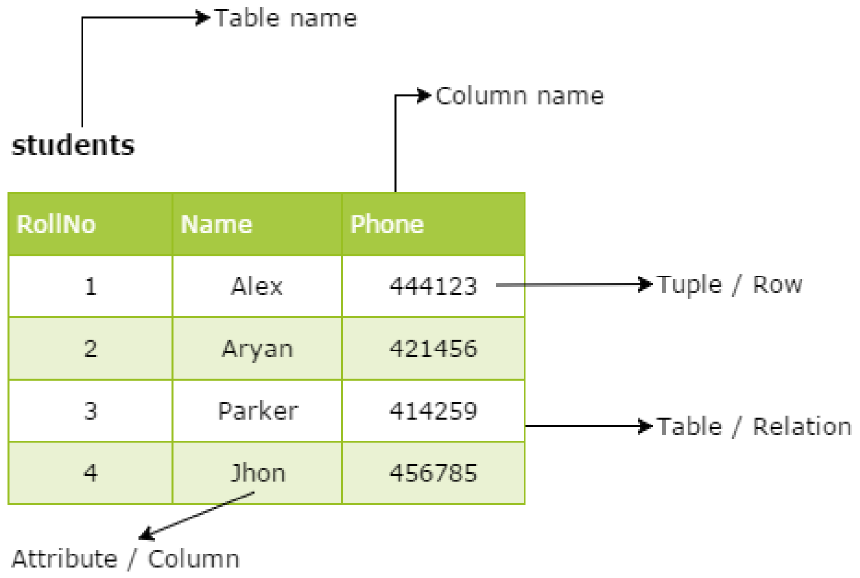
Relational Model



Relational Model Terms

Relational Model: Basic Structure

Formally, given sets $D_1, D_2, \ldots. D_n$ a **relation** r is a subset of

$$D_1 \times D_2 \times \ldots \times D_n$$

Thus, a relation is a set of n-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$

Example: *customer_name* = {Jones, Smith, Curry, Lindsay}

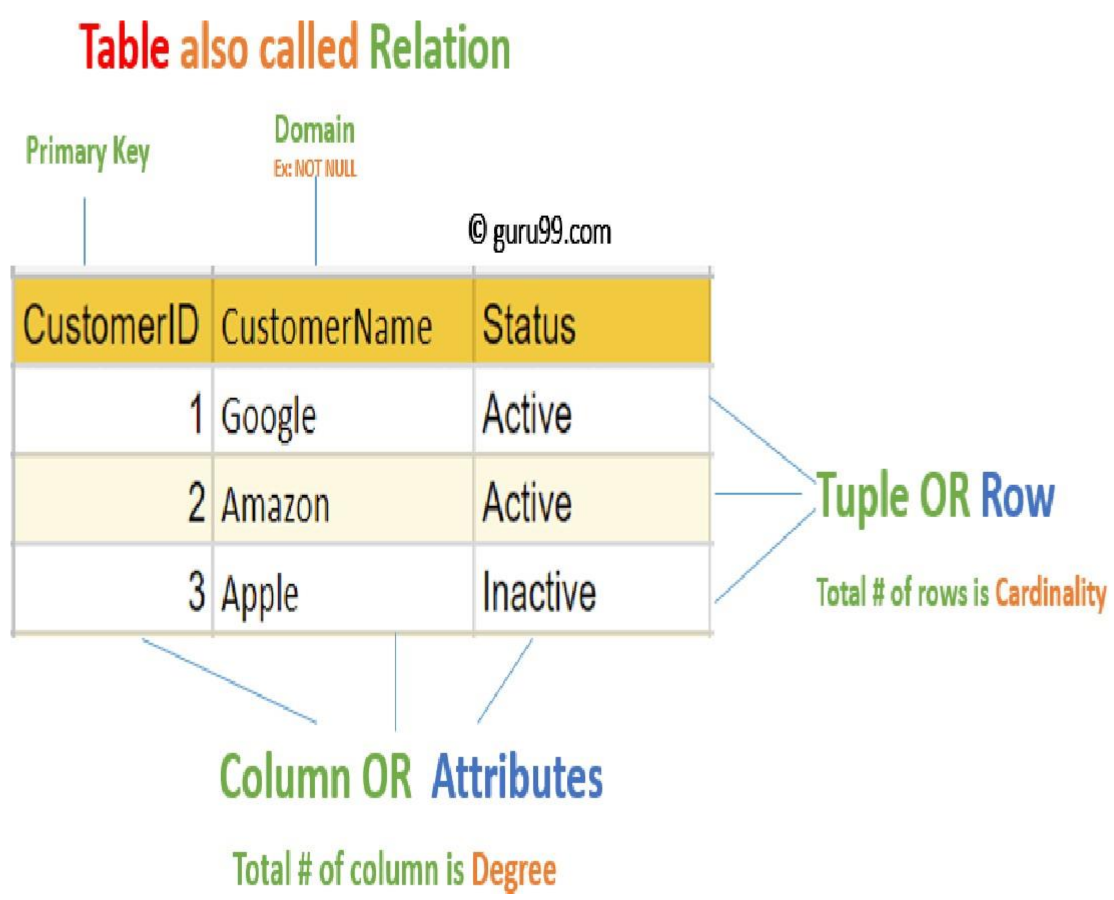
customer_street = {Main, North, Park}

customer_city = {Harrison, Rye, Pittsfield}

Then r = { (Jones, Main, Harrison),
(Smith, North, Rye),
(Curry, North, Rye),
(Lindsay, Park, Pittsfield) }

is a relation over

customer_name , *customer_street*, *customer_city*



Relational Model: Attribute types

- Each attribute of a relation has a name
- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
  - Note: multivalued attribute values are not atomic ({secretary. clerk}) is example of multivalued attribute *position*
  - Note: composite attribute values are not atomic
- The special value *null* is a member of every domain
- The null value causes complications in the definition of many operations

Relational Model: Relation Schema

- $A_1, A_2, \ldots, A_n$ are *attributes*
- $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*

Example:

Customer_schema = (*customer_name*, *customer_street*, *customer_city*)

$r(R)$ is a *relation* on the *relation schema* R

Example:

customer (*Customer_schema*)

Customer

Customer_name	Customer_street	Customer_city

Relational Model: Relation Instance

The current values (*relation instance*) of a relation are specified by a table

An element *t* of *r* is a *tuple*, represented by a *row* in a table

attributes
(or columns)

customer_name	*customer_street*	*customer_city*
Jones	**Main**	Harrison
Smith	North	Rye
Curry	North	Rye
Lindsay	Park	Pittsfield

tuples
(or rows)

Customer

Relational Model: Database

- A database consists of multiple relations
- Information about an enterprise is broken up into parts, with each relation storing one part of the information

  account : stores information about accounts
  depositor : stores information about which customer owns which account
  customer : stores information about customers

- Storing all information as a single relation such as
  bank (*account_number*, *balance*, *customer_name*, ..)
  results in repetition of information (e.g., two customers own an account) and the need for null values (e.g., represent a customer without an account)

customer-id	customer-name	customer-street	customer-city
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The customer table

account-number	balance
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The account table

customer-id	account-number
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The depositor table

Relational Model: Keys

Let K ⊆R

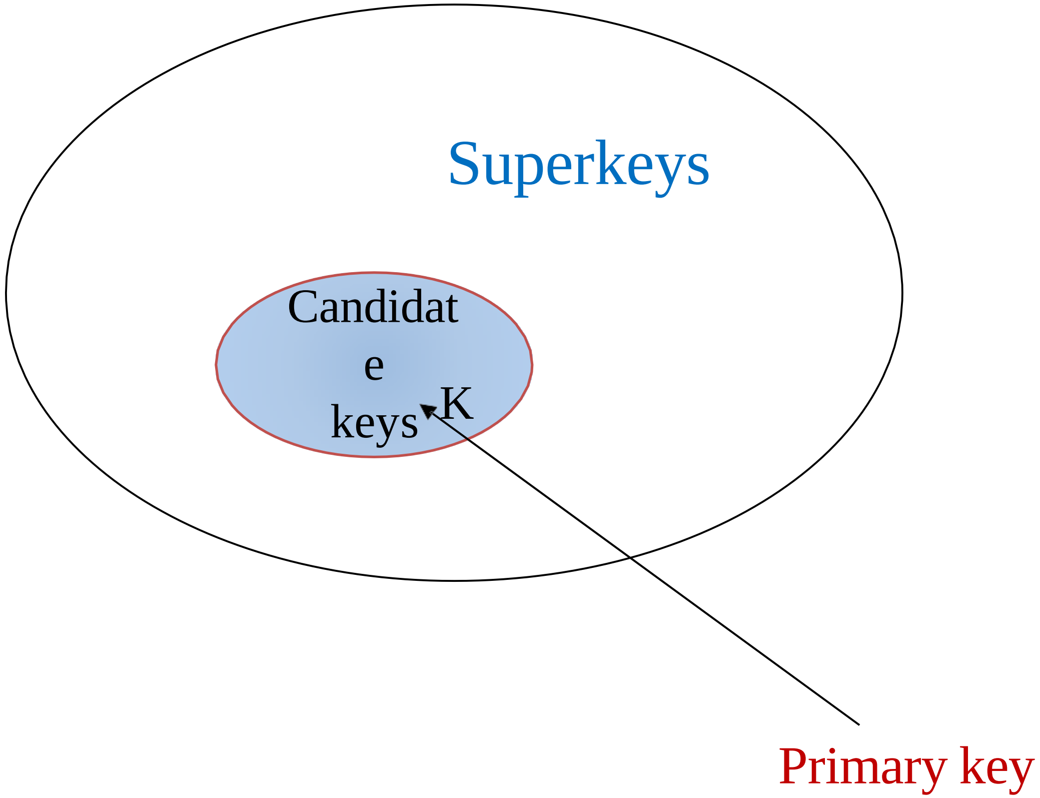
- *K* is a **superkey** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*
  - by “possible *r* ” we mean a relation *r* that could exist in the enterprise we are modeling.
  - Example: {*customer_name*, *customer_street*} and

    {*customer_name*} candidate key->primary key

    are both superkeys of *Customer*, if no two customers can possibly have the same name.

- *K* is a **candidate key** if *K* is minimal

Example: {*customer_name*} is a candidate key for.



- **Primary Key : Any candidate key**

Relational Integrity Constraints

Integrity Constraints

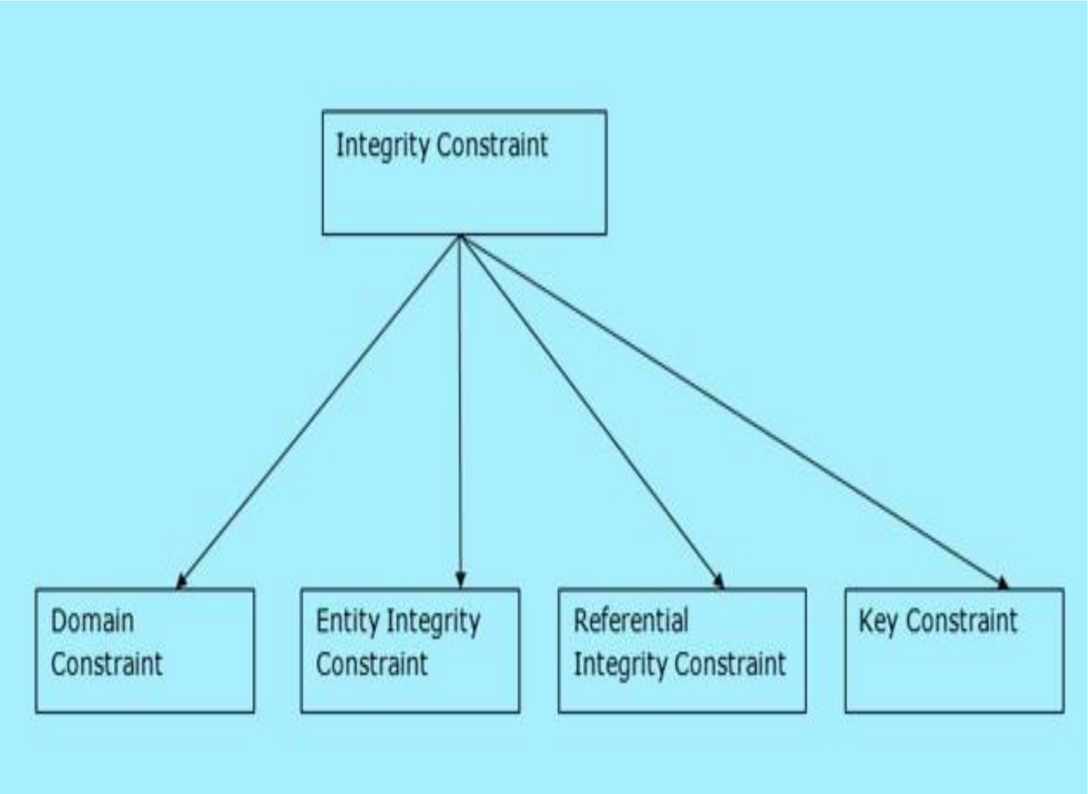
Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.

Examples :

- Domain Constraints
- Referential Integrity Constraints
- Entity Integrity Constraint
- Key Constraints

➢ Triggers

➢ Functional Dependencies



Integrity Constraints : Domain Constraints

- They define valid values for attributes
- They are the most elementary form of integrity constraint.
- They test values inserted in the database, and test queries to ensure that the comparisons makesense.
- use check constraint to ensure that an hourly-wage domain allows only values greater thana specified value.

- Example : **hourly_wages decimal(6,2) check(hourly_wages>=400.00)**
- The domain hourly-wages is declared to be a decimal number with 6 digits, 2 of which are after the decimal point
- The domain has a constraint that ensures that the hourly-wage is greater than 400.00.

Integrity Constraints : Referential Integrity

Ensures that a value that appears in one relation for a given set of attributes also appears fora certain set of attribute in another relation.

- If an account exists in the database with branch name “Perryridge”, then the branch “Perryridge” must actually exist in the database.

account (account-no, branch-name, balance)

A-123	**Perryridge**	**5000**

Foreign Key(branch-name)

branch (branch-name, branch-city, asset)

Perryridge	**Brooklyn**	**500,000**

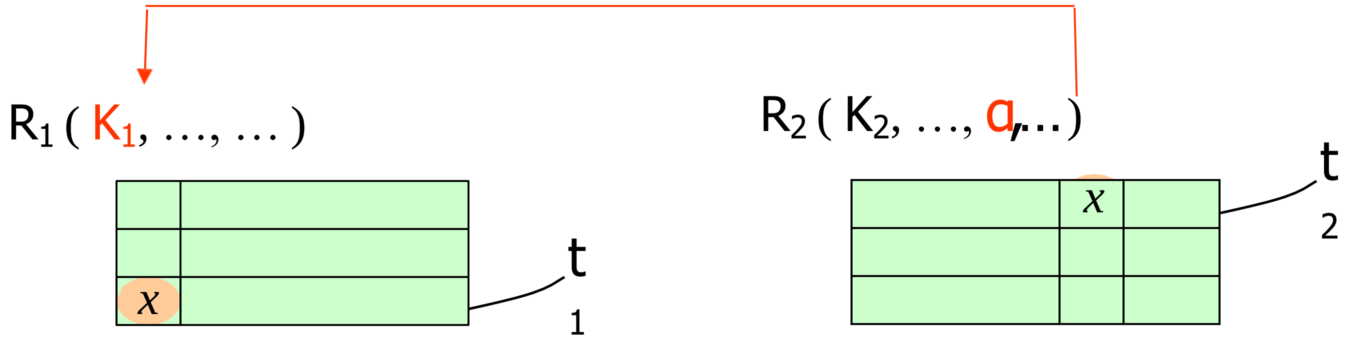
A set of attributes X in R is a foreign key if it is not a primary key of R but it is a primary key of some relation S.

Referential Integrity : Formal Definition

- Let $r_1(R_1)$ and $r_2(R_2)$ be relations with primary keys K_1 and K_2 respectively.

- The subset α of R_2 is a foreign key referencing K_1 in relation r_1, if for every t_2 in r_2 there must be a tuple t_1 in r_1 such that $t_1[K_1]=t_2[\alpha]$.

- Referential integrity constraint: $\prod_{\alpha}(r_2) \subseteq \prod_{K1}(r_1)$



Referential Integrity for Insertion and Deletion

The following tests must be made in order to preserve the following referential integrityconstraint:

- Insert. If a tuple t_2 is inserted into r_2. The system must ensure that there is a tuple t1 in r1 such that $t_1[K] = t_2[\alpha]$. That is

$$t_2[\alpha] \in \prod_K(r_1)$$

- Delete. If a tuple t_1 is deleted from r_1, the system must compute the set of tuples in r_2 that reference t_1:

$$\sigma_{\alpha=t1[K]}(r_2)$$

➢ if this set is not empty, either the delete command is rejected as an error, or the tuples that reference t_1 must themselves be deleted (cascading deletions arepossible)

Referential Integrity for Update

- if a tuple t_2 is updated in relation r_2 and the update modifies values for the foreign key α, then a test similar to the insert case is made. Let t_2' denote the new value of tuple t_2. The system must ensure that

$$t_2'[\alpha] \in \prod_K(r_1)$$

new foreign key value must exist

- if a tuple t_1 is updated in r_1, and the update modifies values for primary key(K), then a test similar to the delete case is made. The system must compute

$$\sigma_{\alpha = t1[K]}(r_2)$$

no foreign keys contain the old primary key

- using the old value of t_1 (the value before the update is applied). If this set is not empty, the update may be rejected as an error, or the update may be applied to the tuples in the set (cascade update), or the tuples in the set may be deleted.

- Parent Table (One's Side)

Dept-Name	Location
IT	AB building
Mechanical	C Building

Child Table: Many side(has Foreign key)

Ins_ID	Name	Contact	dname
11	Priya	9829211	Civil
21	ANjali	02393029	Mech
13	Aniket	389383	EnTC

Integrity Constraints : Entity Integrity Constraints

Entity Integrity

- **Requirement:**
  - All entries are unique and no part of a primary key may be null.
- **Purpose:**
  - Guarantees that each entity will have a unique identity and ensures that foreign key values can properly refer to primary key values.

Example :

PUBLISHER

Publisher_Code	Name	City
F-B1	Fajar Bakti	Malaysia
M-G1	McGraw Hill	UK
P-H1	Prentice Hall	UK
T-H1	Thompson	US

Figure 3-38 The Primary Key

Publisher_Code is the **PRIMARY KEY** in **PUBLISHER** table, so this field cannot have a null value and all entries are unique.

Integrity Constraints :Key Constraints

Key Constraints :

- Candidate Key which uniquely identifies tuples in a table.
- It should be unique and should not be having null values.
- It is applicable to entire entity as a whole so also part of Entity Integrity Constraint.

- Consider relation schema :
- *Cust(<u>cid</u>,cname,contact)*
- Here cid is the attribute which have to be provided a Primary Key Constraint to ensure that it is not having null values and its unique while creating the table.

Triggers

A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database.

To design a trigger mechanism, we must:

- Specify the conditions under which the trigger is to be executed.
- Specify the actions to be taken when the trigger executes.

Example :

Suppose that instead of allowing negative account balances, the bank deals with overdrafts by

- setting the account balance to zero
- creating a loan in the amount of the overdraft
- giving this loan a loan number which is identical to the account number of the overdrawn account.

The condition for executing the trigger is an update to the account relation that results in a negative balance value.

Normalization

Normalization is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly.

Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly.

Anomalies in DBMS

Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp_id , emp_name, emp_address ,emp_dept ,. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	Rick	Delhi	D001
101	Rick	Delhi	D002
123	Maggie	Agra	D890
166	Glenn	Chennai	D900
166	Glenn	Chennai	D004

The above table is not normalized. We will see the problems that we face when a table is not normalized.

Anomalies in DBMS

Update anomaly: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

Employee Table

Anomalies in DBMS

- **Insert anomaly**: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp_dept field doesn’t allow nulls.
- **Delete anomaly**: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.
- To overcome these anomalies we need to normalize the data.

Employee Table

Functional Dependencies

Functional Dependencies

We say an attribute, B, has a *functional dependency* on another attribute, A, if for any two records, which have the same value for A, then the values for B in these two records must be the same. We illustrate this as:

$A \rightarrow B$

Example: Suppose we keep track of employee email addresses, and we only track one email address for each employee. Suppose each employee is identified by their unique employee number. We say there is a functional dependency of email address on employee number:

employee number → email address

Functional Dependencies

EmpNum	EmpEmail	EmpFname	EmpLname
123	jdoe@abc.com	John	Doe
456	psmith@abc.com	Peter	Smith
555	alee1@abc.com	Alan	Lee
633	pdoe@abc.com	Peter	Doe
787	alee2@abc.com	Alan	Lee

If EmpNum is the PK then the FDs:

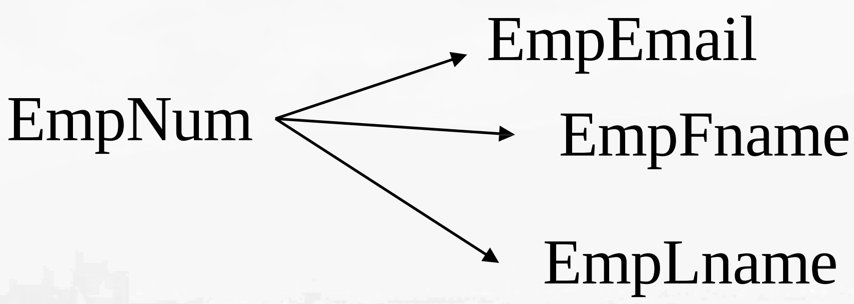
EmpNum → EmpEmail
EmpNum → EmpFname
EmpNum → EmpLname

must exist.

Functional Dependencies

EmpNum → EmpEmail
EmpNum → EmpFname
EmpNum → EmpLname

3 different ways you might see FDs depicted



EmpNum	EmpEmail	EmpFname	EmpLname

Determinant

Functional Dependency

EmpNum → EmpEmail

Attribute on the LHS is known as the ***determinant***

- EmpNum is a determinant of EmpEmail

Transitive dependency

Transitive dependency

Consider attributes A, B, and C, and where

$A \rightarrow B$ and $B \rightarrow C$.

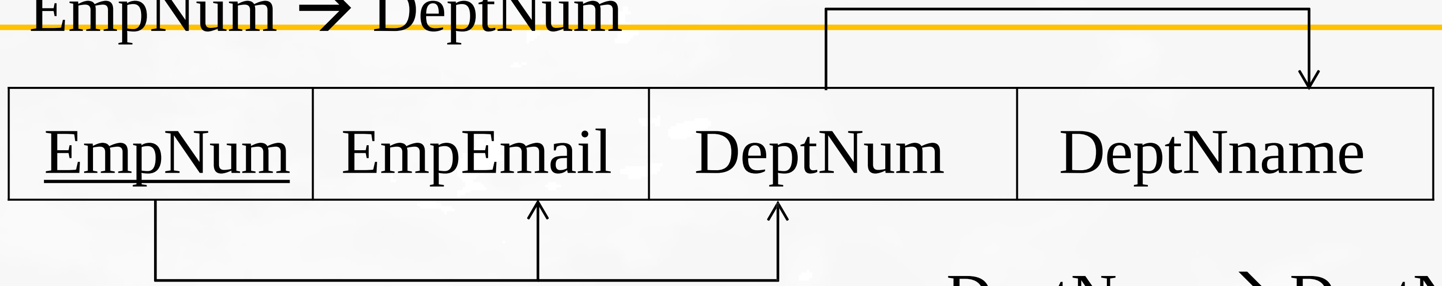
Functional dependencies are transitive, which means that we also have the functional dependency

$A \rightarrow C$

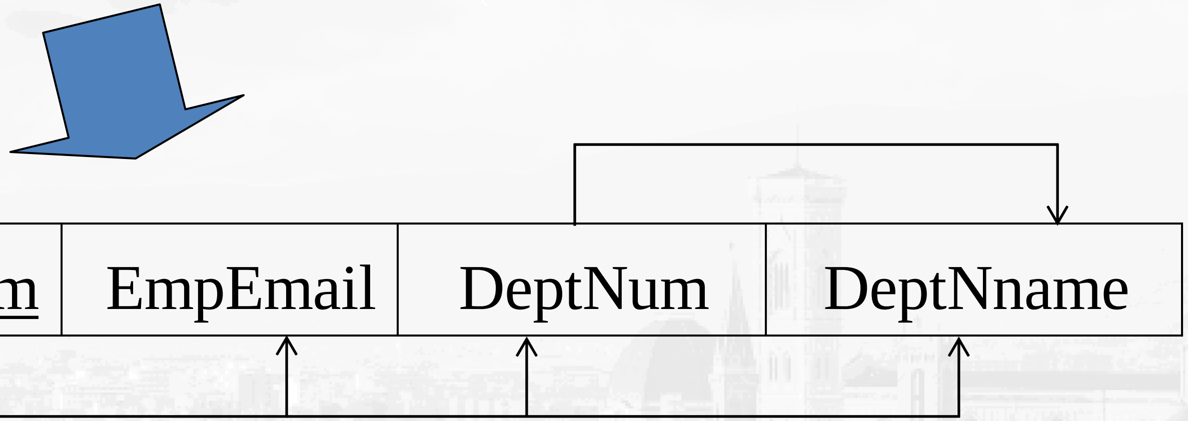
We say that C is transitively dependent on A through B.

Transitive dependency

EmpNum → DeptNum



DeptNum → DeptName



DeptName is *transitively dependent* on EmpNum via DeptNum

EmpNum → DeptName

Normalization

Here are the most commonly used normal forms:

- First normal form(1NF)
- Second normal form(2NF)
- Third normal form(3NF)
- Boyce & Codd normal form (BCNF)

First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212 9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 8123450987

- Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table.
- This table is **not in 1NF** as the rule says "each attribute of a table must have atomic (single) values", the emp_mobile values for employees Jon & Lester violates that rule.

First normal form (1NF)

To make the table complies with 1NF we should have the data like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

Second Normal Form (2NF)

A table is said to be in 2NF if both the following conditions hold:

•Table is in 1NF (First normal form)

•No non-prime attribute is dependent on the proper subset of any candidate key of table.

•An attribute that is not part of any candidate key is known as non-prime attribute

•**Example**: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says “**no** non-prime attribute is dependent on the proper subset of any candidate key of the table”.

Candidate Keys: {teacher_id, subject}
Non prime attribute: teacher_age

Second Normal Form (2NF)

To make the table complies with 2NF we can break it in two tables like this:

teacher_details table **teacher_subject table**

teacher_id	teacher_age
111	38
222	38
333	40

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

Table must be in 2NF

Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies. For e.g.

X -> Z is a transitive dependency if the following three functional dependencies hold true:

X->Y

Y does not ->X

Y->Z

Note: A transitive dependency can only occur in a relation of three of more attributes. This dependency helps us normalizing the database in 3NF (3rd Normal Form).

Third Normal form (3NF)

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Candidate Keys: {emp_id}
Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.

Third Normal form (3NF)

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

employee table

emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008
1201	Steve	222999

employee_zip table

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Boyce Codd normal form (BCNF)

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every functional dependency X->Y, X should be the super key of the table.

Example: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

emp_id	emp_nationality	emp_dept	dept_type	dept_no_of_emp
1001	Austrian	Production and planning	D001	200
1001	Austrian	stores	D001	250
1002	American	design and technical support	D134	100
1002	American	Purchasing department	D134	600

Functional dependencies in the table above:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}
Candidate key: {emp_id, emp_dept}
The table is not in BCNF as neither emp_id nor emp_dept alone are keys.

Boyce Codd normal form (BCNF)

To make the table comply with BCNF we can break the table in three tables like this:

emp_nationality table

emp_id	emp_nationality
1001	Austrian
1002	American

emp_dept table:

emp_dept	dept_type	dept_no_of_emp
Production and planning	D001	200
stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

emp_dept_mapping table

emp_id	emp_dept
1001	Production and planning
1001	stores
1002	design and technical support
1002	Purchasing department
emp_id	emp_dept

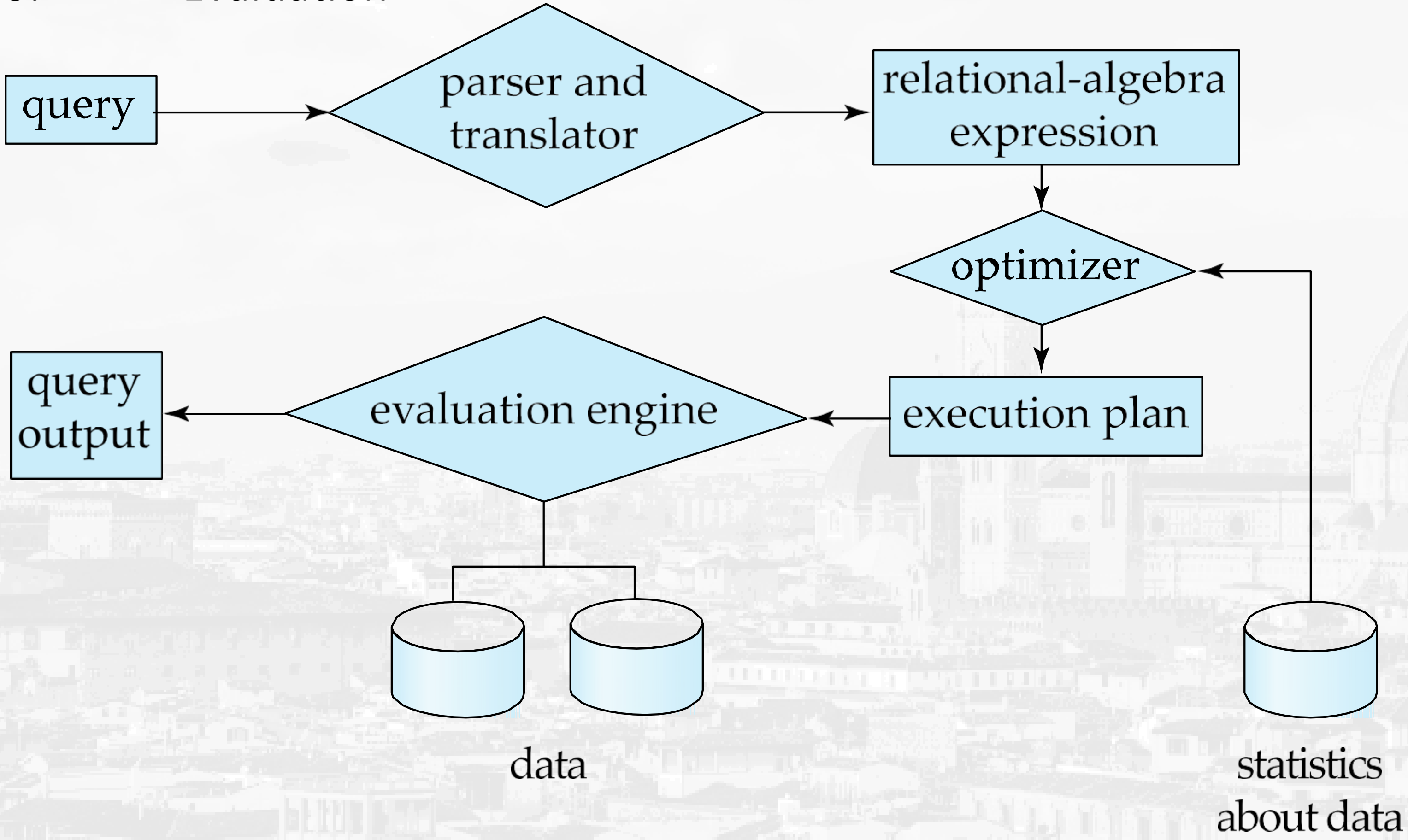
Functional dependencies:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}
Candidate keys:
For first table: emp_id
For second table: emp_dept

Query Processing

Overview
Measures of Query Cost
Selection Operation
Sorting
Join Operation
Evaluation of Expressions

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

Parsing and translation

translate the query into its internal form. This is then translated into relational algebra.

Parser checks syntax, verifies relations

Evaluation

The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Basic Steps in Query Processing : Optimization

A relational algebra expression may have many equivalent expressions

E.g., $\sigma_{salary<75000}(\prod_{salary}(instructor))$ is equivalent to

$\prod_{salary}(\sigma_{salary<75000}(instructor))$

Each relational algebra operation can be evaluated using one of several different algorithms

Correspondingly, a relational-algebra expression can be evaluated in many ways.

Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.

E.g., can use an index on *salary* to find instructors with salary < 75000,

or can perform complete relation scan and discard instructors with salary $\geq$ 75000

Measures of Query Cost

Cost is generally measured as total elapsed time for answering query

- Many factors contribute to time cost
  - *disk accesses, CPU*, or even network *communication*

Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account

- Number of seeks * average-seek-cost
- Number of blocks read * average-block-read-cost
- Number of blocks written * average-block-write-cost
  - Cost to write a block is greater than cost to read a block
    - data is read back after being written to ensure that the write was successful

Measures of Query Cost (Cont.)

For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures

t_T – time to transfer one block

t_S – time for one seek

Cost for b block transfers plus S seeks

$$b * t_T + S * t_S$$

We ignore CPU costs for simplicity, Real systems do take CPU cost into account

We do not include cost to writing output to disk in our cost formulae

Selection Operation

File scan

Algorithm **A1** (**linear search**). Scan each file block and test all records to see whether they satisfy the selection condition.

Cost estimate = b_r block transfers + 1 seek

b_r denotes number of blocks containing records from relation r

If selection is on a key attribute, can stop on finding record

cost = $(b_r /2)$ block transfers + 1 seek

Linear search can be applied regardless of

selection condition or

ordering of records in the file, or

availability of indices

Note: binary search generally does not make sense since data is not stored consecutively

except when there is an index available,

and binary search requires more seeks than index search

Selections Using Indices

Index scan – search algorithms that use an index

selection condition must be on search-key of index.

A2 (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition

$Cost = (h_i + 1) * (t_T + t_S)$

A3 (**primary index, equality on nonkey**) Retrieve multiple records.

Records will be on consecutive blocks

Let b = number of blocks containing matching records

$Cost = h_i * (t_T + t_S) + t_S + t_T * b$

Selections Using Indices

A4 (**secondary index, equality on nonkey**).

Retrieve a single record if the search-key is a candidate key

Cost = $(h_i + 1) * (t_T + t_S)$

Retrieve multiple records if search-key is not a candidate key

each of *n* matching records may be on a different block

Cost = $(h_i + n) * (t_T + t_S)$

Can be very expensive!

Selections Involving Comparisons

Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using a linear file scan,

or by using indices in the following ways:

A5 (**primary index, comparison**). (Relation is sorted on A)

For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there

For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple > *v;* do not use index

A6 (**secondary index, comparison**).

For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.

For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > *v*

In either case, retrieve records that are pointed to requires an I/O for each record

Linear file scan may be cheaper

Implementation of Complex Selections

Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$

A7 (**conjunctive selection using one index**).

Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}(r)$.

Test other conditions on tuple after fetching it into memory buffer.

A8 (**conjunctive selection using composite index**).

Use appropriate composite (multiple-key) index if available.

A9 (**conjunctive selection by intersection of identifiers**).

Requires indices with record pointers.

Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.

Then fetch records from file

If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.

A10 (**disjunctive selection by union of identifiers**).

Applicable if *all* conditions have available indices.

Otherwise use linear scan.

Use corresponding index for each condition, and take union of all the obtained sets of record pointers.

Then fetch records from file

Negation: $\sigma_{\neg\theta}(r)$

Use linear scan on file

If very few records satisfy $\neg\theta$, and an index is applicable to θ

Find satisfying records using index and fetch from file

Join Operation

Several different algorithms to implement joins:

- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge-join
- Hash-join

Materialization in Query Processing

- Materialization is an easy approach for evaluating multiple operations of the given query and storing the results in the temporary relations.
- The result can be the output of any join condition, selection condition, and many more.
- Thus, materialization is the process of creating and setting a view of the results of the evaluated operations for the user query.
- It is similar to the cache memory where the searched data get settled temporarily.
- e.g. Suppose there is a requirement to find the students who are studying in class ‘DESIGN_01’.

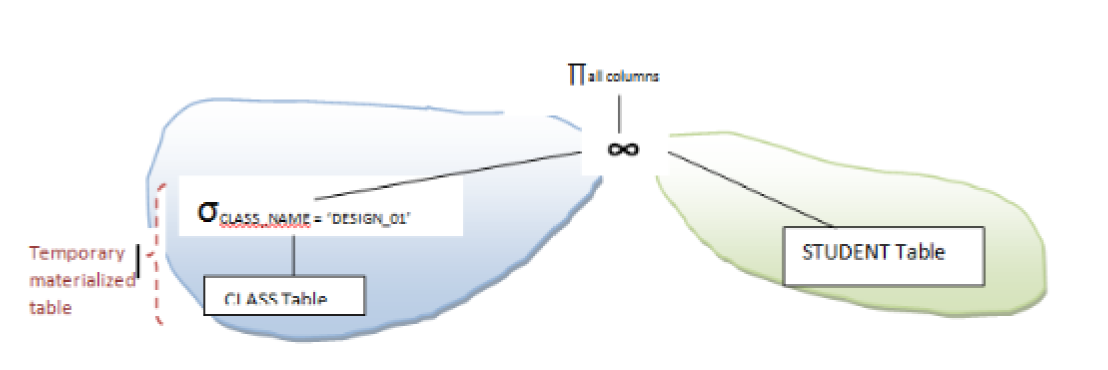
Code

```
SELECT * FROM STUDENT s, CLASS c
     WHERE s.CLASS_ID = c.CLASS_ID AND c.CLASS_NAME = 'DESIGN_01';
```

Here we can observe two queries: one is to select the CLASS_ID of ‘DESIGN_01’ and another is to select the student details of the CLASS_ID retrieved in the first query.

Materialization in Query Processing

- The DBMS also does the same. It breaks the query into two as mentioned above. Once it is broken, it evaluates the first query and stores it in the temporary table in the memory. This temporary table data will be then used to evaluate the second query.

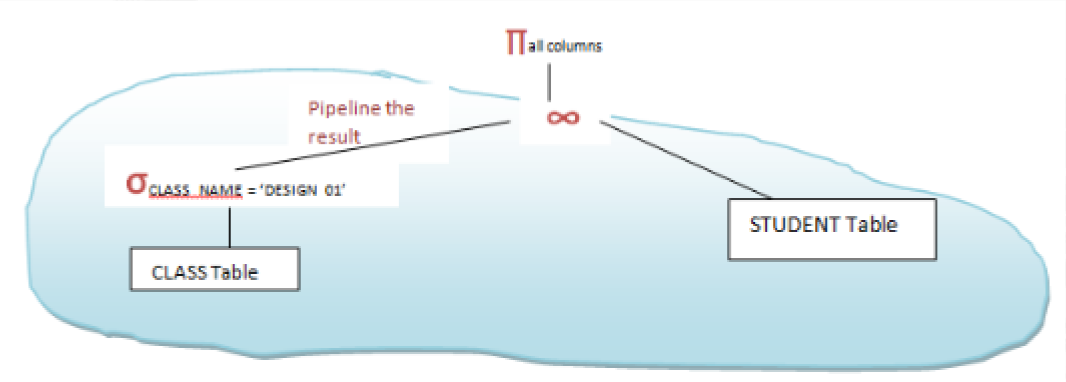


We can have any number of levels and so many numbers of temporary tables.

Cost of evaluation in this method is:
Cost = cost of individual SELECT + cost of write into temporary table

Pipelining in Query Processing

- In this method, DBMS do not store the records into temporary tables.
- Instead, it queries each query and result of which will be passed to next query to process and so on.
- It will process the query one after the other and each will use the result of previous query for its processing.
- In the example above, CLASS_ID of DESIGN_01 is passed to the STUDENT table to get the student details.



- In this method no extra cost of writing into temporary tables.
- It has only cost of evaluation of individual queries; hence it has better performance than materialization.

Pipelining (Cont.)

There are two types of pipelining:

Demand Driven or Lazy evaluation

- In this method, the result of lower level queries are not passed to the higher level automatically.
- It will be passed to higher level only when it is requested by the higher level.
- In this method, it retains the result value and state with it and it will be transferred to the next level only when it is requested.
- In our example above, CLASS_ID for DESIGN_01 will be retrieved, but it will be passed to STUDENT query only when it is requested. Once it gets the request, it is passed to student query and that query will be processed.

Producer Driven or Eager Pipelining

- In this method, the lower level queries eagerly pass the results to higher level queries.
- It does not wait for the higher level queries to request for the results.
- In this method, lower level query creates a buffer to store the results and the higher level queries pulls the results for its use.
- If the buffer is full, then the lower level query waits for the higher level query to empty it. Hence it is also called as PULL and PUSH pipelining.

Pipelining (Cont.)

Implementation of demand-driven pipelining

- Each operation is implemented as an **iterator** implementing the following operations
  - **open()**
    - E.g. file scan: initialize file scan
      - state: pointer to beginning of file
    - E.g.merge join: sort relations;
      - state: pointers to beginning of sorted relations
  - **next()**
    - E.g. for file scan: Output next tuple, and advance and store file pointer
    - E.g. for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
  - **close()**