are $P_1 \lor P_2$, $P_1 \land P_2$ and $P_1 \Rightarrow P_2$.
   - If $P(s)$ is a formula containing a free tuple variable $s$, then
     \[
     \exists s \in r(P(s)) \land \forall s \in r(P(s))
     \]
     are formulae also.

5. Note some equivalences:
   - $P_1 \land P_2 = \neg(\neg P_1 \lor \neg P_2)$
   - $\forall t \in r(P(t)) = \neg \exists t \in r(\neg P(t))$
   - $P_1 \Rightarrow P_2 = \neg P_1 \lor P_2$
3.3.3 Safety of Expressions

1. A tuple relational calculus expression may generate an infinite expression, e.g.
   \[ \{ t \mid \neg(t \in \text{borrow}) \} \]

2. There are an infinite number of tuples that are not in \text{borrow}! Most of these tuples contain values that do not appear in the database.

3. **Safe Tuple Expressions**
   We need to restrict the relational calculus a bit.
   - The domain of a formula \( P \), denoted \( \text{dom}(P) \), is the set of all values referenced in \( P \).
   - These include values mentioned in \( P \) as well as values that appear in a tuple of a relation mentioned in \( P \).
   - So, the domain of \( P \) is the set of all values explicitly appearing in \( P \) or that appear in relations mentioned in \( P \).
   - \( \text{dom}(t \in \text{borrow} \land t[\text{amount}] < 1200) \) is the set of all values appearing in \text{borrow}.
   - \( \text{dom}(t \mid \neg(t \in \text{borrow})) \) is the set of all values appearing in \text{borrow}.

   We may say an expression \( \{ t \mid P(t) \} \) is **safe** if all values that appear in the result are values from \( \text{dom}(P) \).

4. A **safe** expression yields a finite number of tuples as its result. Otherwise, it is called **unsafe**.

3.3.4 Expressive Power of Languages

1. The tuple relational calculus restricted to safe expressions is equivalent in expressive power to the relational algebra.

3.4 The Domain Relational Calculus

1. Domain variables take on values from an attribute's domain, rather than values for an entire tuple.

3.4.1 Formal Definitions

1. An expression is of the form
   \[ \{ (x_1, x_2, \ldots, x_n) \mid P(x_1, x_2, \ldots, x_n) \} \]
   where the \( x_i \), \( 1 \leq i \leq n \), represent domain variables, and \( P \) is a formula.

2. An atom in the domain relational calculus is of the following forms
   - \( (x_1, \ldots, x_n) \in r \) where \( r \) is a relation on \( n \) attributes, and \( x_i, 1 \leq i \leq n \), are domain variables or constants.
   - \( x \Theta y \), where \( x \) and \( y \) are domain variables, and \( \Theta \) is a comparison operator.
   - \( x \Theta c \), where \( c \) is a constant.

3. **Formulae** are built up from atoms using the following rules:
   - An atom is a formula.
   - If \( P \) is a formula, then so are \( \neg P \) and \( (P) \).
   - If \( P_1 \) and \( P_2 \) are formulae, then so are \( P_1 \lor P_2 \), \( P_1 \land P_2 \) and \( P_1 \Rightarrow P_2 \).
   - If \( P(x) \) is a formula where \( x \) is a domain variable, then so are \( \exists x(P(x)) \) and \( \forall x(P(x)) \).
3.4.2 Example Queries

1. Find branch name, loan number, customer name and amount for loans of over $1200.
   $$\{(b, l, c, a) | (b, l, c, a) \in \text{borrow} \land a > 1200\}$$

2. Find all customers who have a loan for an amount greater than $1200.
   $$\{c | \exists b, l, a((b, l, c, a) \in \text{borrow} \land a > 1200)\}$$

3. Find all customers having a loan from the SFU branch, and the city in which they live.
   $$\{(c, x) | \exists b, l, a((b, l, c, a) \in \text{borrow} \land b = \text{SFU} \land \exists y((c, y, x) \in \text{customer}))\}$$

4. Find all customers having a loan, an account or both at the SFU branch.
   $$\{c | \exists b, l, a((b, l, c, a) \in \text{borrow} \land b = \text{SFU})$$
   $$\land \exists b, a, n((b, a, c, n) \in \text{deposit} \land b = \text{SFU})\}$$

5. Find all customers who have an account at all branches located in Brooklyn.
   $$\{c | \forall x, y, z(\neg((x, y, z) \in \text{branch})$$
   $$\lor z \neq \text{Brooklyn} \lor \exists a, n((x, a, c, n) \in \text{deposit}))\}$$

   If you find this example difficult to understand, try rewriting this expression using implication, as in the tuple relational calculus example. Here’s my attempt:

   $$\{c | \forall bn, as, bc$$
   $$((bn, as, bc) \in \text{branch} \land bc = \text{Brooklyn}) \Rightarrow \exists acct, bal((bn, acct, en, bal) \in \text{deposit}))\}$$

   I’ve used two letter variable names to get away from the problem of having to remember what $x$ stands for.

3.4.3 Safety of Expressions

1. As in the tuple relational calculus, it is possible to generate infinite expressions. The solution is similar for domain relational calculus—restrict the form to safe expressions involving values in the domain of the formula.

   Read the text for a complete explanation.

3.4.4 Expressive Power of Languages

1. All three of the following are equivalent:
   - The relational algebra.
   - The tuple relational calculus restricted to safe expressions.
   - The domain relational calculus restricted to safe expressions.
3.5. **Extended Relational-Algebra Operations**

3.5.1 **Generalized Projection**

1. Generalized projection extends the projection operation by allowing arithmetic functions to be used in the projection list.

   \[ \Pi_{F_1, F_2, \ldots, F_n}(E). \]

   where each of \( F_i \) is arithmetic expressions involving constants and attributes in the schema of the relational-algebra expression \( E \).

2. Example. Given a relation `credit_info(enamelimit, credit_balance)`, to find how much more each person may spend, we have

   \[ \Pi_{ename, limit \_credit \_balance}(credit \_info). \]

3.5.2 **Outer join**

1. Outer join: An extension of join to deal with missing information.

2. Two relations in Fig. 3.8, with the relation schemas,

   - `employee (ename, street, city)`
   - `ft\_work (ename, bname, salary)`

3. A join may miss some information on the non-joinable attributes.

4. Three outer-joins: *left outer-join*, *right outer-join*, and *full outer-join*.

5. *Left outer-join*: takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation, and adds them to the result of the natural join.

6. Similarly, we define *right outer-join* and *full outer-join*.
3.5.3 Aggregate functions

1. Aggregate functions: sum, avg, count, min, max.

2. The collections on which aggregate functions are applied can have multiple occurrences of a value: the order in which the value appears is irrelevant. Such collections are called multiset. For example,

\[ \text{sum}_{\text{salary}}(\text{pt} \_\text{works}) \]

3. To eliminate multiple occurrences of a value prior to computing an aggregate function, with the addition of the hyphenated string distinct appended to the end of the function name. For example,

\[ \text{count-distinct}_{\text{name}}(\text{pt} \_\text{works}) \]

4. Grouping and then aggregating: To find the total salary sum of all part-time employees at each branch (not the entire bank!),

\[ \text{bname} \cdot \text{count-distinct}_{\text{salary}}(\text{pt} \_\text{works}) \]

5. The general form of the aggregation operation \( G \) is as follows.

\[ G_{G_1, G_2, \ldots, G_n} F_1 A_1, F_2 A_2, \ldots, F_m A_m (E) \]

where \( E \) is any relational-algebra expression, \( G_1, G_2, \ldots, G_n \) constitute a list of attributes on which to group, each \( F_i \) is an aggregate functions, and each \( A_i \) is an attribute name.

The meaning of the operation:

- The tuples in result of expression \( E \) is partitioned into groups. All tuples in a group has the same values for \( G_1, G_2, \ldots, G_n \), and tuples in different group have different values.
- For each group \( (g_1, g_2, \ldots, g_n) \), the result has a tuple \( (g_1, g_2, \ldots, g_n, a_1, a_2, \ldots, a_m) \) where, for each \( i \), \( a_i \) is the result of applying the aggregate function \( F_i \) on the multiset of values for attribute \( A_i \) in the group.

6. Example. Find sum and max of salary for part-time employees at each branch.

\[ \text{bname} \cdot G \text{sum}_{\text{salary}, \text{max}}_{\text{salary}}(\text{pt} \_\text{works}) \]

3.6 Modification of the Database

1. Up until now, we have looked at extracting information from the database. We also need to add, remove and change information. Modifications are expressed using the assignment operator.
3.6. MODIFICATION OF THE DATABASE

3.6.1 Deletion

1. **Deletion** is expressed in much the same way as a query. Instead of displaying, the selected tuples are removed from the database. We can only delete whole tuples.

   In relational algebra, a deletion is of the form
   \[ r = r - E \]
   where \( r \) is a relation and \( E \) is a relational algebra query. Tuples in \( r \) for which \( E \) is true are deleted.

2. Some examples:
   
   1. Delete all of Smith’s account records.
      
      \[ \text{deposit} \leftarrow \text{deposit} - \sigma_{\text{ename} = \text{"Smith"}}(\text{deposit}) \]
   
   2. Delete all loans with loan numbers between 1300 and 1500.
      
      \[ \text{deposit} \leftarrow \text{deposit} - \sigma_{\text{loan} \geq 1300 \land \text{loan} \leq 1500}(\text{deposit}) \]
   
   3. Delete all accounts at Branches located in Needham.
      
      \[
      \begin{align*}
      r_1 & \leftarrow \sigma_{\text{bcity} = \text{"Needham"}}(\text{deposit} \bowtie \text{branch}) \\
      r_2 & \leftarrow \Pi_{\text{bname,account}, \text{cname,balance}}(r_1) \\
      \text{deposit} & \leftarrow \text{deposit} - r_2
      \end{align*}
      \]

3.6.2 Insertion

1. To insert data into a relation, we either specify a tuple, or write a query whose result is the set of tuples to be inserted. Attribute values for inserted tuples must be members of the attribute’s domain.

2. An insertion is expressed by
   \[ r \leftarrow r \cup E \]
   where \( r \) is a relation and \( E \) is a relational algebra expression.

3. Some examples:
   
   1. To insert a tuple for Smith who has $1200 in account 9372 at the SFU branch.
      
      \[ \text{deposit} \leftarrow \text{deposit} \cup \{ (\text{"SFU"}, 9372, \text{"Smith"}, 1200) \} \]
   
   2. To provide all loan customers in the SFU branch with a $200 savings account.
      
      \[
      \begin{align*}
      r_1 & \leftarrow (\sigma_{\text{bname} = \text{"SFU"}}(\text{borrow})) \\
      r_2 & \leftarrow \Pi_{\text{bname,loan}, \text{cname}}(r_1) \\
      \text{deposit} & \leftarrow \text{deposit} \cup (r_2 \times \{(200)\})
      \end{align*}
      \]

3.6.3 Updating

1. Updating allows us to change some values in a tuple without necessarily changing all.

   We use the update operator, \( \delta \), with the form
   \[ \delta_{A - E}(r) \]
where \( r \) is a relation with attribute \( A \), which is assigned the value of expression \( E \).

The expression \( E \) is any arithmetic expression involving constants and attributes in relation \( r \).

Some examples:

1. To increase all balances by 5 percent.

\[
\delta_{\text{balance}} = \text{balance} + 0.05 \times (\text{deposit})
\]

This statement is applied to every tuple in \( \text{deposit} \).

2. To make two different rates of interest payment, depending on balance amount:

\[
\delta_{\text{balance}} = \text{balance} + 0.05 \times (\sigma_{\text{balance} > 10000}(\text{deposit}))
\]

\[
\delta_{\text{balance}} = \text{balance} + 0.05 \times (\sigma_{\text{balance} \leq 10000}(\text{deposit}))
\]

Note: in this example the order of the two operations is important. (Why?)

### 3.7 Views

1. We have assumed up to now that the relations we are given are the actual relations stored in the database.

2. For security and convenience reasons, we may wish to create a personalized collection of relations for a user.

3. We use the term view to refer to any relation, not part of the conceptual model, that is made visible to the user as a “virtual relation”.

4. As relations may be modified by deletions, insertions and updates, it is generally not possible to store views. (Why?) Views must then be recomputed for each query referring to them.

#### 3.7.1 View Definition

1. A view is defined using the \texttt{create view} command:

\[
\texttt{create view } v \texttt{ as } (\text{query expression})
\]

where \( (\text{query expression}) \) is any legal query expression. The view created is given the name \( v \).

2. To create a view \texttt{all\_customer} of all branches and their customers:

\[
\texttt{create view all\_customer as}
\]

\[
\Pi_{\text{bname, cname}}(\text{deposit}) \cup \Pi_{\text{bname, cnamen}}(\text{borrow})
\]

3. Having defined a view, we can now use it to refer to the virtual relation it creates. View names can appear anywhere a relation name can.

4. We can now find all customers of the SFU branch by writing

\[
\Pi_{\text{cnamen}}(\sigma_{\text{bname} = \text{SFU}}(\text{all\_customer}))
\]

#### 3.7.2 Updates Through Views and Null Values

1. Updates, insertions and deletions using views can cause problems. The modifications on a view must be transformed to modifications of the actual relations in the conceptual model of the database.

2. An example will illustrate: consider a clerk who needs to see all information in the \texttt{borrow} relation except \textit{amount}. Let the view \texttt{loan\,-\,info} be given to the clerk:
3. Since SQL allows a view name to appear anywhere a relation name may appear, the clerk can write:
\[
\text{loan-info} \leftarrow \text{loan-info} \cup \{(\text{"SFU"},3,\text{"Ruth"})\}
\]
This insertion is represented by an insertion into the actual relation \text{borrow}, from which the view is constructed.

However, we have no value for \text{amount}. A suitable response would be
- Reject the insertion and inform the user.
- Insert \((\text{"SFU"},3,\text{"Ruth"},\text{null})\) into the relation.

The symbol \text{null} represents a null or placeholder value. It says the value is unknown or does not exist.

4. Another problem with modification through views: consider the view
\[
\text{create view branch-city as}
\Pi_{\text{branch},\text{city}}(\text{borrow} \bowtie \text{customer})
\]
This view lists the cities in which the borrowers of each branch live. Now consider the insertion
\[
\text{branch-city} \leftarrow \text{branch-city} \cup \{(\text{"Brighton"},\text{"Woodside"})\}
\]
Using nulls is the only possible way to do this (see Figure 3.22 in the textbook).

If we do this insertion with nulls, now consider the expression the view actually corresponds to:
\[
\Pi_{\text{branch},\text{city}}(\text{borrow} \bowtie \text{customer})
\]

As comparisons involving nulls are always \text{false}, this query misses the inserted tuple.

To understand why, think about the tuples that got inserted into \text{borrow} and \text{customer}. Then think about how the view is recomputed for the above query.

### 3.7.3 Views Defined Using Other Views

1. Views can be defined using other views. E.g.,

\[
\text{create view sfu-customer as}
\Pi_{\text{name}}(\sigma_{\text{name}='\text{SFU'}}(\text{all-customer}))
\]
where \text{all-customer} itself is a view.

2. Dependency relationship of views:
- A view \(v_1\) is said to \text{depend directly on} \(v_2\) if \(v_2\) is used in the expression defining \(v_1\).
- A view \(v_1\) is said to \text{depend on} \(v_2\) if and only if there is a path in the dependency graph from \(v_2\) to \(v_1\).
- A view relation \(v\) is said to be \text{recursive} if it depends on itself.

3. View expansion: a way to define the meaning of views in terms of other views.

\[
\text{repeat}
\begin{align*}
&\text{Find any view relation } v_i \text{ in } c_1 \\
&\text{Replace any view relation } v_i \text{ by the expression defining } v_i
\end{align*}
\text{until no more view relations are present in } c_1
\]

As long as the view definition are not recursive, this loop will terminate.