Chapter 6

Relational Database

6.1 Introduction

You can now purchase a music player that stores nearly 10,000 songs. The storage medium is a tiny hard disk, a marvel of hardware engineering. Equally impressive is the software which combines many aspects of compression, error correction and detection, and database manipulation.

First, the compression algorithm manages to store around 300 music CDs, each with around 600MB of storage, on my 20GB player; this is a compression of about 10 to 1. While it is possible to compress music to any extent, because exact reproduction is not expected, you would not want to listen to such music. Try listening to a particularly delicate piece over the telephone! The compression algorithm manages to reproduce music reasonably faithfully.

A music player begins its life expecting harsh treatment, even torture. The devices are routinely dropped, they are subjected to X-ray scans at airports, and left outside in very cold or very hot cars. Yet, the hardware is reasonably resilient, but more impressively, the software works around the hardware glitches using error-correcting strategies some of which we have outlined in an earlier chapter.

The question that concerns us in this chapter is how to *organize* a large number of songs so that we can locate a set of songs quickly. The songs are first stored on a desktop (being imported from a CD or over the internet from a music store); they can be organized there and then downloaded to a player. A naive organization will make it quite frustrating to find that exact song in your player. And, you may wish to listen to all songs which are either by artist A or composer B, in the classical genre, and have not been played more than 6 times in the last 3 months. The subject matter of this chapter is organization of certain kinds of data, like songs, to allow efficient selection of a subset which meets a given search criterion.

For many database applications a set of tuples, called a *table*, is often the appropriate data structure. Let me illustrate it with a small database of movies;

Title	Actor	Director	Genre	Year
Jurassic Park	Jeff Goldblum	Steven Spielberg	Action	1993
Jurassic Park	Sam Neill	Steven Spielberg	Action	1993
Men in Black	Tommy Lee Jones	Barry Sonnenfeld	SciFi	1997
Men in Black	Will Smith	Barry Sonnenfeld	SciFi	1997
Independence Day	Will Smith	Roland Emmerich	SciFi	1996
Independence Day	Bill Pullman	Roland Emmerich	SciFi	1996
My Fair Lady	Audrey Hepburn	George Cukor	Classics	1964
My Fair Lady	Rex Harrison	George Cukor	Classics	1964
The Sound of Music	Julie Andrews	Robert Wise	Classics	1965
The Sound of Music	Christopher Plummer	Robert Wise	Classics	1965
Bad Boys II	Martin Lawrence	Michael Bay	Action	2003
Bad Boys II	Will Smith	Michael Bay	Action	2003
Ghostbusters	Bill Murray	Ivan Reitman	Comedy	1984
Ghostbusters	Dan Aykroyd	Ivan Reitman	Comedy	1984
Tootsie	Dustin Hoffman	Sydney Pollack	Comedy	1982
Tootsie	Jessica Lange	Sydney Pollack	Comedy	1982

Table 6.1: A list of movies arranged in a table

Title	Actor	Director	Genre	Year
Men in Black	Will Smith	Barry Sonnenfeld	SciFi	1997
Independence Day	Will Smith	Roland Emmerich	SciFi	1996

Table 6.2: Result of selection on Table 6.1 (page 180)

see Table 6.1 (page 180). We store the following information for each movie: its title, actor, director, genre and the year of release. We list only the two most prominent actors for a movie, and they have to appear in different tuples; so each movie is being represented by two tuples in the table. We can now easily specify a search criterion such as, find all movies released between 1980 and 2003 in which Will Smith was an actor and the genre is SciFi. The result of this search is a table, shown in Table 6.2 (page 180).

Chapter Outline We introduce the table data structure and some terminology in section 6.2. A table resembles a mathematical relation, though there are some significant differences which we outline in that section. An algbra of relations is developed in section 6.3. The algebra consists of a set of operations on relations (section 6.3.1) and a set of identities over relational expressions (section 6.3.2). The identities are used to process queries efficiently, as shown in section 6.3.3. A standard query language, SQL, is described in section 6.3.4. This chapter is a very short introduction to the topic; for more thorough treatment see the relevant chapters in [35] and [2].

6.2 The Relational Data Model

Central to the relational data model is the concept of *relation*. You are familiar with relations from algebra, which I briefly review below. Next, I will explain relations in databases, which are slightly different.

6.2.1 Relations in Mathematics

The $>$ operator over positive integers is a (binary) relation. We write $5 > 3$, using the relation as an infix operator. More formally, the relation $>$ is a set of pairs:

$$\{(2,1),\ (3,1),\ (3,2),\ \cdots\}$$

A general relation consists of tuples, not necessarily pairs as for binary relations. Consider a *family* relation which consists of triples (c, f, m), where c is the name of a child, and f and m are the father and the mother. Or, the relation $Pythagoras$ which consists of triples (x, y, z) where the components are positive integers and $x^2 + y^2 = z^2$. Or, $Fermat$ which consists of quadruples of positive integers (x, y, z, n), where $x^n + y^n = z^n$ and $n > 2$. (A recent breakthrough in mathematics has established that $Fermat = \phi$.) In databases, the relations need not be binary; in fact, most often, they are not binary.

A relation, being a set, has all the set operations defined on it. We list some of the set operations below which are used in relational algebra.

1. Union: $R \cup S \ = \ \{x |\ x \in R \ \vee \ x \in S\}$
2. Intersection: $R \cap S \ = \ \{x |\ x \in R \ \wedge \ x \in S\}$
3. Difference: $R - S \ = \ \{x |\ x \in R \ \wedge \ x \notin S\}$
4. Cartesian Product: $R \times S \ = \ \{(x, y) |\ x \in R \ \wedge \ y \in S\}$

Thus, given $R \ = \ \{(1,2),(2,3),(3,4)\}$ and $S \ = \ \{(2,3),(3,4),(4,5)\}$, we get

$$\begin{array}{rcl} R \cup S & = & \{(1,2),(2,3),(3,4),(4,5)\} \\ R \cap S & = & \{(2,3),(3,4)\} \\ R - S & = & \{(1,2)\} \\ R \times S & = & \{((1,2),(2,3)),((1,2),(3,4)),((1,2),(4,5)), \\ & & \ ((2,3),(2,3)),((2,3),(3,4)),((2,3),(4,5)), \\ & & \ ((3,4),(2,3)),((3,4),(3,4)),((3,4),(4,5))\} \end{array}$$

In algebra, you have seen *reflexive*, *symmetric*, *asymmetric* and *transitive* binary relations. None of these concepts is of any use in relational algebra.

Year	Genre	Title	Director	Actor
1997	SciFi	Men in Black	Barry Sonnenfeld	Will Smith
1996	SciFi	Independence Day	Roland Emmerich	Will Smith

Table 6.3: A column permutation of Table 6.2 (page 180)

Theatre	Address
General Cinema	2901 S 360
Tinseltown USA	5501 S I.H. 35
Dobie Theater	2021 Guadalupe St
Entertainment Film	6700 Middle Fiskville Rd

Table 6.4: Theatres and their addresses

6.2.2 Relations in Databases

Database relations are inspired by mathematical relations. A database relation is best represented by a matrix, called a *table*, in which (1) each row is a tuple and (2) each column has a name, which is an *attribute* of the relation. Table 6.1 (page 180) shows such a relation; it has 5 attributes: Title, Actor, Director, Genre, Year. There are 16 rows, each is a tuple of the relation.

In both mathematical and database relations, the tuples are distinct and they may appear in any order. The type of an attribute, i.e., the type of values that may appear in that column, is called the *domain* of the attribute. The name of a database relation along with the names and domains of attributes is called a *relational schema*. A schema is a template; an instance of the schema has a number of tuples which fit the template.

The most fundamental difference between mathematical and database relations is that in the latter the columns can be permuted arbitrarily keeping the same relation. Thus, Table 6.2 (page 180) and Table 6.3 (page 182) represent the same relation. Therefore, we have the identity (we explain $R \times S$, the cartesian product of database relations R and S, in section 6.3.1).

$$R \times S \;=\; S \times R$$

For mathematical relations, this identity does not hold because the components cannot be permuted.

A relational database is a set of relations with distinct relation names. The relations in Tables 6.1 (page 180), 6.4 (page 182), and 6.5 (page 183) make up a relational database. Typically, every relation in a database has a common attribute with some other relation.

Theatre	Title	Time	Rating
General Cinema	Jurassic Park	Sat, 9PM	G
General Cinema	Men in Black	Sat, 9PM	PG
General Cinema	Men in Black	Sun, 3PM	PG
Tinseltown USA	Independence Day	Sat, 9PM	PG-13
Dobie Theater	My Fair Lady	Sun, 3PM	G
Entertainment Film	Ghostbusters	Sun, 3PM	PG-13

Table 6.5: Theatres, Movies, Time and Rating

6.3 Relational Algebra

An algebra consists of (1) elements, (2) operations and (3) identities. For example, to do basic arithmetic over integers we define: (1) elements to be integers, (2) operations to be $+$, $-$, $\times$, $\div$, and (3) identities such as,

$$x + y = y + x$$
$$x \times (y + z) = x \times y + x \times z$$

where x, y and z range over the elements (i.e., integers).

We define an algebra of database relations in this section. The elements are database relations. We define a number of operations on them in section 6.3.1 and several identities in section 6.3.2.

6.3.1 Operations on Database Relations

Henceforth, R, S and T denote relations, and a and b are sets of attributes. Relations R and S are *union-compatible*, or just *compatible*, if they have the same set of attributes.

Union $R \cup S$ is the union of *compatible* relations R and S. Relation $R \cup S$ includes all tuples from R and S with duplicates removed.

Intersection $R \cap S$ is the intersection of *compatible* relations R and S. Relation $R \cap S$ includes all tuples which occur in both R and S.

Difference $R-S$ is the set difference of *compatible* relations R and S. Relation $R - S$ includes all tuples which are in R and not in S.

Cartesian Product or Cross Product $R \times S$ is the cross product of relations R and S. The relations need not be compatible. Assume for the moment that the attributes of R and S are disjoint. The set of attributes of $R \times S$ are the ones from both R and S. Each tuple of R is concatenated with each tuple of S to form tuples of $R \times S$. Two database relations are shown in Table 6.6

Title	Actor	Director	Year
Jurassic Park	Sam Neill	Steven Spielberg	1993
Men in Black	Tommy Lee Jones	Michael Bay	2003
		Ivan Reitman	1984

Table 6.6: Two relations separated by vertical line

Title	Actor	Director	Year
Jurassic Park	Sam Neill	Steven Spielberg	1993
Jurassic Park	Sam Neill	Michael Bay	2003
Jurassic Park	Sam Neill	Ivan Reitman	1984
Men in Black	Tommy Lee Jones	Steven Spielberg	1993
Men in Black	Tommy Lee Jones	Michael Bay	2003
Men in Black	Tommy Lee Jones	Ivan Reitman	1984

Table 6.7: Cross Product of the two relations in Table 6.6 (page 184)

(page 184); they are separated by a vertical line. Their cross product is shown in Table 6.7 (page 184).

The cross product in Table 6.7 makes no sense. We introduce the *join* operator later in this section which takes a more "intelligent" cross product.

If R and S have common attribute names, the names are changed so that we have disjoint attributes. One strategy is to prefix the attribute name by the name of the relation. So, if you are computing *Prof* $\times$ *Student* where both *Prof* and *Student* have an attribute *id*, an automatic renaming may create *Profid* and *Studentid*. This does not always work, for instance, in *Prof* $\times$ *Prof*. Manual aid is then needed. In this chapter, we write $R \times S$ only if the attributes of R and S are disjoint.

Note a subtle difference between mathematical and database relations for cross product. For tuple (r, s) in R and (u, v) in S, their mathematical cross product gives a tuple of tuples, $((r, s), (u, v))$, whereas the database cross product gives a tuple containing all 4 elements, (r, s, u, v).

The number of tuples in $R \times S$ is the number of tuples in R times the number in S. Thus, if R and S have 1,000 tuples each, $R \times S$ has a million tuples and $R \times (S \times S)$ has a billion. So, cross product is rarely computed in full. It is often used in conjunction with other operations which can be applied in a clever sequence to eliminate explicit computations required for a cross product.

Projection The operations we have described so far affect only the rows (tuples) of a table. The next operation, *projection*, specifies a set of attributes of a relation that are to be retained to form a relation. Projection removes all other attributes (columns), and removes any duplicate rows that are created as a result. We write $\pi_{u,v}(R)$ to denote the relation which results by retaining only the attributes u and v of R. Let R be the relation shown in Table 6.1

(page 180). Then, $\pi_{Title, Director, Genre, Year}(R)$ gives Table 6.9 (page 186) and $\pi_{Title, Actor}(R)$ gives Table 6.10 (page 186).

Selection The selection operation chooses the tuples of a relation that satisfy a specified predicate. A predicate uses attribute names as variables, as in $year \geq 1980 \wedge year \leq 2003 \wedge actor = \text{“}Will\ Smith\text{”} \wedge genre = \text{“}SciFi\text{”}$. A tuple satisfies a predicate if the predicate is *true* when the attribute names are replaced by the corresponding values from the tuple. We write $\sigma_p(R)$ to denote the relation consisting of the subset of tuples of R that satisfy predicate p. Let R be the relation in Table 6.1 (page 180). Then, $\sigma_{year \geq 1980 \wedge year \leq 2003 \wedge actor = \text{“}Will\ Smith\text{”} \wedge genre = \text{“}SciFi\text{”}}(R)$ is shown in Table 6.2 (page 180) and $\sigma_{actor = \text{“}Will\ Smith\text{”} \wedge genre = \text{“}Comedy\text{”}}(R)$ is the empty relation.

Join There are several *join* operators in relational algebra. We study only one which is called *natural join*, though we simply call it *join* in this chapter. The join of R and S is written as $R \bowtie S$. Here, R and S need not be compatible; typically, they will have some common attributes.

The join is a more refined way of taking the cross product. As in the cross product, take each tuple r of R and s of S. If r and s have no common attributes, or do not match in their common attributes, then their join produces an empty tuple. Otherwise, concatenate r and s keeping only one set of values for the common attributes (which match). Consider Tables 6.4 (page 182) and 6.5 (page 183). Their join is shown in Table 6.8 (page 185). And, the join of Tables 6.9 (page 186) and 6.10 (page 186) is Table 6.1 (page 180).

Theatre	Title	Time	Rating	Address
General Cinema	Jurassic Park	Sat, 9PM	G	2901 S 360
General Cinema	Men in Black	Sat, 9PM	PG	2901 S 360
General Cinema	Men in Black	Sun, 3PM	PG	2901 S 360
Tinseltown USA	Independence Day	Sat, 9PM	PG-13	5501 S I.H. 35
Dobie Theater	My Fair Lady	Sun, 3PM	G	2021 Guadalupe St
Entertainment Film	Ghostbusters	Sun, 3PM	PG-13	6700 Middle Fiskville

Table 6.8: Join of Tables 6.4 and 6.5

If R and S have no common attributes, we see that $R \bowtie S$ is an empty relation, though it has all the attributes of R and S. We will avoid taking $R \bowtie S$ if R and S have no common attributes.

Writing $attr(R)$ for the set of attributes of R, we have

$$attr(R \bowtie S) = attr(R) \cup attr(S), \text{ and}$$
$$x \in R \bowtie S \equiv (attr(R) \cap attr(S) \neq \phi) \wedge \pi_{attr(R)}(x) \in R \wedge \pi_{attr(S)}(x) \in S$$

The condition $attr(R) \cap attr(S) \neq \phi$, i.e., R and S have a common attribute, is essential. Without this condition, $R \bowtie S$ would be $R \times S$ in case the attributes are disjoint.

Title	Director	Genre	Year
Jurassic Park	Steven Spielberg	Action	1993
Men in Black	Barry Sonnenfeld	SciFi	1997
Independence Day	Roland Emmerich	SciFi	1996
My Fair Lady	George Cukor	Classics	1964
The Sound of Music	Robert Wise	Classics	1965
Bad Boys II	Michael Bay	Action	2003
Ghostbusters	Ivan Reitman	Comedy	1984
Tootsie	Sydney Pollack	Comedy	1982

Table 6.9: Compact representation of a portion of Table 6.1 (page 180)

The join operator selects only the tuples which match in certain attributes; so, join results in a much smaller table than the cross product. Additionally, the result is usually more meaningful. In many cases, a large table can be decomposed into two much smaller tables whose join recreates the original table. See the relations in Tables 6.9 (page 186) and 6.10 (page 186) whose join gives us the relation in Table 6.1. The storage required for these two relations is much smaller than that for Table 6.1 (page 180).

Title	Actor
Jurassic Park	Jeff Goldblum
Jurassic Park	Sam Neill
Men in Black	Tommy Lee Jones
Men in Black	Will Smith
Independence Day	Will Smith
Independence Day	Bill Pullman
My Fair Lady	Audrey Hepburn
My Fair Lady	Rex Harrison
The Sound of Music	Julie Andrews
The Sound of Music	Christopher Plummer
Bad Boys II	Martin Lawrence
Bad Boys II	Will Smith
Ghostbusters	Bill Murray
Ghostbusters	Dan Aykroyd
Tootsie	Dustin Hoffman
Tootsie	Jessica Lange

Table 6.10: Table 6.1 (page 180) arranged by Title and Actor

Exercise 80

Suppose R and S are compatible. Show that $R \bowtie S = R \cap S$.

6.3.2 Identities of Relational Algebra

We develop a number of identities in this section. I don't prove the identities; I recommend that you do. These identities are used to transform a relational expression into an equivalent form whose evaluation is more efficient, a procedure known as query optimization. Query optimization can reduce evaluation time of relational expressions by several orders of magnitude. In the following, R , S and T denote relations, a and b are sets of attributes, and p and q are predicates.

1. (Selection splitting) $\sigma_{p \wedge q}(R) \ = \ \sigma_p(\sigma_q(R))$

2. (Commutativity of selection)

   $$\sigma_p(\sigma_q(R)) \ = \ \sigma_q(\sigma_p(R))$$

   This is a corollary of Selection splitting given above.

3. (Projection refinement) Let a and b be subsets of attributes of relation R, and $a \subseteq b$. Then,

   $$\pi_a(R) \ = \ \pi_a(\pi_b(R))$$

4. (Commutativity of selection and projection) Given that p names only attributes in a,

   $$\pi_a(\sigma_p(R)) \ = \ \sigma_p(\pi_a(R))$$

5. (Commutativity and Associativity of union, cross product, join)

   $R \cup S \ = \ S \cup R$
   $(R \cup S) \cup T \ = \ R \cup (S \cup T)$
   $R \times S \ = \ S \times R$
   $(R \times S) \times T \ = \ R \times (S \times T)$
   $R \bowtie S \ = \ S \bowtie R$
   $(R \bowtie S) \bowtie T \ = \ R \bowtie (S \bowtie T)$,
   provided R and S have common attributes and so do S and T, and no attribute is common to all three relations.

6. (Selection pushing)

   $\sigma_p(R \cup S) \ = \ \sigma_p(R) \cup \sigma_p(S)$
   $\sigma_p(R \cap S) \ = \ \sigma_p(R) \cap \sigma_p(S)$
   $\sigma_p(R - S) \ = \ \sigma_p(R) - \sigma_p(S)$

   Suppose predicate p names only attributes of R. Then,

$$\sigma_p(R \times S) \;=\; \sigma_p(R) \times S$$
$$\sigma_p(R \bowtie S) \;=\; \sigma_p(R) \bowtie S$$

7. (Projection pushing)

$$\pi_a(R \cup S) \;=\; \pi_a(R) \cup \pi_a(S)$$

8. (Distributivity of projection over join)

$$\pi_a(R \bowtie S) \;=\; \pi_a(\pi_b(R) \bowtie \pi_c(S))$$

where R and S have common attributes d, a is a subset of attributes of both R and S, b is a's subset from R plus d and c is a's subset from S plus d. That is,

$$a \subseteq attr(R) \;\cup\; attr(S)$$
$$b = (a \cap attr(R)) \;\cup\; d$$
$$c = (a \cap attr(S)) \;\cup\; d$$
$$d = attr(R) \;\cap\; attr(S)$$ □

Selection splitting law says that evaluations of $\sigma_{p \wedge q}(R)$ and $\sigma_p(\sigma_q(R))$ are interchangeable; so, apply either of the following procedures: look at each tuple of R and decide if it satisfies $p \wedge q$, or first identify the tuples of R which satisfy q and from those identify the ones which satisfy p. The benefit of one strategy over another depends on the relative costs of access times to the tuples and predicate evaluation times. For large databases, which are stored in secondary storage, access time is the major cost. Then it is preferable to evaluate $\sigma_{p \wedge q}(R)$.

It is a good heuristic to apply projection and selection to as small a relation as possible. Therefore, it is almost always better to evaluate $\sigma_p(R) \bowtie S$ instead of $\sigma_p(R \bowtie S)$, i.e., apply selection to R which tends to be smaller than $R \bowtie S$. Similarly, distributivity of projection over join is often used in query optimizations.

Exercise 81

Suppose predicate p names only the attributes of S. Show that $\sigma_p(R \bowtie S) \;=\; R \bowtie \sigma_p(S)$.

Exercise 82

Show that $\pi_a(R \cap S) \;=\; \pi_a(R) \cap \pi_a(S)$ does not necessarily hold.

6.3.3 Example of Query Optimization

We consider the relations in Tables 6.1 (page 180), 6.5 (page 183), and 6.4 (page 182). We call these relations R, S and T, respectively. Relation R is prepared by some movie distribution agency independent of the theatres; theatre owners in Austin compile the databases S and T. Note that T is relatively stable.

We would like to know the answer to: What are the addresses of theatres where Will Smith is playing on Saturday at 9PM. We write a relational expression for this query and then transform it in several stages to a form which can be efficiently evaluated. Let predicates

p be $Actor =$ Will Smith
q be $Time =$ Sat, 9PM

The query has the form $\pi_{Address}(\sigma_{p \wedge q}(x))$, where x is a relation yet to be defined. Since x has to include information about $Actor$, $Time$ and $Address$, we take x to be $R \bowtie S \bowtie T$. Relation x includes many more attributes than the ones we desire; we will project away the unneeded attributes. The selection operation extracts the tuples which satisfy the predicate $p \wedge q$, and then the projection operation simply lists the addresses. So, the entire query is

$$\pi_{Address}(\sigma_{p \wedge q}\langle R \bowtie S \bowtie T\rangle)$$

Above and in the following expressions, we use brackets of different shapes to help readibility.

We transform this relational expression.

$$\begin{array}{ll} & \pi_{Address}(\sigma_{p \wedge q}\langle R \bowtie S \bowtie T\rangle) \\ \equiv & \{\text{Associativity of join; note that the required conditions are met}\} \\ & \pi_{Address}(\sigma_{p \wedge q}\langle (R \bowtie S) \bowtie T\rangle) \\ \equiv & \{\text{Selection pushing over join}\} \\ & \pi_{Address}(\sigma_{p \wedge q}\langle R \bowtie S\rangle \bowtie T) \\ \equiv & \{\text{See lemma below. } p \text{ names only the attributes of } R \text{ and } q \text{ of } S\} \\ & \pi_{Address}(\langle \sigma_p(R) \bowtie \sigma_q(S)\rangle \bowtie T) \\ \equiv & \{\text{Distributivity of projection over join; } d = \{Theatre\}\} \\ & \pi_{Address}(\pi_{Theatre}\langle \sigma_p(R) \bowtie \sigma_q(S)\rangle \bowtie \pi_{Address, Theatre}(T)) \\ \equiv & \{\pi_{Address, Theatre}(T) = T\} \\ & \pi_{Address}(\pi_{Theatre}\langle \sigma_p(R) \bowtie \sigma_q(S)\rangle \bowtie T) \\ \equiv & \{\text{Distributivity of projection over join;} \\ & \quad \text{the common attribute of } \sigma_p(R) \text{ and } \sigma_q(S) \text{ is } Title\} \\ & \pi_{Address}(\langle \pi_{Title}(\sigma_p(R)) \bowtie \pi_{Theatre, Title}(\sigma_q(S))\rangle \bowtie T) \end{array}$$

Lemma Suppose predicate p names only the attributes of R and q of S. Then,

$$\sigma_{p \wedge q}(R \bowtie S) \;=\; \sigma_p(R) \bowtie \sigma_q(S)$$

Proof:

$$\begin{array}{ll} & \sigma_{p \wedge q}(R \bowtie S) \\ \equiv & \{\text{Selection splitting}\} \\ & \sigma_p\langle \sigma_q(R \bowtie S)\rangle \\ \equiv & \{\text{Commutativity of join}\} \\ & \sigma_p\langle \sigma_q(S \bowtie R)\rangle \\ \equiv & \{\text{Selection pushing over join}\} \end{array}$$

$$
\begin{array}{ll}
 & \sigma_p\langle\sigma_q(S) \bowtie R\rangle \\
\equiv & \{\text{Commutativity of join}\} \\
 & \sigma_p\langle R \bowtie \sigma_q(S)\rangle \\
\equiv & \{\text{Selection pushing over join}\} \\
 & \sigma_p(R) \bowtie \sigma_q(S)
\end{array}
$$

Compare the original query $\pi_{Address}(\sigma_{p\wedge q}\langle R \bowtie S \bowtie T\rangle)$ with the transformed query $\pi_{Address}(\langle\pi_{Title}(\sigma_p(R)) \bowtie \pi_{Theatre,\,Title}(\sigma_q(S))\rangle \bowtie T)$ in terms of the efficiency of evaluation. The original query would first compute $R \bowtie S \bowtie T$, a very expensive operation involving three tables. Then selection operation will go over all the tuples again, and the projection incurs a small cost. In the transformed expression, selections are applied as soon as possible, in $\sigma_p(R)$ and $\sigma_q(S)$. This results in much smaller relations, 3 tuples in $\sigma_p(R)$ and 3 in $\sigma_q(S)$. Next, projections will reduce the number of columns in both relations, though not the number of rows. The join of the resulting relation is much more efficient, being applied over smaller tables. Finally, the join with T and projection over *Address* is, again, over smaller tables.

6.3.4 Additional Operations on Relations

The operations on relations that have appeared so far are meant to move the data around from one relation to another. There is no way to compute with the data. For example, we cannot ask: How many movies has Will Smith acted in since 1996. To answer such questions we have to count (or add), and none of the operations allow that. We describe two classes of operations, *Aggregation* and *Grouping*, to do such processing. Aggregation operations combine the values in a column in a variety of ways. Grouping creates a number of subrelations from a relation based on some specified attribute values, applies a specified aggregation operation on each, and stores the result in a relation.

Aggregation The following aggregation functions are standard; all except *Count* apply to numbers. For attribute t of a relation,

Count: the number of distinct values (in t)
Sum: sum
Avg: average
Min: minimum
Max: maximum

We write $\mathbf{A}_{f\ t,\ g\ u,\ h\ v\ldots}(R)$ where f, g and h are aggregation functions (shown above) and t, u and v are attribute names in R. The result is a relation which has just one tuple, with values obtained by applying f, g and h to the values of attributes t, u and v of R, respectively. The number of columns in the result is the number of attributes chosen.

Student Id	Dept	Q1	Q2	Q3
216285932	CS	61	72	49
228544932	CS	35	47	56
859454261	CS	72	68	75
378246719	EE	70	30	69
719644435	EE	60	70	75
549876321	Bus	56	60	52

Table 6.11: Relation *Grades*

Avg Q1
59

Table 6.12: Relation *Grades*, Table 6.11, averaged on Q1

Example of Aggregation Consider the *Grades* relation in Table 6.11 (page 191).
Now $\mathbf{A}_{Avg\ Q1}(Grades)$ creates Table 6.12 (page 191).
We create Table 6.13 (page 191) by $\mathbf{A}_{Min\ Q1,\ Min\ Q2,\ Min\ Q3}(Grades)$.

Min Q1	Min Q2	Min Q3
35	30	49

Table 6.13: Min of each quiz from relation *Grades*, Table 6.11

Consider the names of the attributes in the result Table 6.13, created by $\mathbf{A}_{Min\ Q1,\ Min\ Q2,\ Min\ Q3}(Grades)$. We have simply concatenated the name of the aggregation function and the attribute in forming those names. In general, the user specifies what names to assign to each resulting attribute; we do not develop the notation for such specification here.

Grouping A grouping operation has the form ${}_g\mathbf{A}_L(R)$ where g is a group (see below) and $\mathbf{A}_L(R)$ is the aggregation (L is a list of function, attribute pairs and R is a relation). Whereas $\mathbf{A}_L(R)$ creates a single tuple, ${}_g\mathbf{A}_L(R)$ typically creates multiple tuples. The parameter g is a set of attributes of R. First, R is divided into subrelations R_0, $R_1 \cdots$, based on the attributes g; tuples in each R_i have the same values for g and tuples from different R_is have different values. Then aggregation is applied to each subrelation R_i. The resulting relation has one tuple for each R_i.

Example of Grades, contd. Compute the average score in each quiz for each department. We write ${}_{Dept}\mathbf{A}_{Avg\ Q1,\ Avg\ Q2,\ Avg\ Q3}(Grades)$ to get Table 6.14 (page 192). Count the number of students in each department whose total score exceeds 170: ${}_{Dept}\mathbf{A}_{Count,\ Student\ Id}\langle\sigma_{Q1+Q2+Q3>170}(Grades)\rangle$.

Dept	Avg Q1	Avg Q2	Avg Q3
CS	56	62	60
EE	65	50	72
Bus	56	60	52

Table 6.14: Avg of each quiz by department from relation *Grades*, Table 6.11

Query Language SQL A standard in the database community, SQL is a widely used language for data definition and manipulation. SQL statements can appear as part of a C++ program, and, also they can be executed from a command line. A popular version is marketed as MySQL.

Query facility of SQL is based on relational algebra (most SQL queries can be expressed as relational expressions). But, SQL also provides facilities to insert, delete and update items in a database.