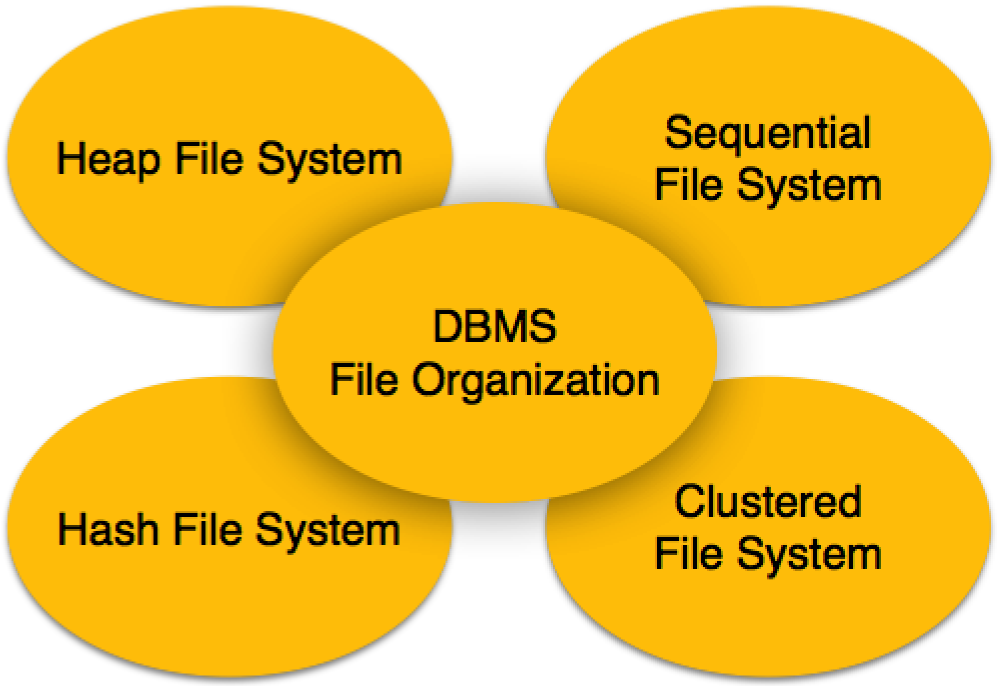
Relative data and information is stored collectively in file formats. A file is sequence of records stored in binary format. A disk drive is formatted into several blocks, which are capable for storing records. File records are mapped onto those disk blocks.

File Organization

The method of mapping file records to disk blocks defines file organization, i.e. how the file records are organized. The following are the types of file organization



[*Image: File Organization*]

- **Heap File Organization**: When a file is created using Heap File Organization mechanism, the Operating Systems allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of software to manage the records. Heap File does not support any ordering, sequencing or indexing on its own.
- **Sequential File Organization**: Every file record contains a data field (attribute) to uniquely identify that record. In sequential file organization mechanism, records are placed in the file in the some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.
- **Hash File Organization**: This mechanism uses a Hash function computation on some field of the records. As we know, that file is a collection of records, which has to be mapped on some block of the disk space allocated to it. This mapping is defined that the hash computation. The output of hash determines the location of disk block where the records may exist.
- **Clustered File Organization**: Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in a same disk block, that is, the ordering of records is not based on primary key or

search key. This organization helps to retrieve data easily based on particular join condition. Other than particular join condition, on which data is stored, all queries become more expensive.

File Operations

Operations on database files can be classified into two categories broadly.

- **Update Operations**
- **Retrieval Operations**

Update operations change the data values by insertion, deletion or update. Retrieval operations on the other hand do not alter the data but retrieve them after optional conditional filtering. In both types of operations, selection plays significant role. Other than creation and deletion of a file, there could be several operations, which can be done on files.

- **Open**: A file can be opened in one of two modes, read mode or write mode. In read mode, operating system does not allow anyone to alter data it is solely for reading purpose. Files opened in read mode can be shared among several entities. The other mode is write mode, in which, data modification is allowed. Files opened in write mode can be read also but cannot be shared.
- **Locate**: Every file has a file pointer, which tells the current position where the data is to be read or written. This pointer can be adjusted accordingly. Using find (seek) operation it can be moved forward or backward.
- **Read**: By default, when files are opened in read mode the file pointer points to the beginning of file. There are options where the user can tell the operating system to where the file pointer to be located at the time of file opening. The very next data to the file pointer is read.
- **Write**: User can select to open files in write mode, which enables them to edit the content of file. It can be deletion, insertion or modification. The file pointer can be located at the time of opening or can be dynamically changed if the operating system allowed doing so.
- **Close**: This also is most important operation from operating system point of view. When a request to close a file is generated, the operating system removes all the locks (if in shared mode) and saves the content of data (if altered) to the secondary storage media and release all the buffers and file handlers associated with the file.

The organization of data content inside the file plays a major role here. Seeking or locating the file pointer to the desired record inside file behaves differently if the file has records arranged sequentially or clustered, and so on.

It is used to determine an efficient file organization for each base relation. For example, if we want to retrieve student records in alphabetical order of name, sorting the file by student name is a good file organization. However, if we want to retrieve all students whose marks is in a certain range, a file ordered by student name would not be a good file organization. Some file organizations are efficient for bulk loading data into the database but inefficient for retrieve and other activities.

The objective of this selection is to choose an optimal file organization for each relation.

Types of File Organization

In order to make effective selection of file organizations and indexes, here we present the details different types of file Organization. These are:

- Heap File Organization
- Hash File Organization
- Indexed Sequential Access Methods (ISAM) File Organization
- B+- tree File Organization
- Cluster File Organization

Heap (unordered) File Organization

An unordered file, sometimes called a heap file, is the simplest type of file organization.

Records are placed in file in the same order as they are inserted. A new record is inserted in the last page of the file; if there is insufficient space in the last page, a new page is added to the file. This makes insertion very efficient. However, as a heap file has no particular ordering with respect to field values, a linear search must be performed to access a record. A linear search involves reading pages from the file until the required is found. This makes retrievals from heap files that have more than a few pages relatively slow, unless the retrieval involves a large proportion of the records in the file.

To delete a record, the required page first has to be retrieved, the record marked as deleted, and the page written back to disk. The space with deleted records is not reused. Consequently, performance progressively deteriorates as deletion occurs. This means that heap files have to be periodically reorganized by the Database Administrator (DBA) to reclaim the unused space of deleted records.

Heap files are one of the best organizations for bulk loading data into a table, as records are inserted at the end of the sequence; there is no overhead of calculating what page the record should go on.

Pros of Heap storage

Heap is a good storage structure in the following situations:

When data is being bulk-loaded into the relation.

The relation is only a few pages long. In this case, the time to locate any tuple is Short, even if the entire relation has been searched serially.

When every tuple in the relation has to be retrieved (in any order) every time the relation is accessed. For example, retrieve the name of all the students.

Cons of Heap storage

Heap files are inappropriate when only selected tuples of a relation are to be accessed.

Hash File Organization

In a hash file, records are not stored sequentially in a file instead a hash function is used to calculate the address of the page in which the record is to be stored.

The field on which hash function is calculated is called as Hash field and if that field acts as the key of the relation then it is called as Hash key. Records are randomly distributed in the file so it is also called as Random or Direct files. Commonly some arithmetic function is applied to the hash field so that records will be evenly distributed throughout the file.

Pros of Hash file organization

Hash is a good storage structure in the following situations:

When tuples are retrieve based on an exact match on the hash field value, particularly if the access order is random. For example, if the STUDENT relation is hashed on Name then retrieval of the tuple with Name equal to "Rahat Bhatia" is efficient.

Cons of Hash file organization

Hash is not a good storage structure in the following situations:

When tuples are retrieved based on a range of values for the hash field. For example, retrieve all students whose name begins with the "R".

ER to Relation model

ER Model when conceptualized into diagrams gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to Relational schema that is, it is possible to create relational schema using ER diagram. Though we cannot import all the ER constraints into Relational model but an approximate schema can be generated.

There are more than one processes and algorithms available to convert ER Diagrams into Relational Schema. Some of them are automated and some of them are manual process. We may focus here on the mapping diagram contents to relational basics.

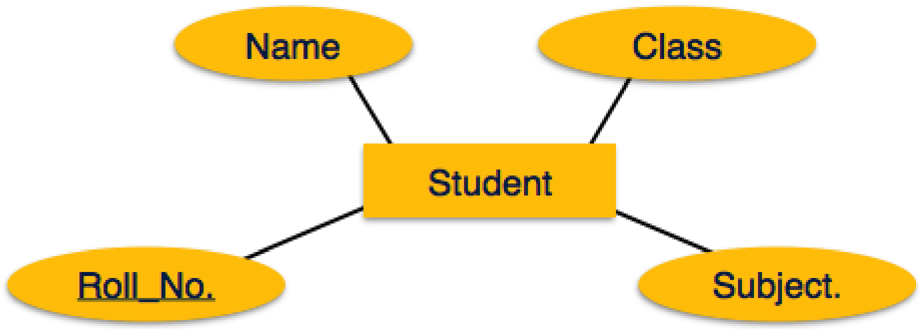
ER Diagrams mainly comprised of:

- Entity and its attributes
- Relationship, which is association among entities.

Mapping Entity

An entity is a real world object with some attributes.

Mapping Process (Algorithm):



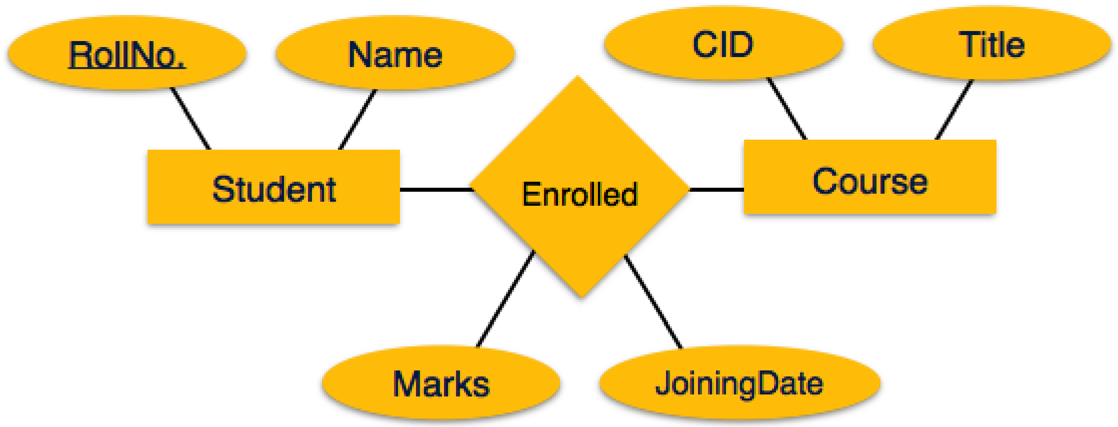
[*Image: Mapping Entity*]

- Create table for each entity
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key

Mapping relationship

A relationship is association among entities.

Mapping process (Algorithm):



[*Image: Mapping relationship*]

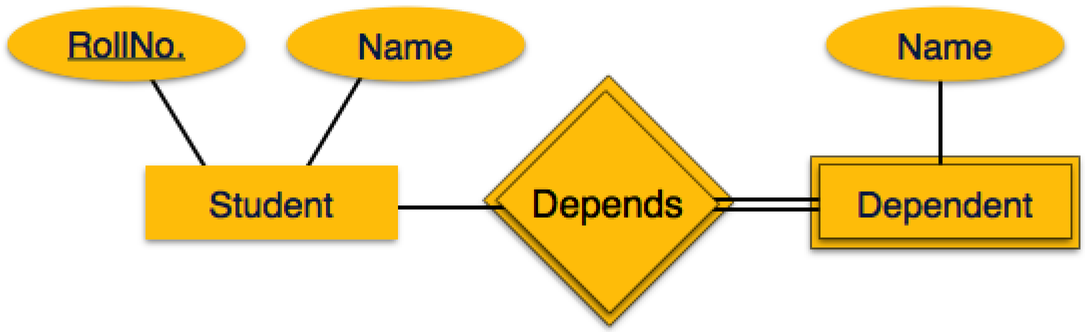
- Create table for a relationship

- Add the primary keys of all participating Entities as fields of table with their respective data types.
- If relationship has any attribute, add each attribute as field of table.
- Declare a primary key composing all the primary keys of participating entities.
- Declare all foreign key constraints.

Mapping Weak Entity Sets

A weak entity sets is one which does not have any primary key associated with it.

Mapping process (Algorithm):



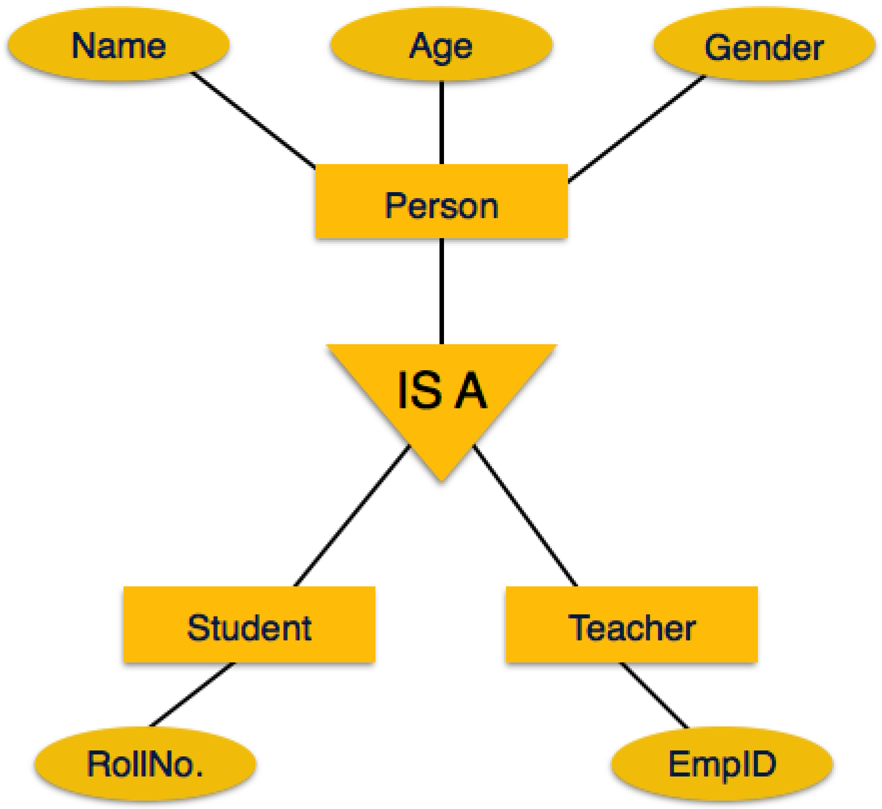
[*Image: Mapping Weak Entity Sets*]

- Create table for weak entity set
- Add all its attributes to table as field
- Add the primary key of identifying entity set
- Declare all foreign key constraints

Mapping hierarchical entities

ER specialization or generalization comes in the form of hierarchical entity sets.

Mapping process (Algorithm):

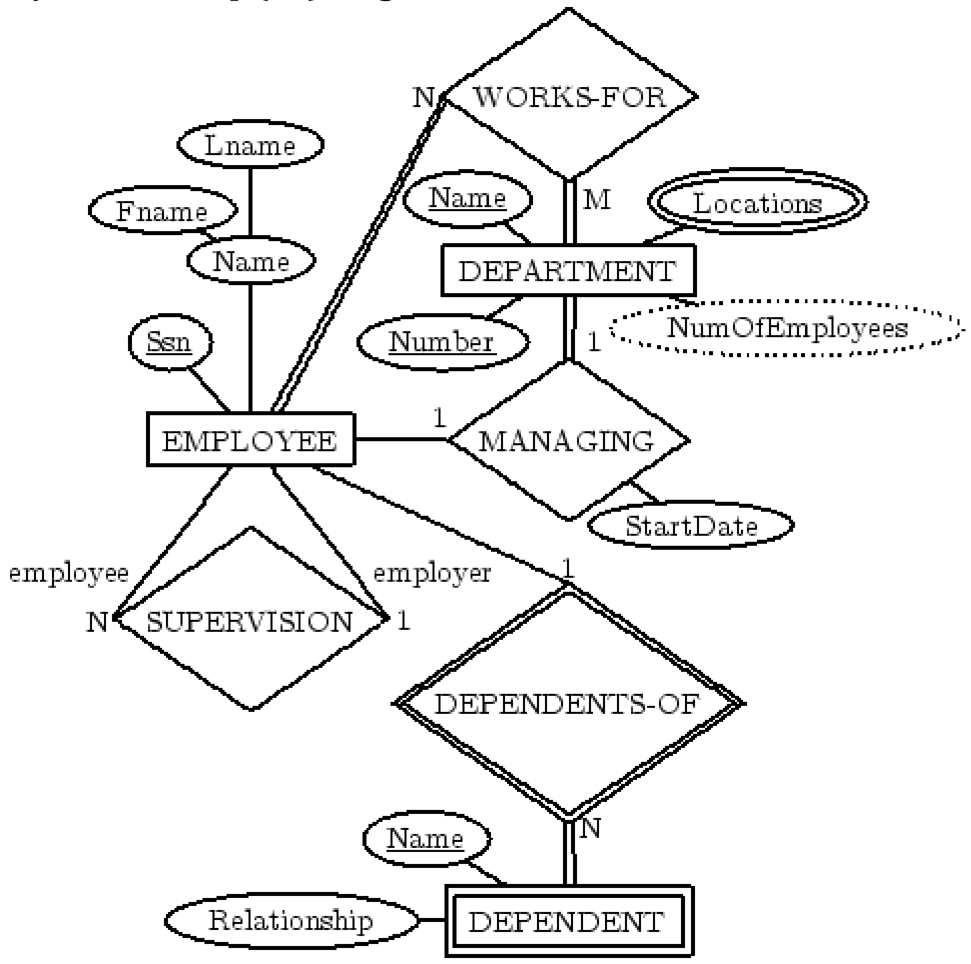


[*Image: Mapping hierarchical entities*]

- Create tables for all higher level entities
- Create tables for lower level entities
- Add primary keys of higher level entities in the table of lower level entities
- In lower level tables, add all other attributes of lower entities.
- Declare primary key of higher level table the primary key for lower level table
- Declare foreign key constraints.

ER-to-Relational Data Model

9.1 An Entity-Relationship (ER) Diagram



9.2 A Relational Data Schema

EMPLOYEE

Fname	Lname	SSN¯	SupervisorSSN

DEPARTMENT

Name	NUMBER¯	MANAGER-SSN	StartDate

DEPENDENT

Relationship	EMPL-SSN¯	Name¯

DEP-LOCATION

Location	DEP-NUMBER

WORKS-FOR

EmployeeSSN	DeptNumber

Comments:

EMPLOYEE (SupervisorSSN) → EMPLOYEE (SSN)

DEPARTMENT (MANAGER-SSN) → EMPLOYEE (SSN)

DEPENDENT (EMPL-SSN) → EMPLOYEE (SSN)

DEP-LOCATION (DEP-NUMBER) → DEPARTMENT (NUMBER)

WORKS-FOR (EmployeeSSN) → EMPLOYEE (SSN)

WORKS-FOR (DeptNumber) → DEPARTMENT (NUMBER)

unique: DEPARTMENT.NAME

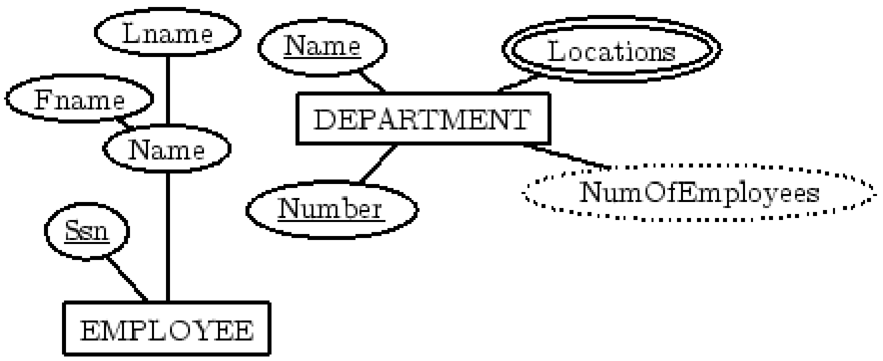
```
create table EMPLOYEE{
  Fname...
  Lname...
  SSN... primary key
  SupervisorSSN... reference EMPLOYEE(SSN)
}
create table DEPARTMENT{
  NAME...
  NUMBER... primary key
  MANAGER-SSN... references EMPLOYEE(SSN)
  StartDate
}
create table DEPENDENT{
  Relationship...
  EMPL-SSN... references EMPLOYEE(SSN)
  Name ...
  primary key(EMPL-SSN,Name)
}
```

```
create table DEP-LOCATION{
  Location... primary key
  DEPNUMBER... references DEPARTMENT(Number)
}
create table WORKS-FOR{
  EmployeeSSN... references EMPLOYEE(SSN)
 DeptNumber... references DEPARTMENT(Number)
}
```

9.3 The Mapping

(Strong) Entity Type into Relation

- Include the simple attributes
- Include the simple components of the composite attributes
- Identify the primary keys
- Don’t include: non-simple components of composite attributes, foreign keys, derived attributes, relational attributes



EMPLOYEE

Fname	Lname	SSN¯

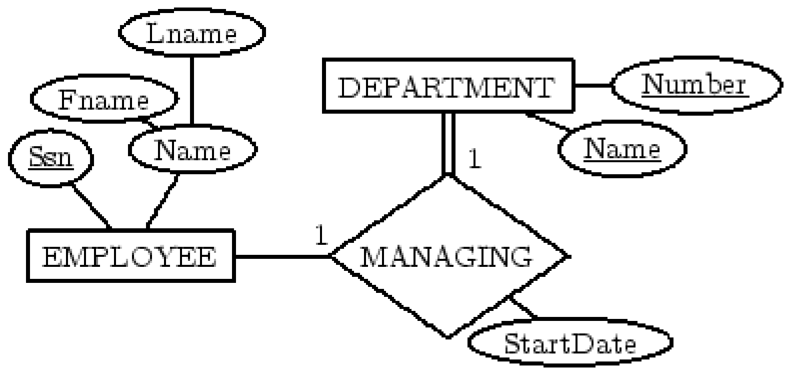
DEPARTMENT

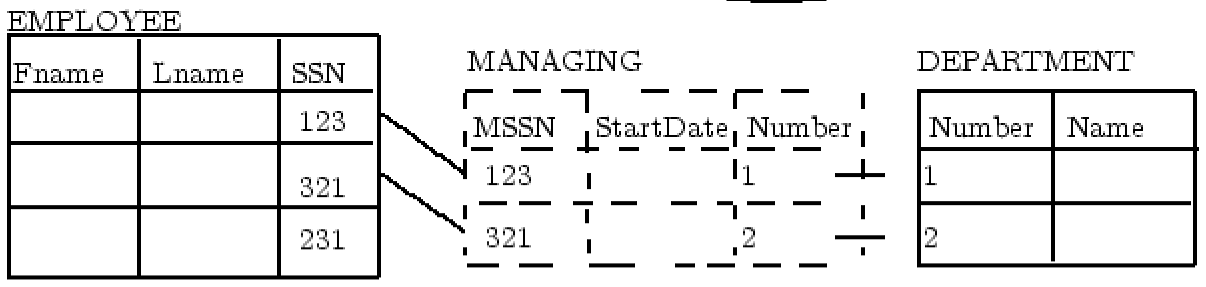
Name	Number¯

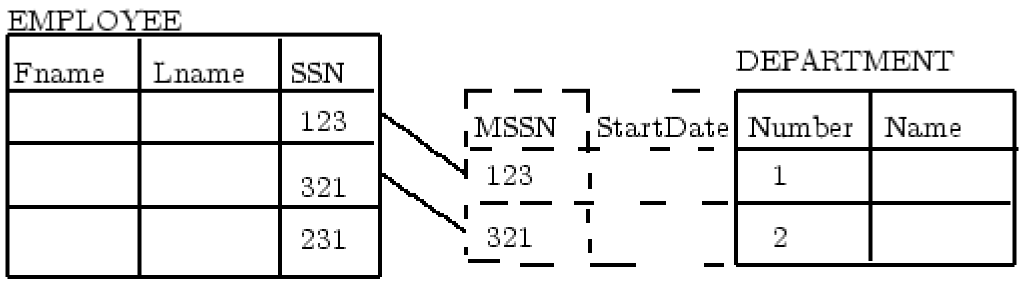
Binary 1:1 Relationship Types into Foreign Keys

- Include as foreign keys, in the relation of one entity type, the primary keys of the other entity type
- Include also the simple attributes of the relationship type

- If possible, the first entity type should have total participation in the relationship (to save memory!)





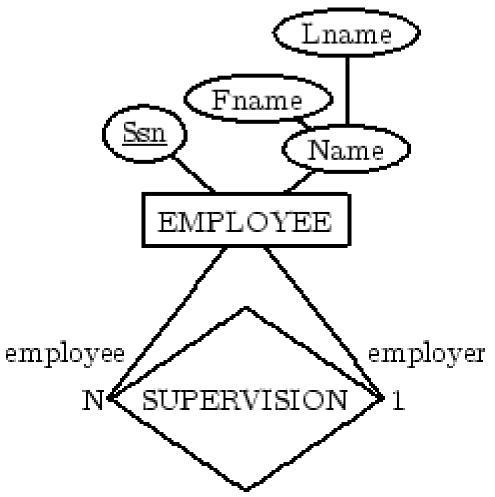


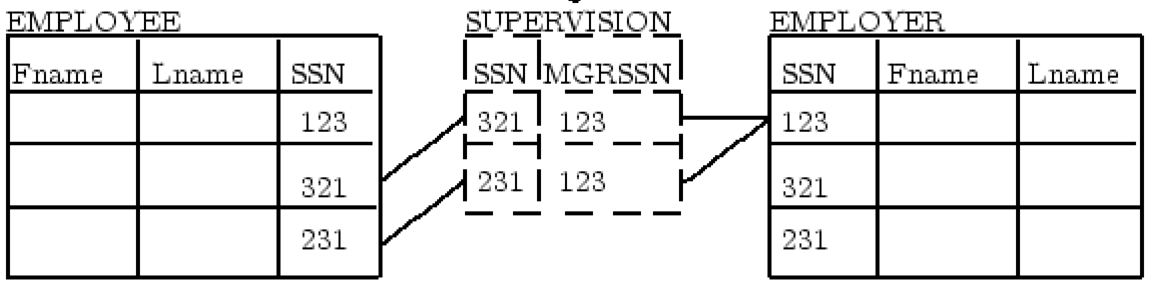
DEPARTMENT

MANAGER-SSN	StartDate

Binary 1:N Relationship Types into Foreign Keys

- Add as foreign keys, to the relation of the entity type at the N side, the primary keys of the entity type at the 1 side (don't duplicate records!)
- Include also the simple attributes of the relationship type





EMPLOYEE

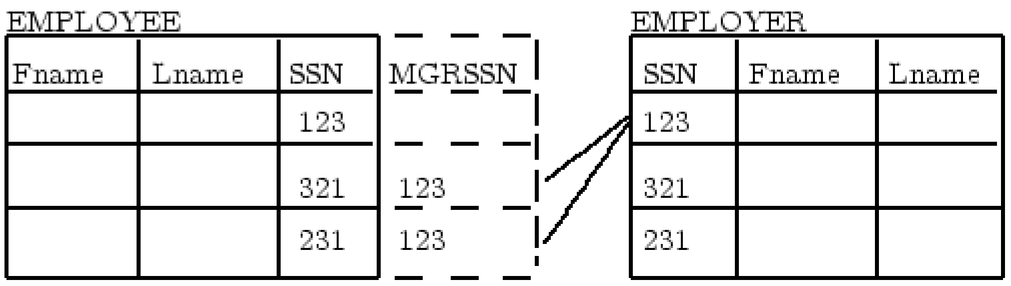
Fname	Lname	SSN
		123
		321
		231

SUPERVISION

SSN	MGRSSN
321	123
231	123

EMPLOYER

SSN	Fname	Lname
123		
321		
231		

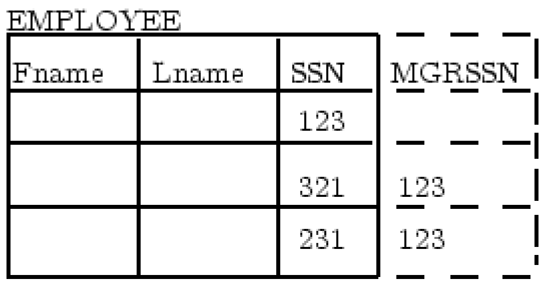


EMPLOYEE

Fname	Lname	SSN	MGRSSN
		123	
		321	123
		231	123

EMPLOYER

SSN	Fname	Lname
123		
321		
231		



EMPLOYEE

Fname	Lname	SSN	MGRSSN
		123	
		321	123
		231	123

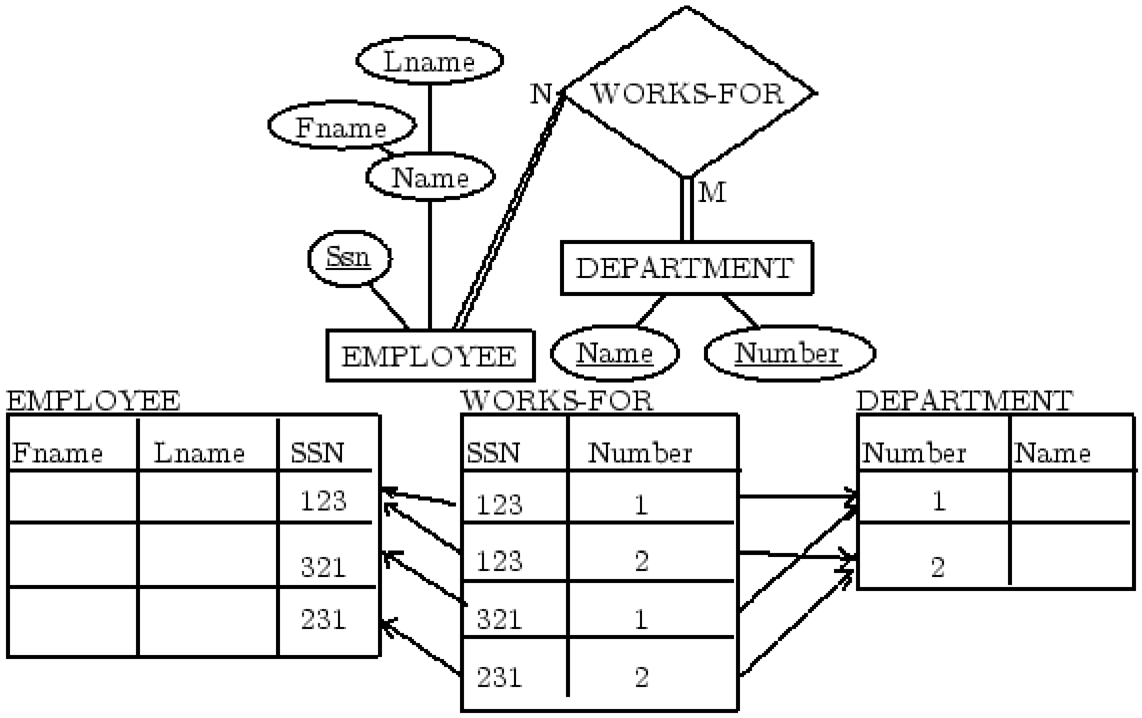
EMPLOYEE

SupervisorSSN

Binary M:N Relationship Type into Relation

- We don’t want to duplicate records!

- Set as foreign keys the primary keys of the participating entity types
- Include the simple attributes of the relationship type



EMPLOYEE

Fname	Lname	SSN
		123
		321
		231

WORKS-FOR

SSN	Number
123	1
123	2
321	1
231	2

DEPARTMENT

Number	Name
1	
2	

WORKS-FOR

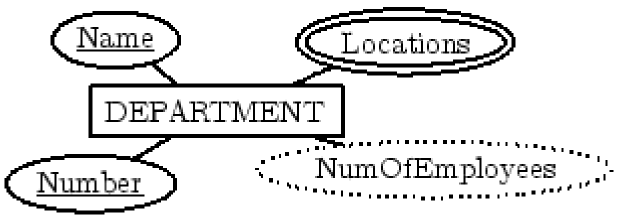
EmployeeSSN	DeptNumber

N-Ary Relationship Type

Similar to binary M:N relationship type

Multivalued Attribute into Relation

- Include the given attribute
- Include as foreign keys the primary attributes of the entity/relationship type owning the multivalued attribute
- Keys not designated within primary keys are to be mentioned as such in side comments



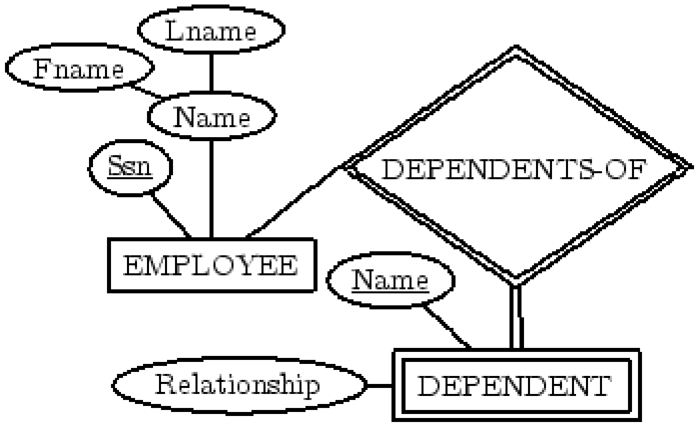
DEP-LOCATION

Location	DEP-NUMBER⁻

Resembles the treatment of a relationship type.

Weak Entity+Relationship Types into Relation

- Include simple attributes
- Add the owner's primary key attributes, as foreign key attributes
- Declare into a primary key the partial keys of the weak entity type combined with those imported from the owner



DEPENDENT

Relationship	EMPL-SSN⁻	NAME⁻

Set operations

Relations in relational algebra are seen as sets of tuples, so we can use basic set operations.

Review of concepts and operations from set theory

- set
- element
- no duplicate elements (but: multiset = bag)
- no order among the elements (but: ordered set)
- subset
- proper subset (with fewer elements)
- superset
- union
- intersection
- set difference
- cartesian product

Projection

Example: The table **E** (for **EMPLOYEE**)

nr	name	salary
1	John	100
5	Sarah	300
7	Tom	100

SQL	Result	Relational algebra
`select salary from E`	**salary** 100 300	**PROJECT**$_{salary}$(E)
`select nr, salary from E`	**nr** **salary** 1 100 5 300 7 100	**PROJECT**$_{nr, salary}$(E)

Note that there are no duplicate rows in the result.

Selection

The same table **E** (for **EMPLOYEE**) as above.

SQL	Result	Relational algebra

`select *`
`from E`
`where salary < 200`
	nr **name** **salary**
	1 John 100
	7 Tom 100
		$\textbf{SELECT}_{\text{salary < 200}}(E)$

`select *`
`from E`
`where salary < 200`
`and nr >= 7`
	nr **name** **salary**
	7 Tom 100
		$\textbf{SELECT}_{\text{salary < 200 and nr >= 7}}(E)$

Note that the select operation in relational algebra has nothing to do with the SQL keyword `select`. Selection in relational algebra returns those tuples in a relation that fulfil a condition, while the SQL keyword `select` means "here comes an SQL statement".

Relational algebra expressions

SQL	Result	Relational algebra

`select name, salary`
`from E`
`where salary < 200`
	name **salary**
	John 100
	Tom 100
		$\textbf{PROJECT}_{\text{name, salary}}(\textbf{SELECT}_{\text{salary < 200}}(E))$
		or, step by step, using an intermediate result
		
		Temp <- $\textbf{SELECT}_{\text{salary < 200}}(E)$
		Result <- $\textbf{PROJECT}_{\text{name, salary}}(\text{Temp})$

Notation

The operations have their own symbols. The symbols are hard to write in HTML that works with all browsers, so I'm writing **PROJECT** etc here. The real symbols:

Operation	My HTML	Symbol
Projection	**PROJECT**	π
Selection	**SELECT**	σ

Operation	My HTML	Symbol
Cartesian product	**X**	$\times$
Join	**JOIN**	$\bowtie$

Renaming	**RENAME**	ρ
Union	**UNION**	$\cup$
Intersection	**INTERSECTION**	$\cap$
Assignment	<-	$\leftarrow$

Left outer join	**LEFT OUTER JOIN**	⟕
Right outer join	**RIGHT OUTER JOIN**	⟖
Full outer join	**FULL OUTER JOIN**	⟗
Semijoin	**SEMIJOIN**	$\ltimes$

Example: The relational algebra expression which I would here write as

$\textbf{PROJECT}_{\textbf{Namn}}\ (\ \textbf{SELECT}_{\textbf{Medlemsnummer} < 3}\ (\ \textbf{Medlem}\ )\ )$

should actually be written

$$\pi_{\text{Namn}}\ (\sigma_{\text{Medlemsnummer} < 3}\ (\text{Medlem})\ )$$

Cartesian product

The *cartesian product* of two tables combines each row in one table with each row in the other table.

Example: The table **E** (for **EMPLOYEE**)

enr	ename	dept
1	Bill	A
2	Sarah	C
3	John	A

Example: The table **D** (for **DEPARTMENT**)

dnr	dname
A	Marketing

B	Sales
C	Legal

SQL	Result					Relational algebra
	enr	ename	dept	dnr	dname	
`select * from E, D`	1	Bill	A	A	Marketing	E **X** D
	1	Bill	A	B	Sales	
	1	Bill	A	C	Legal	
	2	Sarah	C	A	Marketing	
	2	Sarah	C	B	Sales	
	2	Sarah	C	C	Legal	
	3	John	A	A	Marketing	
	3	John	A	B	Sales	
	3	John	A	C	Legal	

- Seldom useful in practice.
- Usually an error.
- Can give a huge result.

Join (sometimes called "inner join")

The cartesian product example above combined each employee with each department. If we only keep those lines where the **dept** attribute for the employee is equal to the **dnr** (the department number) of the department, we get a nice list of the employees, and the department that each employee works for:

SQL	Result					Relational algebra
	enr	ename	dept	dnr	dname	
`select * from E, D where dept = dnr`	1	Bill	A	A	Marketing	**SELECT**$_{dept = dnr}$ (E **X** D) *or, using the equivalent join operation*
	2	Sarah	C	C	Legal	
	3	John	A	A	Marketing	E **JOIN**$_{dept = dnr}$ D

- A very common and useful operation.
- Equivalent to a cartesian product followed by a select.

- Inside a relational DBMS, it is usually much more efficient to calculate a join directly, instead of calculating a cartesian product and then throwing away most of the lines.
- Note that the same SQL query can be translated to several different relational algebra expressions, which all give the same result.
- If we assume that these relational algebra expressions are executed, inside a relational DBMS which uses relational algebra operations as its lower-level internal operations, different relational algebra expressions can take very different time (and memory) to execute.

Natural join

A normal inner join, but using the join condition that columns with the same names should be equal. Duplicate columns are removed.

Renaming tables and columns

Example: The table **E** (for **EMPLOYEE**)

nr	name	dept
1	Bill	A
2	Sarah	C
3	John	A

Example: The table **D** (for **DEPARTMENT**)

nr	name
A	Marketing
B	Sales
C	Legal

We want to join these tables, but:

- Several columns in the result will have the same name (**nr** and **name**).
- How do we express the join condition, when there are two columns called **nr**?

Solutions:

- Rename the attributes, using the *rename* operator.
- Keep the names, and prefix them with the table name, as is done in SQL. (This is somewhat unorthodox.)

SQL	Result	Relational algebra
`select * from E as E(enr, ename, dept), D as D(dnr, dname) where dept = dnr`	see table below	$(\textbf{RENAME}_{(enr, ename, dept)}(E))\ \textbf{JOIN}_{dept = dnr}\ (\textbf{RENAME}_{(dnr, dname)}(D))$
`select * from E, D where dept = D.nr`	see table below	$E\ \textbf{JOIN}_{dept = D.nr}\ D$

```
select *
from E as
E(enr, ename,
dept),
     D as
D(dnr, dname)
where dept =
dnr
```

enr	ename	dept	dnr	dname
1	Bill	A	A	Marketing
2	Sarah	C	C	Legal
3	John	A	A	Marketing

```
select *
from E, D
where dept =
D.nr
```

nr	name	dept	nr	name
1	Bill	A	A	Marketing
2	Sarah	C	C	Legal
3	John	A	A	Marketing

You can use another variant of the renaming operator to change the name of a table, for example to change the name of **E** to **R**. This is necessary when joining a table with itself (see below).

$\textbf{RENAME}_{R}(E)$

A third variant lets you rename both the table and the columns:
$\textbf{RENAME}_{R(enr, ename, dept)}(E)$

Aggregate functions

Example: The table **E** (for **EMPLOYEE**)

nr	name	salary	dept
1	John	100	A
5	Sarah	300	C
7	Tom	100	A
12	Anne	**null**	C

SQL	Result	Relational algebra
`select sum(salary) from E`	**sum**	$F_{sum(salary)}(E)$

	500	

Note:

- Duplicates are not eliminated.
- **Null** values are ignored.

SQL	Result	Relational algebra
`select count(salary) from E`	Result: **count** 3	$F_{\text{count(salary)}}(E)$
`select count(distinct salary) from E`	Result: **count** 2	$F_{\text{count(salary)}}(\textbf{PROJECT}_{\text{salary}}(E))$

You can calculate aggregates "grouped by" something:

SQL	Result		Relational algebra
`select sum(salary) from E group by dept`	**dept**	**sum**	${}_{\text{dept}}F_{\text{sum(salary)}}(E)$
	A	200	
	C	300	

Several aggregates simultaneously:

SQL	Result			Relational algebra
`select sum(salary), count(*) from E group by dept`	**dept**	**sum**	**count**	${}_{\text{dept}}F_{\text{sum(salary), count(*)}}(E)$
	A	200	2	
	C	300	1	

Standard aggregate functions: sum, count, avg, min, max

Hierarchies

Example: The table **E** (for **EMPLOYEE**)

nr	name	mgr
1	Gretchen	**null**
2	Bob	1
5	Anne	2
6	John	2
3	Hulda	1
4	Hjalmar	1
7	Usama	4

Going up in the hierarchy *one level*: What's the name of John's boss?

SQL	Result	Relational algebra

`select b.name from E p, E b where p.mgr = b.nr and p.name = "John"`	**name**
	Bob
		PROJECT$_{\text{bname}}$ ([**SELECT**$_{\text{pname = "John"}}$(**RENAME**$_{\text{P(pnr, pname, pmgr)}}$(E))] **JOIN**$_{\text{pmgr = bnr}}$ [**RENAME**$_{\text{B(bnr, bname, bmgr)}}$(E)])
		or, in a less wide-spread notation
		
		PROJECT$_{\text{b.name}}$ ([**SELECT**$_{\text{name = "John"}}$(**RENAME**$_{\text{P}}$(E))] **JOIN**$_{\text{p.mgr = b.nr}}$ [**RENAME**$_{\text{B}}$(E)])
		
		or, step by step
		
		P <- **RENAME**$_{\text{P(pnr, pname, pmgr)}}$(E)
		B <- **RENAME**$_{\text{B(bnr, bname, bmgr)}}$(E)
		J <- **SELECT**$_{\text{name = "John"}}$(P)
		C <- J **JOIN**$_{\text{pmgr = bnr}}$ B
		R <- **PROJECT**$_{\text{bname}}$(C)

Notes about renaming:

- We are joining **E** with itself, both in the SQL query and in the relational algebra expression: it's like joining two tables with the same name and the same attribute names.
- Therefore, some renaming is required.
- **RENAME**$_{\text{P}}$(E) **JOIN**$_{\dots}$ **RENAME**$_{\text{B}}$(E) is a start, but then we still have the same attribute names.

Going up in the hierarchy *two levels*: What's the name of John's boss' boss?

SQL	Result	Relational algebra
`select ob.name from E p, E b, E ob where b.mgr = ob.nr where p.mgr = b.nr and p.name = "John"`	**name** Gretchen	**PROJECT**$_{\text{ob.name}}$ (([**SELECT**$_{\text{name = "John"}}$(**RENAME**$_{\text{P}}$(E))] **JOIN**$_{\text{p.mgr = b.nr}}$ [**RENAME**$_{\text{B}}$(E)]) **JOIN**$_{\text{b.mgr = ob.nr}}$ [**RENAME**$_{\text{OB}}$(E)]) *or, step by step* P <- **RENAME**$_{\text{P(pnr, pname, pmgr)}}$(E) B <- **RENAME**$_{\text{B(bnr, bname, bmgr)}}$(E) OB <- **RENAME**$_{\text{OB(obnr, obname, obmgr)}}$(E) J <- **SELECT**$_{\text{name = "John"}}$(P) C1 <- J **JOIN**$_{\text{pmgr = bnr}}$ B C2 <- C1 **JOIN**$_{\text{bmgr = bbnr}}$ OB R <- **PROJECT**$_{\text{obname}}$(C2)

Recursive closure

Both one and two levels up: What's the name of John's boss, *and* of John's boss' boss?

SQL	Result	Relational algebra
`(select b.name ...) union (select ob.name ...)`	**name** Bob Gretchen	(...) **UNION** (...)

Recursively: What's the name of *all* John's bosses? (One, two, three, four or more levels.)

- Not possible in (conventional) relational algebra, but a special operation called **transitive closure** has been proposed.
- Not possible in (standard) SQL (SQL2), but in SQL3, and using SQL + a host language with loops or recursion.

Outer join

Example: The table **E** (for **EMPLOYEE**)

enr	ename	dept
1	Bill	A

2	Sarah	C
3	John	A

Example: The table **D** (for **DEPARTMENT**)

dnr	dname
A	Marketing
B	Sales
C	Legal

List each employee together with the department he or she works at:

SQL	Result	Relational algebra

`select *`
`from E, D`
`where edept = dnr`
or, using an explicit join
`select *`
`from (E join D on edept = dnr)`
	<table><tr><th>enr</th><th>ename</th><th>dept</th><th>dnr</th><th>dname</th></tr><tr><td>1</td><td>Bill</td><td>A</td><td>A</td><td>Marketing</td></tr><tr><td>2</td><td>Sarah</td><td>C</td><td>C</td><td>Legal</td></tr><tr><td>3</td><td>John</td><td>A</td><td>A</td><td>Marketing</td></tr></table>	E **JOIN**$_{\text{edept = dnr}}$ D

No employee works at department B, Sales, so it is not present in the result. This is probably not a problem in this case. But what if we want to know the number of employees at each department?

SQL	Result	Relational algebra

`select dnr, dname, count(*)`
`from E, D`
`where edept = dnr`
`group by dnr, dname`
or, using an explicit join
`select dnr, dname, count(*)`
`from (E join D on edept = dnr)`
`group by dnr, dname`
	<table><tr><th>dnr</th><th>dname</th><th>count</th></tr><tr><td>A</td><td>Marketing</td><td>2</td></tr><tr><td>C</td><td>Legal</td><td>1</td></tr></table>	$_{\text{dnr, dname}}$**F**$_{\text{count(*)}}$(E **JOIN**$_{\text{edept = dnr}}$ D)

No employee works at department B, Sales, so it is not present in the result. It disappeared already in the join, so the aggregate function never sees it. But what if we want it in the result, with the right number of employees (zero)?

Use a *right outer join*, which keeps all the rows from the right table. If a row can't be connected to any of the rows from the left table according to the join condition, **null** values are used:

SQL	Result					Relational algebra
`select * from (E right outer join D on edept = dnr)`	**enr**	**ename**	**dept**	**dnr**	**dname**	E **RIGHT OUTER JOIN**$_{\text{edept = dnr}}$ D
	1	Bill	A	A	Marketing	
	2	Sarah	C	C	Legal	
	3	John	A	A	Marketing	
	null	**null**	**null**	B	Sales	
`select dnr, dname, count(*) from (E right outer join D on edept = dnr) group by dnr, dname`	**dnr**	**dname**	**count**			$_{\text{dnr, dname}}\mathbf{F}_{\text{count(*)}}$(E **RIGHT OUTER JOIN**$_{\text{edept = dnr}}$ D)
	A	Marketing	2			
	B	Sales	1			
	C	Legal	1			
`select dnr, dname, count(enr) from (E right outer join D on edept = dnr) group by dnr, dname`	**dnr**	**dname**	**count**			$_{\text{dnr, dname}}\mathbf{F}_{\text{count(enr)}}$(E **RIGHT OUTER JOIN**$_{\text{edept = dnr}}$ D)
	A	Marketing	2			
	B	Sales	0			
	C	Legal	1			

Join types:

- **JOIN** = "normal" join = inner join
- **LEFT OUTER JOIN** = left outer join
- **RIGHT OUTER JOIN** = right outer join
- **FULL OUTER JOIN** = full outer join

Outer union

Outer union can be used to calculate the union of two relations that are *partially union compatible*. Not very common.

Example: The table **R**

A	**B**
1	2
3	4

Example: The table **S**

B	**C**
4	5
6	7

The result of an outer union between R and S:

A	**B**	**C**
1	2	**null**
3	4	**5**
null	6	**7**

Division

Who works on (at least) *all* the projects that Bob works on?

Semijoin

A join where the result only contains the columns from one of the joined tables. Useful in distributed databases, so we don't have to send as much data over the network.

Update

To update a named relation, just give the variable a new value. To add all the rows in relation **N** to the relation **R**:

R <- R **UNION** N

Relational algebra

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yields relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

Fundamental operations of Relational algebra:

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

These are defined briefly as follows:

Select Operation (σ)

Selects tuples that satisfy the given predicate from a relation.

Notation $\sigma_p(r)$
Where *p* stands for selection predicate and r stands for relation. *p* is prepositional logic formulae which may use connectors like and, or and not. These terms may use relational operators like: =, ≠, ≥, < , >, ≤.

For example:

$\sigma_{subject="database"}(Books)$

Output : Selects tuples from books where subject is 'database'.

$\sigma_{subject="database" \text{ and } price="450"}(Books)$

Output : Selects tuples from books where subject is 'database' and 'price' is 450.

$\sigma_{subject="database" \text{ and } price < "450" \text{ or } year > "2010"}(Books)$

Output : Selects tuples from books where subject is 'database' and 'price' is 450 or the publication year is greater than 2010, that is published after 2010.

Project Operation (∏)

Projects column(s) that satisfy given predicate.

Notation: $\prod_{A1, A2, An}$ (r)
Where a_1, a_2 , a_n are attribute names of relation r.

Duplicate rows are automatically eliminated, as relation is a set.

for example:

```
∏subject, author (Books)
```

Selects and projects columns named as subject and author from relation Books.

Union Operation (∪)

Union operation performs binary union between two given relations and is defined as:

```
r ∪ s = { t | t ∈ r or t ∈ s}
```

Notion: r U s

Where r and s are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold:

- r, s must have same number of attributes.
- Attribute domains must be compatible.

Duplicate tuples are automatically eliminated.

```
∏ author (Books) ∪ ∏ author (Articles)
```

Output : Projects the name of author who has either written a book or an article or both.

Set Difference (−)

The result of set difference operation is tuples which present in one relation but are not in the second relation.

Notation: r − s

Finds all tuples that are present in r but not s.

```
∏ author (Books) − ∏ author (Articles)
```

Output: Results the name of authors who has written books but not articles.

Cartesian Product (X)

Combines information of two different relations into one.

Notation: r X s

Where r and s are relations and there output will be defined as:

r X s = { q t | q ∈ r and t ∈ s}

```
∏ author = 'tutorialspoint'(Books X Articles)
```

Output : yields a relation as result which shows all books and articles written by tutorialspoint.

Rename operation (ρ)

Results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. rename operation is denoted with small greek letter rho *ρ*

Notation: ρ_x (E)

Where the result of expression E is saved with name of x.

Additional operations are:

- Set intersection
- Assignment
- Natural join

Relational Calculus

In contrast with Relational Algebra, Relational Calculus is non-procedural query language, that is, it tells what to do but never explains the way, how to do it.

Relational calculus exists in two forms:

Tuple relational calculus (TRC)

Filtering variable ranges over tuples

Notation: { T | Condition }

Returns all tuples T that satisfies condition.

For Example:

```
{ T.name |  Author(T) AND T.article = 'database' }
```

Output: returns tuples with 'name' from Author who has written article on 'database'.

TRC can be quantified also. We can use Existential (∃)and Universal Quantifiers (∀).

For example:

```
{ R| ∃T   ∈ Authors(T.article='database' AND R.name=T.name)}
```

Output : the query will yield the same result as the previous one.

Domain relational calculus (DRC)

In DRC the filtering variable uses domain of attributes instead of entire tuple values (as done in TRC, mentioned above).

Notation:

{ a_1, a_2, a_3, ..., a_n | P (a_1, a_2, a_3, ... ,a_n)}

where a1, a2 are attributes and P stands for formulae built by inner attributes.

For example:

```
{< article, page, subject > |  ∈ TutorialsPoint ∧ subject = 'database'}
```

Output: Yields Article, Page and Subject from relation TutorialsPoint where Subject is database.

Just like TRC, DRC also can be written using existential and universal quantifiers. DRC also involves relational operators.

Expression power of Tuple relation calculus and Domain relation calculus is equivalent to Relational Algebra.

Assuming you're joining on columns with no duplicates, which is by far the most common case:

- An inner join of A and B gives the result of A intersect B, i.e. the inner part of a venn diagram intersection.
- An outer join of A and B gives the results of A union B, i.e. the outer parts of a venn diagram union.

Examples

Suppose you have two Tables, with a single column each, and data as follows:

```
A    B
-    -
1    3
2    4
3    5
4    6
```

Note that (1,2) are unique to A, (3,4) are common, and (5,6) are unique to B.

Inner join

An inner join using either of the equivalent queries gives the intersection of the two tables, i.e. the two rows they have in common.

```
select * from a INNER JOIN b on a.a = b.b;
select a.*,b.*  from a,b where a.a = b.b;
```

```
a | b
--+--
3 | 3
4 | 4
```

Left outer join

A left outer join will give all rows in A, plus any common rows in B.

```
select * from a LEFT OUTER JOIN b on a.a = b.b;
select a.*,b.*  from a,b where a.a = b.b(+);
```

```
a |  b
--+-----
1 | null
2 | null
3 |    3
4 |    4
```

Full outer join

A full outer join will give you the union of A and B, i.e. All the rows in A and all the rows in B. If something in A doesn't have a corresponding datum in B, then the B portion is null, and vice versa.

```
select * from a FULL OUTER JOIN b on a.a = b.b;
```

```
 a   |  b
-----+-----
   1 | null
   2 | null
   3 |    3
   4 |    4
null |    6
null |    5
```

down vote

Joins can be categorized as:

Inner joins (the typical join operation, which uses some comparison operator like = or <>). These include equi-joins and natural joins.

Inner joins use a comparison operator to match rows from two tables based on the values in common columns from each table. For example, retrieving all rows where the student identification number is the same in both the students and courses tables.

Outer joins. Outer joins can be a left, a right, or full outer join.

Outer joins are specified with one of the following sets of keywords when they are specified in the FROM clause:

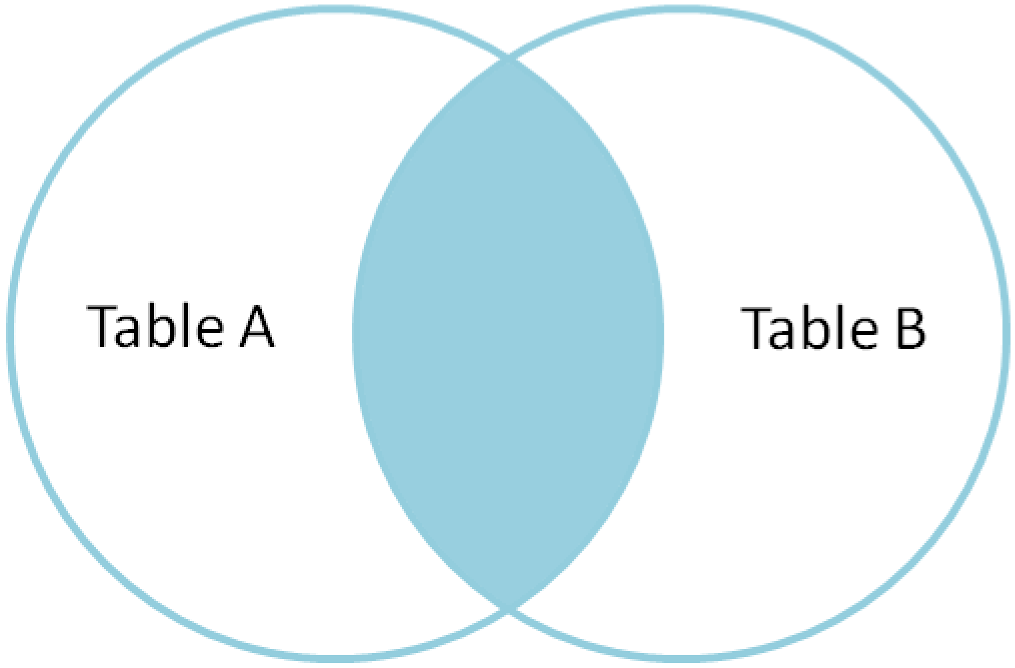
LEFT JOIN or LEFT OUTER JOIN -The result set of a left outer join includes all the rows from the left table specified in the LEFT OUTER clause, not just the ones in which the joined columns match. When a row in the left table has no matching rows in the right table, the associated result set row contains null values for all select list columns coming from the right table.

RIGHT JOIN or RIGHT OUTER JOIN - A right outer join is the reverse of a left outer join. All rows from the right table are returned. Null values are returned for the left table any time a right table row has no matching row in the left table.

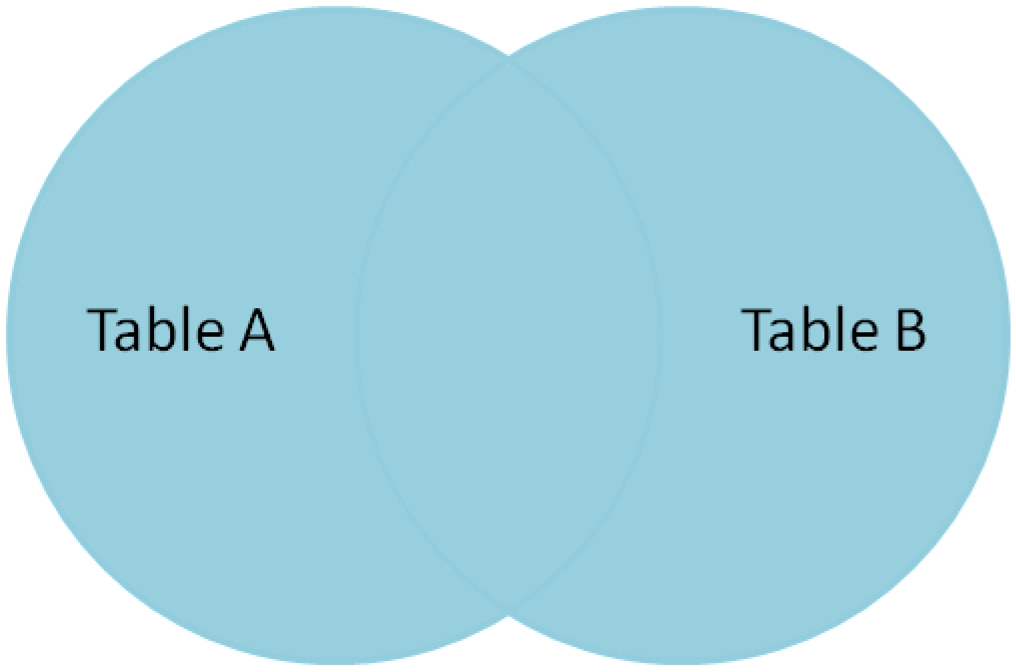
FULL JOIN or FULL OUTER JOIN - A full outer join returns all rows in both the left and right tables. Any time a row has no match in the other table, the select list columns from the other table contain null values. When there is a match between the tables, the entire result set row contains data values from the base tables.

Cross joins - Cross joins return all rows from the left table, each row from the left table is combined with all rows from the right table. Cross joins are also called Cartesian products. (A Cartesian join will get you a Cartesian product. A Cartesian join is when you join every row of one table to every row of another table. You can also get one by joining every row of a table to every row of itself.)

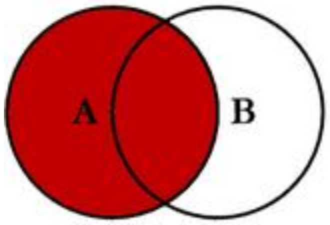
Inner Join



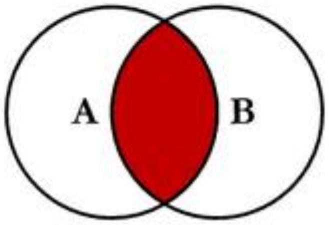
Full Outer Join



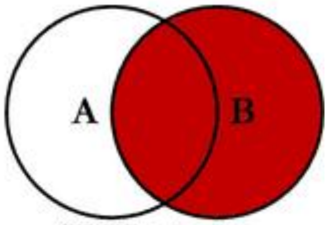
SQL JOINS



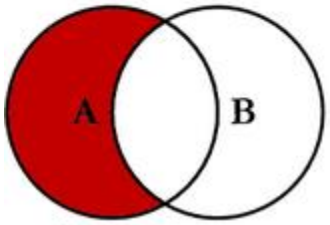
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



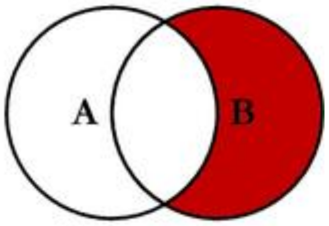
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



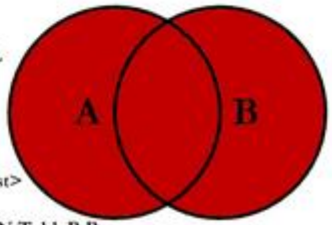
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



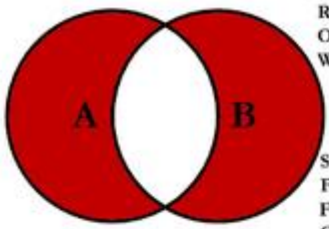
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```