The Relational Algebra

Relational set operators:

The data in relational tables are of limited value unless the data can be manipulated to generate useful information.

Relational Algebra defines the theoretical foundation of manipulating table content using the eight relational operators:

SELECT, PROJECT, JOIN, INTERSECT, UNION, DIFFERENCE, PRODUCT, AND DIVIDE.

The relational operators have the property of closure that is applying operators on relations produce relations.

1. SELECT, also known as RESTRICT, yields values for all the rows found in a table that satisfy a given condition. SELECT yields a horizontal subset of a table.
2. PROJECT yields all values for selected attributes. PROJECT yields a vertical subset of a table
3. UNION: combines all rows from two or more tables, excluding duplicate rows. In order to be used in a UNION, the tables must have the same attribute characteristics, that it the attributes and their domains must be compatible. When two or more tables share the same number of columns and when their corresponding columns share the same or compatible domains, they are said to be union-compatible.
4. INTERSECT: yields only the rows that appear in both tables. As with UNION, the tables must be union-compatible to yield valid results.

5. DIFFERENCE: yields all rows in one table that are not found in the other table, that is, it subtracts one table from the other. As with the UNION, the tables must be UNION-compatible to yield valid results.
6. PRODUCT: yields all possible pairs of rows from two tables- also known as Cartesian product. Therefore, if one table has six rows and the other table has three, the PRODUCT yields a list composed of 6 x 3= 18 rows.
7. JOIN: allows information to be combined from two or more tables. JOIN allows the use of independent tables linked by common attributes.
   A natural join links tables be selecting only the rows with common values in their common attributes. A natural join is the result of a three stage process:
   a. First, a PRODUCT of the tables is created
   b. Second, a SELECT is performed on the output of Step a) to yield only the rows whose values are equal.
   c. A PROJECT is performed on the results of Step b to yield a single copy of each attribute, thereby eliminating duplicate columns. The final outcome of a natural join yields a table that does not include unmatched pairs and provides only copies of the matches.

The relational algebra is very important for several reasons:

1. it provides a formal foundation for relational model operations.
2. and perhaps more important, it is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs)
3. some of its concepts are incorporated into the SQL standard query language for RDBMSs.

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level *declarative* language for specifying relational queries.
The relational algebra is often considered to be an integral part of the relational data model. Its operations include two groups:

1. Set operations from mathematical set theory; a set of tuples in the *formal* relational model and include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT (also known as CROSS PRODUCT).
2. Operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others.

Unary Relational Operations:
The SELECT Operation these are applicable because each relation is defined to be
The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**. One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition.
Alternatively, we can consider the SELECT operation to *restrict* the tuples in a relation to only those tuples that satisfy the condition.
The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are discarded.

For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:
$\sigma_{Dno=4}$(EMPLOYEE)
$\sigma_{Salary>30000}$(EMPLOYEE)

In general, the SELECT operation is denoted by

$\sigma_{<\text{selection condition}>}(R)$

where the symbol σ(sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation *R*.

Notice that *R* is generally a *relational algebra expression* whose result is a relation—the simplest such expression is just the name of a database relation. The relation resulting from the SELECT operation has the *same attributes* as *R*.

The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<attribute name> <comparison op> <constant value>

or

<attribute name> <comparison op> <attribute name>

Where:

<attribute name> is the name of an attribute of *R*,

<comparison op> is normally one of the operators {=, <, ≤, >, ≥,≠ }, and

<constant value> is a constant value from the attribute domain.

Clauses can be connected by the standard Boolean operators *and*, *or*, and *not* to form a general selection condition. For example, to select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

σ (Dno=4 **AND** Salary>25000) **OR** (Dno=5 **AND** Salary>30000)(EMPLOYEE)

Notice that all the comparison operators in the set {=, <, ≤, >, ≥, ≠ }, can apply to attributes whose domains are *ordered values*, such as numeric or date domains.

Domains of strings of characters are also considered to be ordered based on the collating sequence of the characters (concatenation).

If the domain of an attribute is a set of *unordered values*, then only the comparison operators in the set {=, ≠} can be used.
An example of an unordered domain is the domain Color = { 'red', 'blue', 'green', 'white', 'yellow'}, where no order is specified among the various colors

Some domains allow additional types of comparison operators; for example, a domain of character strings may allow the comparison operator SUBSTRING_OF.
In general, the result of a SELECT operation can be determined as follows:

-The <selection condition> is applied independently to each *individual tuple t* in *R*. This is done by substituting each occurrence of an attribute *Ai* in the selection condition with its value in the tuple *t*[*Ai*].
-If the condition evaluates to TRUE, then tuple *t* is **selected.**
-All the selected tuples appear in the result of the SELECT operation.

The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:

- (cond1 **AND** cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
- (cond1 **OR** cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
- (**NOT** cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

The SELECT operator is **unary**; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually*; hence, selection conditions cannot involve more than one tuple.

The **degree** of the relation resulting from a SELECT operation—its number of attributes—is the same as the degree of *R*.
The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in *R*. That is, $|\sigma_c (R)| \leq |R|$ for any condition *C*.

The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.

the SELECT operation is **commutative**; that is,
$\sigma_{<cond1>} (\sigma_{<cond2>} (R)) = \sigma_{<cond2>} (\sigma_{<cond1>}(R))$

Hence, a sequence of SELECTs can be applied in any order.

In addition, we can always combine a **cascade** (or **sequence**) of SELECT operations into a single
SELECT operation with a conjunctive (AND) condition; that is,
$\sigma_{<cond1>}(\sigma_{<cond2>} (...(\sigma_{<cond}n_{>} (R)) ...)) = \sigma_{<cond1>\textbf{AND}<cond2>\ \textbf{AND}...\textbf{AND}<condN>} (R)$

In SQL, the SELECT condition is typically specified in the WHERE clause of a query.
For example, the following operation:

$\sigma_{Dno=4\ \textbf{AND}\ Salary>25000}$ (EMPLOYEE)

would correspond to the following SQL query:

```
SELECT *
FROM EMPLOYEE
WHERE Dno=4 AND Salary>25000;
```

The PROJECT Operation

The **PROJECT** operation selects certain *columns* from the table and discards the other columns.

If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only.

Therefore, the result of the PROJECT operation can be visualized as a *vertical partition* of the relation into two relations:
one has the needed columns (attributes) and contains the result of the operation, and the other contains the discarded columns.

For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

$\pi_{\text{Lname, Fname, Salary}}(\text{EMPLOYEE})$

The general form of the **PROJECT** operation is

$\pi_{\text{<attribute list>}}(R)$

where π(pi) is the symbol used to represent the PROJECT operation, and <attribute list> is the desired sublist of attributes from the attributes of relation *R*.
Again, notice that *R* is, in general, a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a database relation.

The result of the PROJECT operation has only the attributes specified in <attribute list> *in the same order as they appear in the list*.

Hence, its **degree** is equal to the number of attributes in <attribute list>.

If the attribute list includes only nonkey attributes of *R*, duplicate tuples are likely to occur.

The PROJECT operation contains a key, *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**.

Duplicate elimination involves sorting or some other technique to detect duplicates and thus adds more processing.

If duplicates are not eliminated, the result would be a **multiset** or **bag** of tuples rather than a set. This was not permitted in the formal relational model, but is allowed in SQL.

For example, consider the following PROJECT operation:
$\pi_{Sex, Salary}(EMPLOYEE)$

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in *R*.

If the projection list is a superkey of *R*—that is, it includes some key of *R*—the resulting relation has the *same number* of tuples as *R*.
$\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$ as long as <list2> contains the attributes in <list1>; otherwise, the left-hand side is an incorrect expression.

It is also noteworthy that commutability *does not* hold on PROJECT.

In SQL, the PROJECT attribute list is specified in the SELECT clause of a query. For example, the following operation:
$\pi_{Sex, Salary}(EMPLOYEE)$ would correspond to the following SQL query:
SELECT DISTINCT Sex, Salary
FROM EMPLOYEE

Notice that if we remove the keyword **DISTINCT** from this SQL query, then duplicates will not be eliminated. This option is not available in the formal relational algebra.

Sequences of Operations and the RENAME Operation

In general, for most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations.

In the latter case, we must give names to the relations that hold the intermediate results.

For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation.

We can write a single relational algebra expression, also known as an **in-line expression**, as follows:

$\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$

Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno=5}}(\text{EMPLOYEE})$

$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5\_EMPS})$

It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression.

We can also use this technique to **rename** the attributes in the intermediate and result relations.

To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

TEMP$\leftarrow\sigma_{Dno=5}$(EMPLOYEE)
R(First_name, Last_name, Salary) $\leftarrow\pi_{Fname, Lname, Salary}$(TEMP)

If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order.
For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—as a unary operator.
The general RENAME operation when applied to a relation *R* of degree *n* is denoted by any of the following three forms:

1. $\rho_{S(B1, B2, ..., Bn)}(R)$
2. or $\rho_S(R)$
3. or $\rho_{(B1, B2, ..., Bn)}(R)$

where the symbol ρ (rho) is used to denote the RENAME operator, *S* is the new relation name, and *B*1, *B*2, ..., *Bn* are the new attribute names.
The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only.
If the attributes of *R* are (*A*1, *A*2, ..., *An*) in that order, then each *Ai* is renamed as *Bi*.
$\rho_{S(B1, B2, ..., Bn)}(R)$ or $\rho_S(R)$ or $\rho_{(B1, B2, ..., Bn)}(R)$
Renaming in SQL is accomplished by aliasing using **AS**, as in the following example:

SELECT E.Fname **AS** First_name, E.Lname **AS** Last_name, E.Salary **AS** Salary
FROM EMPLOYEE **AS** E **WHERE** E.Dno=5,

Relational Algebra Operations from Set Theory:

The UNION, INTERSECTION, and MINUS Operations
The next group of relational algebra operations is the standard mathematical operations on sets.
For example, to retrieve the Social Security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:

RESULT1 $\leftarrow \pi_{Ssn}$(DEP5_EMPS)
RESULT2$\leftarrow \pi_{Super_ssn}$(DEP5_EMPS)
RESULT$\leftarrow$RESULT1 $\cup$ RESULT2

The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn of all employees who directly supervise an employee who works in department 5.
The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both, while eliminating any duplicates.

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION**, **INTERSECTION**, and **SET DIFFERENCE** (also called **MINUS** or **EXCEPT**).
These are **binary** operations; that is, each is applied to two set (of tuples).When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples**; this condition has been called *union compatibility* or *type compatibility*.

Two relations $R(A1, A2, ..., An)$ and $S(B1, B2, ..., Bn)$ are said to be **union compatible** (or **type compatible**) if they have the same degree n and if $\text{dom}(Ai) = \text{dom}(Bi)$ for $1 \leq i \leq n$.

This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations R and S as follows:

- UNION: The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
- INTERSECTION: The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S.
- SET DIFFERENCE (or MINUS): The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S.

We will adopt the convention that the resulting relation has the same attribute name as the *first* relation R. It is always possible to rename the attributes in the result using the rename operator.

Notice that both UNION and INTERSECTION are *commutative operations*; that is,
$R \cup S = S \cup R$ and $R \cap S = S \cap R$
Both UNION and INTERSECTION can be treated as n-ary operations applicable to any number of relations because both are also *associative operations;* that is,
$R \cup (S \cup T) = (R \cup S) \cup T$ and $(R \cap S) \cap T = R \cap (S \cap T)$

The MINUS operation is *not commutative;* that is, in general,
$R - S \neq S - R$

Note that INTERSECTION can be expressed in terms of union and set difference as follows:
$R \cap S = ((R \cup S) - (R - S)) - (S - R)$

In SQL, there are three operations—UNION, INTERSECT, and EXCEPT—that correspond to the set operations described here.
In addition, there are multiset operations (UNION ALL, INTERSECT ALL, and EXCEPT ALL) that do not eliminate duplicates.

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation
The **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT** or **CROSS JOIN**—which is denoted by X This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible.
In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set).

In general, the result of $R(A1, A2, ..., An)$ X $S(B1, B2, ..., Bm)$ is a relation Q with degree $n + m$ attributes $Q(A1, A2, ..., An, B1, B2, ..., Bm)$, in that order.
The resulting relation Q has one tuple for each combination of tuples—one from R and one from S.

Hence, if R has nR tuples (denoted as $|R| = nR$), and S has mS tuples, then $R . S$ will have $nR * nS$ tuples.
The n-ary CARTESIAN PRODUCT operation is an extension of the above concept, which produces new tuples by concatenating all possible combinations of tuples from n underlying relations.

In general, the CARTESIAN PRODUCT operation applied by itself is generally meaningless. It is mostly useful when followed by a selection

that matches values of attributes coming from the component relations.

For example, suppose that we want to retrieve a list of names of each female employee's dependents with the employee first name and last name. We can do this as follows:

FEMALE_EMPS $\leftarrow \sigma_{Sex='F'}$(EMPLOYEE)

EMPNAMES $\leftarrow \pi_{Fname, Lname, Ssn}$(FEMALE_EMPS)

EMP_DEPENDENTS $\leftarrow$EMPNAMES.DEPENDENT

ACTUAL_DEPENDENTS $\leftarrow \sigma_{Ssn=Essn}$(EMP_DEPENDENTS)

RESULT $\leftarrow \pi_{Fname, Lname, Dependent\_name}$(ACTUAL_DEPENDENTS)

The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMPNAMES with DEPENDENT.

For example, suppose that we want to retrieve a list of names of each female employee's dependents with the employee first name and last name. We can do this as follows:

Original EMPLOYEE table:

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston,	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston	M	55000	null	1

FEMALE_EMPS $\leftarrow \sigma_{Sex='F'}$(EMPLOYEE)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston	F	25000	333445555	5

EMPNAMES $\leftarrow \pi_{Fname,\ Lname,\ Ssn}(FEMALE\_EMPS)$

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

DEPENDENT table:

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-08	Spouse
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Cartesian Product of EMPNAMES and DEPENDENT

EMP_DEPENDENTS $\leftarrow$ EMPNAMES X DEPENDENT

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	Relationship
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	Daughter
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	Son
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	Spouse
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-08	Spouse
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	Daughter
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	Spouse
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	Daughter
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	Son
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	Spouse
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-08	Spouse
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	Daughter
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	Spouse
Joyce	English	453453453	333445555	Alice	F	1986-04-05	Daughter
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	Son
Joyce	English	453453453	333445555	Joy	F	1958-05-03	Spouse
Joyce	English	453453453	987654321	Abner	M	1942-02-08	Spouse
Joyce	English	453453453	123456789	Alice	F	1988-12-30	Daughter
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	Spouse

ACTUAL_DEPENDENTS $\leftarrow \sigma_{Ssn=Essn}$**(EMP_DEPENDENTS)**

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	Relationship
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-08	Spouse

RESULT $\leftarrow \pi_{Fname, Lname, Dependent\_name}$ **(ACTUAL_DEPENDENTS)**

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

In EMP_DEPENDENTS, every tuple from EMPNAMES is combined with every tuple from DEPENDENT, giving a result that is not very meaningful (every dependent is combined with *every* female employee).

We want to combine a female employee tuple only with her particular dependents—namely, the DEPENDENT tuples whose Essn value match the Ssn value of the EMPLOYEE tuple.

The ACTUAL_DEPENDENTS relation accomplishes this. The EMP_DEPENDENTS relation is a good example of the case where relational algebra can be correctly applied to yield results that make no sense at all.

It is the responsibility of the user to make sure to apply only meaningful operations to relations.

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations.
We can SELECT *related tuples only* from the two relations by specifying an appropriate selection condition after the Cartesian product, as we did in the preceding example.

Because this sequence of CARTESIAN PRODUCT followed by SELECT is quite commonly used to combine *related tuples* from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation.
In SQL, CARTESIAN PRODUCT can be realized by using the CROSS JOIN option in joined tables. Alternatively, if there are two tables in the WHERE clause and there is no corresponding join condition in the query, the result will also be the CARTESIAN PRODUCT of the two

Binary Relational Operations:

The JOIN Operation
The **JOIN** operation, denoted by ⋈**,** is used to combine *related tuples* from two relations into single "longer" tuples.

This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations.

To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple.

We do this by using the JOIN operation and then projecting the result over the necessary attributes, as follows:
DEPT_MGR <- **DEPARTMENT** $\bowtie_{\mathbf{Mgr\_ssn=Ssn}}$ **EMPLOYEE**

RESULT <- $\pi_{\text{Dname, Lname, Fname}}$(DEPT_MGR)

Note that Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation. This referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.
The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation.

However, JOIN is very important because it is used very frequently when specifying database queries.

Consider the earlier example illustrating CARTESIAN PRODUCT, which included the following sequence of operations:
EMP_DEPENDENTS <-EMPNAMES X DEPENDENT
ACTUAL_DEPENDENTS<- $\sigma_{\text{ssn=Essn}}$(EMP_DEPENDENTS)

These two operations can be replaced with a single JOIN operation as follows:

ACTUAL_DEPENDENTS <-EMPNAMES $\bowtie_{\text{Ssn=Essn}}$ DEPENDENT

The general form of a JOIN operation on two relations:
$R(A1, A2, ..., An)$ and $S(B1, B2, ..., Bm)$ is

$R \bowtie_{\text{<join condition>}} S$

The result of the JOIN is a relation *Q* with *n* + *m* attributes *Q*(*A*1, *A*2, ..., *An*, *B*1, *B*2, *Bm*) in that order; *Q* has one tuple for each combination of tuples—one from *R* and one from *S*—*whenever the combination satisfies the join condition*.

The join condition is specified on attributes from the two relations *R* and *S* and is evaluated for each combination of tuples.

Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation *Q as a single combined tuple*.
A general join condition is of the form <condition> **AND** <condition> **AND**...**AND** <condition>
where each <condition> is of the form *Ai* θ *Bj*, *Ai* is an attribute of *R*, *Bj* is an attribute of *S*, *Ai* and *Bj* have the same domain, and θ (theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠ }.

A JOIN operation with such a general join condition is called a **THETA JOIN**.
Tuples whose join attributes are NULL or for which the join condition is FALSE *do not* appear in the result. In that sense, the JOIN operation does *not* necessarily preserve all of the information in the participating relations, because tuples that do not get combined with matching ones in the other relation do not appear in the result.

The EQUIJOIN and NATURAL JOIN
The most common use of JOIN involves join conditions with equality comparisons only.
Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN**. Both previous examples were EQUIJOINs. Notice that in the

result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple.

For example, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes *requires the values to be identical* in every tuple in the result.

Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN**—denoted by *—was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.

The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first.

Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project.

In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

PROJ_DEPT <- PROJECT * $\rho_{(Dname, Dnum, Mgr\_ssn, Mgr\_start\_date)}$(DEPARTMENT)

The same query can be done in two steps by creating an intermediate table DEPT as follows:

DEPT <- $\rho_{(Dname, Dnum, Mgr\_ssn, Mgr\_start\_date)}$(DEPARTMENT)
PROJ_DEPT <- PROJECT * DEPT

The attribute Dnum is called the **join attribute** for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations.

In the PROJ_DEPT relation, each tuple combines a PROJECT tuple with the DEPARTMENT tuple for the department that controls the project, but *only one join attribute value* is kept.

If the attributes on which the natural join is specified already *have the same names in both relations*, renaming is unnecessary.

For example, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write DEPT_LOCS <-DEPARTMENT * DEPT_LOCATIONS

The resulting relation combines each department with its locations and has one tuple for each location.
In general, the join condition for NATURAL JOIN is constructed by equating *each pair of join attributes* that have the same name in the two relations and combining these conditions with **AND**.

There can be a list of join attributes from each relation, and each corresponding pair must have the same name.
A more general, *but nonstandard* definition for NATURAL JOIN is
Q* <- *R* * $_{(<list1>),(<list2>)}$ *S
In this case, <list1> specifies a list of *i* attributes from *R*, and <list2> specifies a list of *i* attributes from *S*. The lists are used to form equality comparison conditions between pairs of corresponding attributes, and the conditions are then ANDed together. Only the list corresponding to attributes of the first relation *R*—<list1>— is kept in the result *Q*.
If no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples.

In general, if *R* has *nR* tuples and *S* has *nS* tuples, the result of a JOIN operation *R* <join condition> *S* will have between zero and *nR* * *nS* tuples.

The expected size of the join result divided by the maximum size *nR* * *nS* leads to a ratio called **join selectivity**, which is a property of each join condition.

If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS JOIN.

A single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins**, to distinguish them from a different join variation called *outer joins*

Informally, an *inner join* is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. Note that sometimes a join may be specified between a relation and itself.
The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an *n-way join*. For example, consider the following three-way join:

((PROJECT $_{\text{Dnum=Dnumber}}$ DEPARTMENT) $_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE)

This combines each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager. The net result is a consolidated relation in which each tuple contains this project-department-manager combined information.
In SQL, JOIN can be realized in several different ways. The first method is to specify the <join conditions> in the WHERE clause, along with any other selection conditions.
The second way is to use a nested relation
Another way is to use the concept of joined tables.

The construct of joined tables was added to SQL to allow the user to specify explicitly all the various types of joins, because the other methods were more limited. It also allows the user to clearly distinguish join conditions from the selection conditions in the WHERE clause.

A JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation. Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations.
Hence, the various JOIN operations are also *not strictly necessary* for the expressive power of the relational algebra. However, they are important to include as separate operations
because they are convenient to use and are very commonly applied in database applications.

OPERATION	**PURPOSE**	**NOTATION**
SELECT	Selects all tuples that satisfy the selection condition from a relation *R*	$\sigma_{<\text{selection condition}>}(R)$
PROJECT	Produces a new relation with only some of the attributes of *R*, and removes duplicate tuples.	$\pi_{<\text{attribute list}>}(R)$
THETA JOIN	Produces all combinations of tuples from *R*1 and *R*2 that satisfy the join condition.	$R1\theta_{<\text{join condition}>} R2$
EQUIJOIN	Produces all the combinations of tuples from *R*1 and *R*2 that satisfy a join condition with only equality comparisons.	$R1_{<\text{join condition}>} R2$, OR $R1_{(<\text{join attributes 1}>),(<\text{join attributes 2}>)} R2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of *R*2 are not included in the resulting relation;	$R1,_{<\text{join condition}>} R2$, OR $R1,_{(<\text{join attributes 1}>),(<\text{join}}$

	if the join attributes have the same names, they do not have to be specified at all.	attributes 2>) *R*2 OR *R*1 * *R*2
UNION	Produces a relation that includes all the tuples in *R*1 or *R*2 or both *R*1 and *R*2; *R*1 and *R*2 must be union compatible.	*R*1∪ *R*2
INTERSECTION	Produces a relation that includes all the tuples in both *R*1 and *R*2; *R*1 and *R*2 must be union compatible.	*R*1 ∩*R*2
DIFFERENCE	Produces a relation that includes all the tuples in *R*1 that are not in *R*2; *R*1 and *R*2 must be union compatible.	*R*1 – *R*2
CARTESIAN PRODUCT	Produces a relation that has the attributes of *R*1 and *R*2 and includes as tuples all possible combinations of tuples from *R*1 and *R*2.	*R*1x*R*2

Notation for Query Trees

In this section we describe a notation typically used in relational systems to represent queries internally. The notation is called a *query tree* or sometimes it is known as a *query evaluation tree* or *query execution tree*. It includes the relational algebra operations being executed and is used as a possible data structure for the internal representation of the query in an RDBMS.

A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf*

nodes of the tree, and represents the relational algebra operations as internal nodes.

An execution of the query tree consists of executing an internal node operation whenever its operands (represented by its child nodes) are available, and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Query: *For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.* This query corresponds to the following relational algebra expression:

$\pi_{<Pnumber, Dnum, Lname, Address, Bdate>}$

(

((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber}(DEPARTMENT))

\bowtie_{Mgr_ssn=Ssn}(EMPLOYEE))