Suggested Reading

Fundamentals of Database systems, 7th Ed
S. Navathe, R. Elmasri,

Introduction to Database Systems
C.J. Date, 8th Ed, 35th Anniversary of first published!!

Database System Concepts, 7th Ed.
Silberschatz, Korth. Sudarshan

Modern Database Management
Jeffrey A. Hoffer, Mary B. Prescott, Fred R. McFadden,
8th Ed

Overview

- Databases
    - Classifications
    - NoSQL, NewSQL
    - Cloud Databases, OLTP, OLAP
- Relational DBMS
    - Functional - Views
    - Logical Database Design
        - Key Types,
        - Referential Integrity,
        - Normalisation ???
    - Physical Database Design:
        - File Types : Heap, Sequential, Indexed Seq, Hash,
        - Index : Index Files, Clustering,
        - Partitioning

Transaction Management
- Concurrency Issues
- ACID Properties
    - Isolation Levels
- Concurrency Control:
    - Locking
    - Versioning : Multi-version concurrency control (MVCC)
    - Innodb Transaction Logs

Recovery

- Concepts
- Checkpointing
- Recovery Procedures : Undo, undo/redo
- Rollback, Rollforward, etc

Distributed Databases

- Strategies for distributing data
- Transparencies
- Two-Phase Commit Mechanism
- Distributed Query Optimisation
- Concurrency Control

Database Introduction
Ubiquitous

DBMS is at heart of today's operational and analytical business systems.

Database (DBMS) is where data is stored, managed, secured, analysed and served to applications / users.

Huge variety of database products to consider

Relational databases around since 50's, so they predate mobile devices, social media, internet applications and big data types - or extreme scale -.

Classification of Database Management Systems
- Single-user Databases - Ms Access
- Multi-user Databases – SQLServer, MySql, DB2
- Client-Server - MySQL
  - Fat Client / Thin Server, Thin Client / Fat Server
- Centralised Database
- Distributed Database
  - Homogeneous, Heteregeneous
  - Federated Databases - Independent DBMSs implementing some "cooperation functions":
- Newer Databases
  - NewSQL
  - NoSQL

http://db-engines.com/en/ranking

Big Data

- Both Structured and Unstructured data
- 90% of the data in the world today has been created in the last two years alone, petabyte, Exabyte, …..
- Big data is difficult to work with using relational database
- 3V's : Volumn, Velocity, Variety

Data coming from everywhere, Internet of things (IoT) e.g.

- Sensors used to gather climate information
- Posts to social media sites
- Digital pictures and videos
- Software logs, cameras
- Gps trails
- Purchase transaction records
- Cell phone gps signals
- Traffic, And many more…

RDBMS

- Undisputed leader, Maturity
- ACID, Recovery, Backup, Expertise, etc
- Guarantee performance of 1000+ transactions per sec
- SQL the de-facto standard for data processing
- SQL 1986 → SQL 2011

NoSQL

- Fastest-growing type of DBMS
- Scalability, millions of queries per second
- Flexible schema / data model
- Store data in a flexible and variety of formats
    1. Key-Value Pair
    2. Document Store
    3. Columnar
    4. Graph store

NewSQL

- Distributed architectures leading to much higher performance throughput
- Retain both SQL and ACID features using either
    o Completely new database
    o New storage engines, replacing Innnodb

NoSQL

1. **Key-Value Pair** - **Redis** - data is accessed using a key, e.g. ISBN, PPS#, GPS co-ordinate

   E.G. key is ISBN, value is rest of the information about the book.

   Key must be unique and can therefore be queried

   Huge amount of <u>unstructured</u> data - **blob**

   Move the responsibility for understanding how data is stored out of the database into the applications that retrieves the data

   Extremely fast for writing, and extremely fast for reading and updating...if you have the key

   Slow on multiple updates and if you have to query the entire store

2. **Document Store** : **MongoD**B, **CouchDB-** (eBay, NYT, Sourceforge, Shutterfly, The Guardian, SAP)

   Manages and stores data at **document** level.

   Similar to key-value pair, but <u>structure</u> is imposed on data - Document Structure, JSON, XML, pdf, word, etc

   Each document has a unique key, Queries / Searching quicker as data in structured format

3. **Columnar Database**– Cassandra (Facebook,Netflix)

Data stored as columns rather than as rows, so key points to column/s of data rather than row

ID,Firstname, Lastname, Occupation
1:Barry, Smith, Teacher
2:Jim, Finnegan, Doctor
3:Don, Jones, Lawyer

The Columnar Database stores columns together, like this:
1:Barry,2:Jim,3:Don
1:Smith,2:Finnegan,3:Jones
1:Teacher,2:Doctor,3:Lawyer

Suitable for mainly reads, infrequent writes

Faster querying and processing of data while storing data that's somewhat related

Highly compressed data as each column has same datatype

No fixed schema, meaning column names and keys can be added / deleted on the fly.

4. **Graph Store** –

Focuses on relationships between values

Stores data using a graph structure with nodes, edges and properties to represent and store data. Every element contains a direct pointer to its adjacent element

No concept of ACID

Distributed architectures leading to high-performance throughput

The nature of your data dictates the choice of database technologies.

Advantages of NoSQL Databases
- Data scaling, Big data : Massive amount of dynamic data stored

- Economics
  Clusters of <u>cheap</u> servers to manage the exploding data and transaction volumes

Disadvantages of NoSQL

- Lack of Maturity
- Support : Open Source
- Expertise

Classifications of databases

On-Line **Transaction** Processing (OLTP)

Used in everyday business, institutions, and organisations.

Stores **up-to-the-minute dynamic** data

Data is in a constant state of flux.

Used by retail stores, manufacturing companies, hospitals and clinics

On-Line **Analytical** Processing (OLAP)

Analytical is used to track historical data (**static data**), including trends, statistical data like weather reports, sports scores, etc.

Data is never modified,although new data might often be added.

Information from an analytical database reflects a point-in-time snapshot of the data

Cloud Databases

Implemented as either

- **Virtual Machines** : Upload own machine image with a DBMS installed and configured on it, or use ready-made machine images, e.g. Ms Azure, AWS,

- **DaaS** : Database service provider takes responsibility for installing and maintaining the database, and application owners pay according to usage, e.g. Xeround

In this course we will use WAMP 3.0
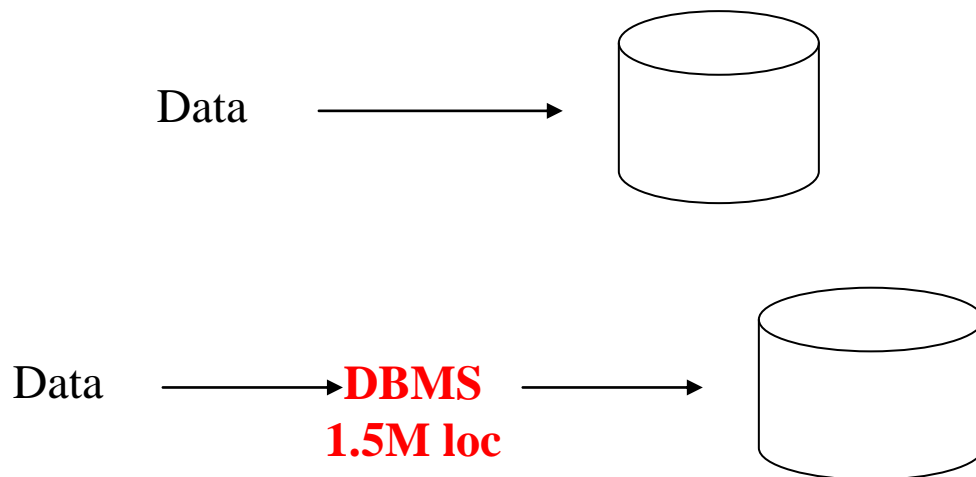
MariaDB free and very similar to MySQL, but
- has XtraDB as its storage engine
- Innodb storage engine in MySQL
- C:\wamp\bin\mysql\mysql5.6.17\data**ibdata1**

Michael Widenius – MySQL founder

File based Data Storage

Oldest and **fastest** way for data storage and retrieval (reads/writes) is using the file system

Data   ⟶

Data   ⟶ **DBMS 1.5M loc** ⟶

End user responsible for software that will read / write the file

Use file-based system if:

1. Database and application are simple, well-defined and not expected to change – static

2. Strict performance requirements as DBMS has significant overhead, e.g. European Space Agency

Database Definition:

"Large, integrated, shared pool of information in a form suitable for handling by a computer, basis upon which user within an organisation can draw inferences in conducting its business"

http://en.wikipedia.org/wiki/List_of_relational_database_management_systems

Advantages of DBMS Approach

- Data Sharing

- Centralised Data Management

- Improved security by restricting unauthorised access – passwords, encryption, privileges, etc

- Common policies for Security, Backup & Recovery, and Integrity Controls / Constraints

- Reduce <u>Application</u> Development Time

- Reduce Data Redundancy

- Physical Data Independence (aka Data Insulation)

    Applications is independent to changes in storage structure and access techniques - Mapping

    If physical layout changes, application need not change, only the mapping tables

Disadvantages of DBMS Approach

- Database systems are complex – MySQL documentation has 26 chapters - difficult, and time-consuming to understand, design and deploy

- Slow, lots of code in DBMS – MySQL 1.5m lines of code

- Substantial hardware (SSD vs HDD, RAID, etc) and software start-up costs – Oracle very expensive, MySql free!!

- Specialised personnel required

- Extensive conversion costs in moving from a file-based system to a database system

SSD vs HDD

SSD (Solid State Drive) access data completely electronically, faster speeds, more expensive
Same concept of blocks, sectors, clusters

HDD access data electro**mechanically,** slower but cheaper

Tyically 17ms (HDD) compared with 0.1ms(SSD)

https://en.wikipedia.org/wiki/Solid-state_drive

SSD Performance characteristics

- Super fast <u>sequential</u> reads and writes

- Fast <u>random</u> reads, slow <u>random</u> writes

- No requirement to defrag disk

- More reliable as they have no moving parts - quieter

- Less susceptible to shock - dropping

- Use less power, much less heat, and don't vibrate

- Price per bit - more expensive than HD
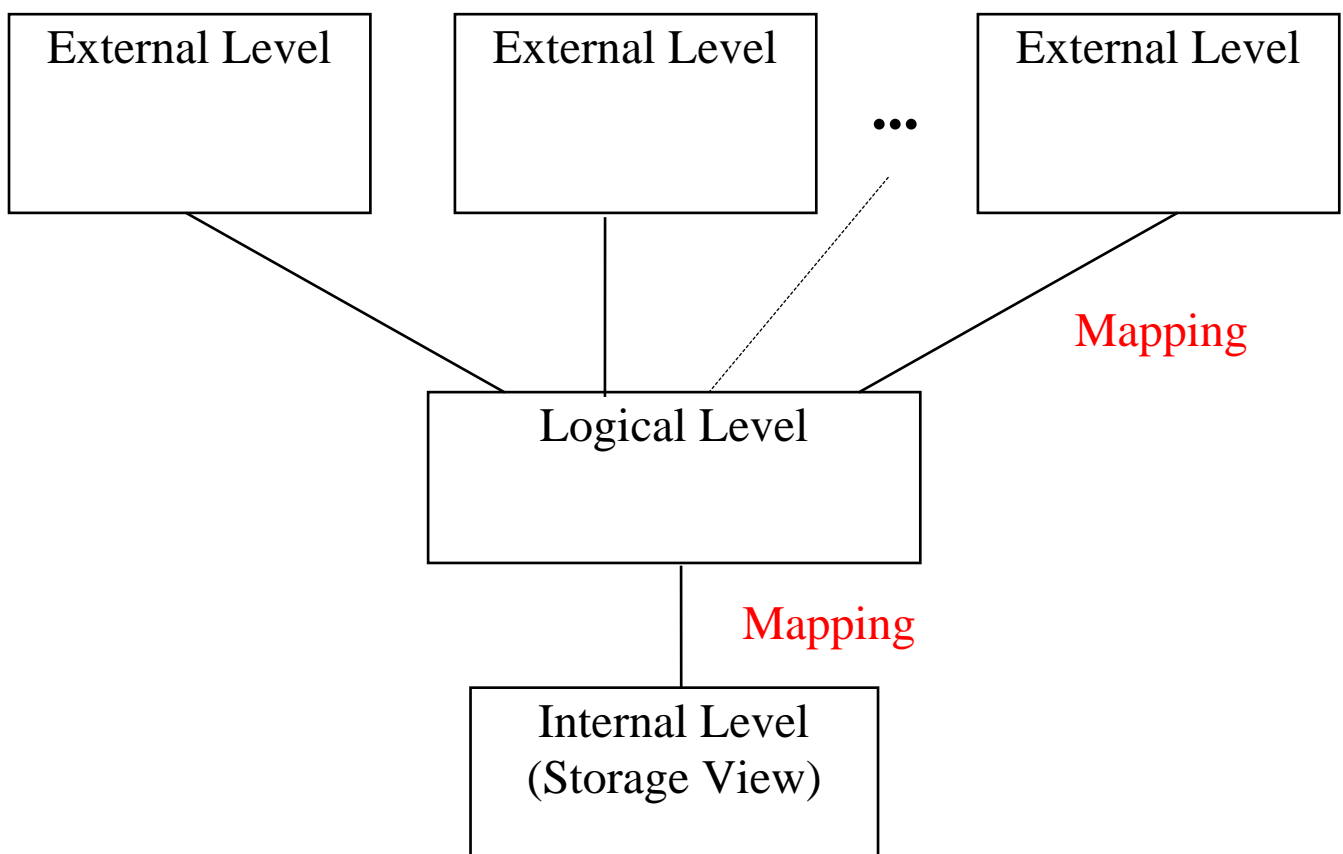
- SDD $0.20 / GB, HDD $0.04

But…
- SSDs have issues with mixed/random reads and writes

- Performance may degrade over time

- When stored unpowered offline, HDD retains data significantly longer than SSDs.

- Issue with multiple power outages

DBMS Architecture

Three layers within a Database:

      1. Functional (External, Subschema) Layer
      2. Logical (Conceptual, **Schema**) Layer
      3. Physical (Internal) Layer

External Level	External Level	...	External Level

Mapping

Logical Level

Mapping

Internal Level (Storage View)

Two levels of mappings**:**

    External / Conceptual Mapping
    Conceptual / Internal Mapping

- Physical Level: How data is stored physically on disk

  Storage Engine, File organisation / type, INDEX's, clustering, hashing, Sequential, etc

  Only **one** physical level

- Logical Level: **Tables** that make up the database

  Users interact with logical layer using SQL

  Translates SQL request into disk I/O read/writes, etc

  Ex: Select * from Employee where name = "Kelly";

  Only **one** logical Database

- External Level: **Views** of the Database

  Upper most level in the database architecture, it implements the concept of abstraction as much as possible

  Hides the working of the database from users.

  It maintains security of database by giving users access only to the data which they need at a particular time. Any data that is not needed will not be displayed.

  User interested in some portion of the Database, user creates View

Data Independence

Physical Data Independence means that changes in the physical storage of the Database have no effect on the logical Database

E.g. Change
- From C: drive to F: drive,
- From sequential access file to hash file

Change physical access but no change to logical database.

Achieved by buffering (i.e. hiding) logical Database from physical Database

Logical Data Independence is where changes are made to logical layer without having to change external schema / application
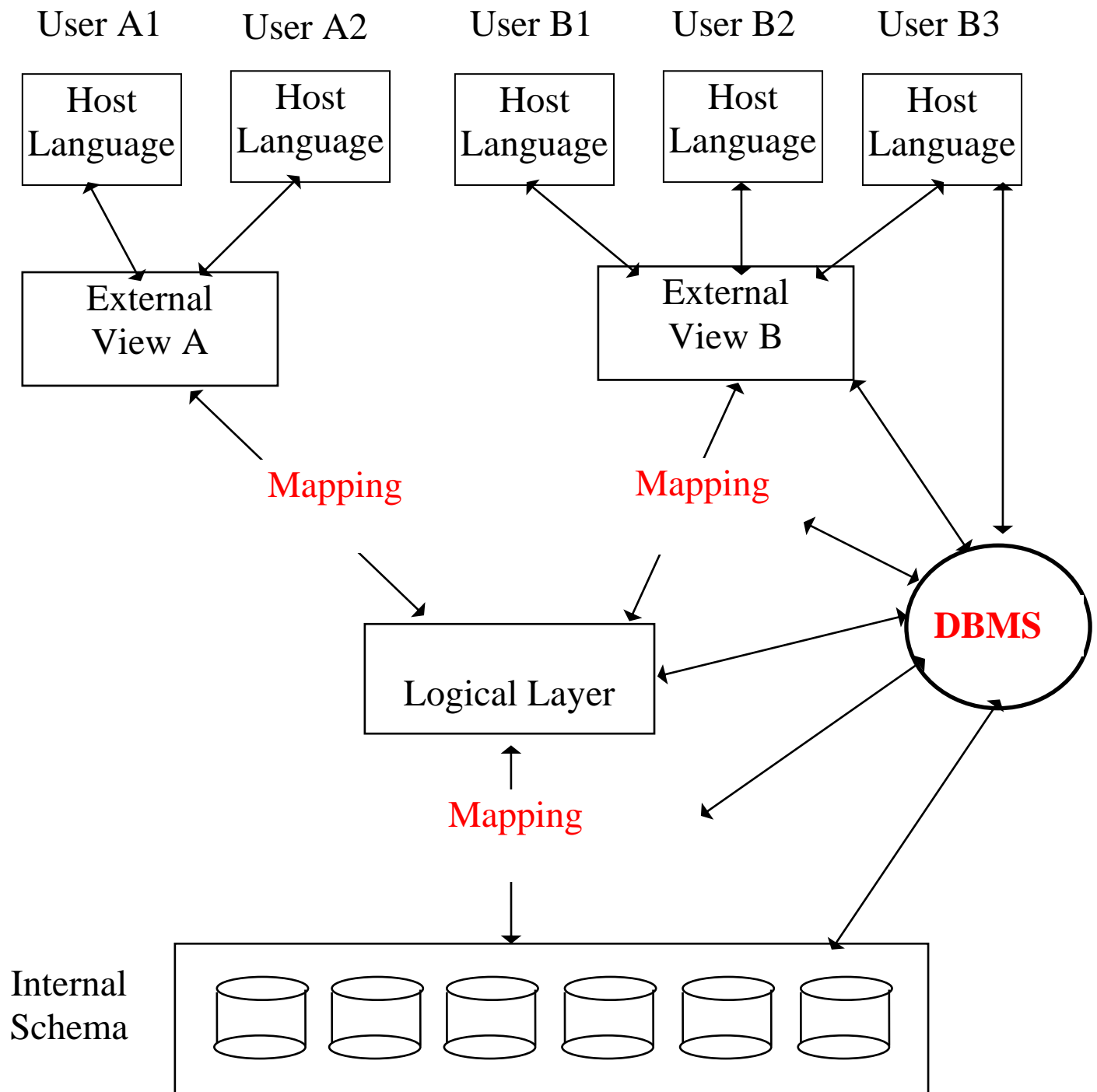
E.g. Add new column, change data type of column

Achieved by buffering (i.e. hiding) logical Database from Application.

Reduces the costs involved in software maintenance

Without it, many programs would have to be modified and recompiled every time logical Database was reorganised

External Layer
Views

Views are **Stored Queries**, when query run it produces a result set, Views rebuild every time query executed

Views are an organisational tool, not a performance enhancement tool, primarily used for convenience and security

View : Create view V As
   ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE
   Select Col-1, Col-2 from T where Col-3 > 100;

Table T

Col-1	Col-2	Col-3
10	~	101
20	~	103
30	~	66

View V

Col-1	Col-2
10	~
20	~

Query : Select * From **V**;

Query : Select * From **V** where Col-1 >=10;

Simple Views are updateable, depends on the *WHERE* conditions in view, Complex Views are not updateable

Query : Select * From V;

What actually gets executed if ….

Merge :
> Select Col-1, Col-2 from T where Col-3 > 100;

TempTable:
> Select Col-1, Col-2 from T where Col-3 > 100;

Query : Select * from V where Col-1 < 100;

What actually gets executed is ….

Merge :
> Select Col-1, Col-2 from T where (Col-3 > 100) and
> (Col-1 < 100);

TempTable:
> Select Col-1, Col-2 from T where Col-3 > 100;
> [TEMP TABLE]
> Select * from [TEMP TABLE] where Col-1 < 100;

Advantages of Views

- **Limits data** access to specific users. Use View to expose only non-sensitive data

- View provides **extra security** layer

- **Simplify complex queries**, Views can hide the complexity of underlying tables to the end-users. View can use simple SQL statements instead of complex statements with many joins.

- Enables computed columns.

Disadvantages of Views

- **Performance**: Querying data from a database view can be slow especially if the view is created based on other views.

- **Tables Dependency**: If you change the structure of those tables that view associated with, you have to change the view as well.

View : Create view V as
Select Col-1, Col-2 from T where col-3 > 100;

Table T

Col-1	Col-2	Col-3

View V

| Col-1	Col-2

Logical Layer
Relational Data Model (RDM):

Database is a set of relations (i.e. tables) in which data is stored in rows (tuples) and columns (fields)

Data perceived as being in table but at physical level data can be stored in any one of the following file types (Storage Engines):

- Sequential Files ISAM, myISAM,
- Index Files
- Hash File
- Pointer Chains
- CSV, TSV

File type will be determined by **Storage Engine** choosen by user when creating table - MySql default InnoDB
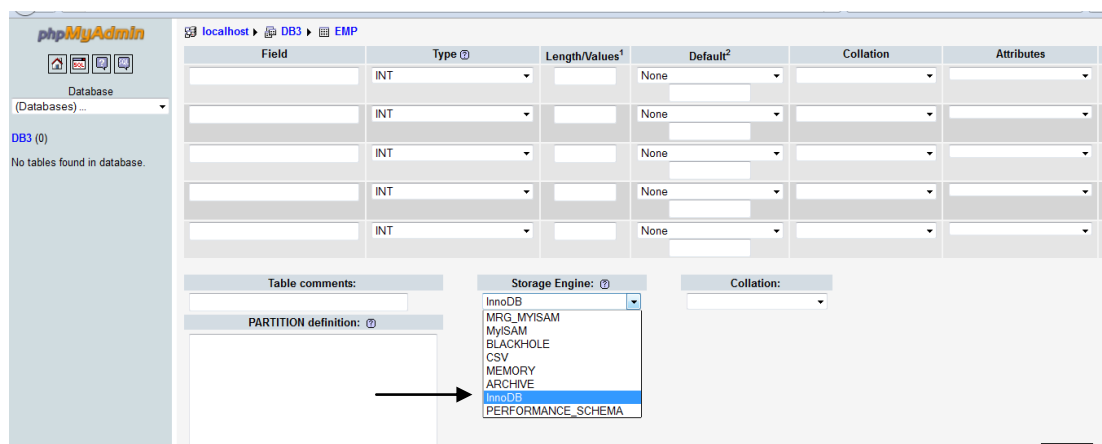
Create Table Emp (

......

) Engine=InnoDB;

Relational databases support *dialects* of SQL, '99,

- DDL - Create Table
- DML - Select, Insert, Update, Delete

Database Design

Type of Database : Static vs Dynamic
Type of query: Student Id, etc

- **Tables**
- **Key Types**

    Primary Key: Attribute or *group of attributes* that uniquely identifies a tuple/row/record within a table

    Composite Key: Primary key that contains more than one attribute / field

    Candidate Key: Attribute that 'could' be chosen as the primary key, not generally common

    Foreign Key: Attribute that is the primary key of another table, Used to implement **Referential Integrity** and relationships between tables

- **Normalisation**
    - Anomalies
    - Functional Dependence
    - Multi-Valued Attributes
   - Repeating Group
    - Normal Forms

Foreign Keys
Relationships / constraint

Referential Integrity is a database concept that ensures that <u>relationships</u> (and by implication data) between tables remain <u>correct</u> and <u>consistent</u>

Any Foreign Key field must agree with the primary key that is referenced by the foreign key

Foreign Keys exists as a <u>constraint</u> on the table to stop you from inserting something that doesn't point to anything – **Referential Integrity**

Customer
(Parent)
Primary Key

One to One
One to Many
Many to Many

Customer_Sales
(Child)
Foreign Key
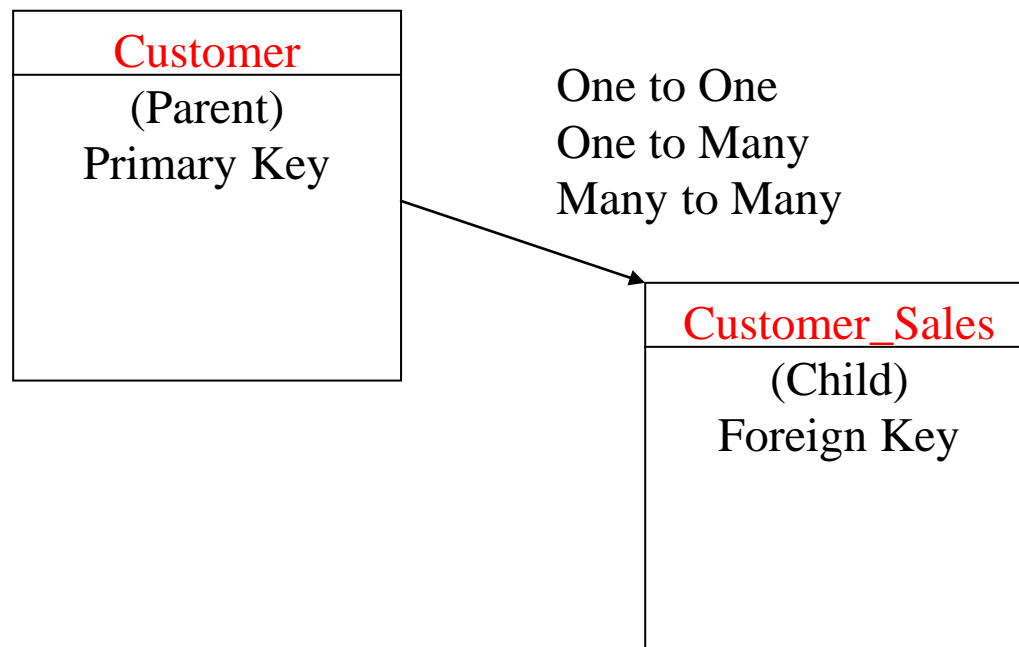
When a table (Child) has a Foreign Key relationship to another table (Parent), the concept of **Referential Integrity** states that you cannot insert a record in the Child table unless there is a corresponding record in the Parent table

Create Table **Customer**

(Customer_Id Integer(10),

   Name varchar(30),

   Primary Key (Customer_Id));

Create Table **Customer_Sales**

(Transaction_Id Integer(2),

   Amount   Decimal(7,2) zerofill,

   Customer_Id   Integer(2) UNIQUE

   Tx_Date TIMESTAMP

   Primary Key(Transaction_id),

   FOREIGN KEY (Customer_Id) references Customer(Customer_Id)

   ON DELETE RESTRICT **ON UPDATE** RESTRICT **);**

Customer	
Customer_Id (Primary Key)	Name
100	Kelly
200	Smith
300	O'Sullivan
400	O'Byrne

1: Many relationship

Customer_Sales			
Tx_Id	Amount	Customer_Id (Foreign Key)	Tx_Date
1	23.50	100	15/4/16
2	80.99	200	5/1/16
3	23.50	300	8/2/16
4	80.99	400	8/3/16

Customer_Id is Primary Key in the **Customer** Table- PK
Customer_Id is Foreign Key in the **Customer_Sales** Table

Referential Integrity only stipulates that there is a corresponding data value in the Foreign Key column,

Possible that some of the other columns (Amount,Tx_Date) can change or be set to NULL

Insertions

- Possible to Insert into the parent table no issue
- Possible to Insert into child table **<u>provided</u>** there is a corresponding entry in the Parent table
- Cannot insert into child if no parent exists

Deletion

- Can delete parent if no child exists
- Can delete child even if a parent exists
- If we try to delete parent (pk, i.e. row) and child exists, three scenarios possible
  1. **Restrict**: Default, cannot delete parent row
  2. **Cascade**: Delete parent row and delete matching row in child table
  3. **Set Null** : Delete parent row and set fk in child table to Null

Update

- If we try to Update parent pk and child exists, three scenarios possible
  1. **Restrict**: Default, cannot update parent
  2. **Cascade:** Update fk in child table
  3. **Set Null :** Set fk in child table to Null

mySQL Referential Actions

Innodb supports checking of foreign key constraints, other MySQL storage engines do not

If you Update / Delete **PARENT** table, can specify the resultant action on the child table :

- RESTRICT - Cannot delete or update the child if a parent exists

    Default

- CASCADE - Automatically delete or update matching rows in child table

- SET NULL - Set the child to NULL, (data column must not be declared NOT NULL)

    Possible to delete a parent and set child to null, orphaned child

In MySQL you can disable Foreign Key's with SET FOREIGN_KEY_CHECKS=0

Anomalies

Relational Database theory interested in eliminating anomalies from Database

Student_Activity

SID	Activity	Fee
100	Sking	200
150	Swimming	50
175	Squash	50
200	Swimming	50

Insertion Anomalies

Add a new activity: Tennis costs 25

Issue is cannot add tennis until some student signs up for tennis otherwise would result in blank SID attribute

Deletion Anomalies

Assume SID=175 drops out so row deleted

Issue in that information about Squash is lost

Modification Anomalies

Assume price of swimming increases to 60

Two tuples must be updated, possible to accidentally update only one tuple resulting in inconsistency in Database

Better representation would be :

Student

SID	Activity
100	Skiing
150	Swimming
175	Squash
200	Swimming

Activity

Activity	Fees
Skiing	200
Swimming	50
Squash	50

No insertion, deletion and modification anomalies in new tables created above

Functional Dependence
X→ Y

X and **Y** are two columns in a table in a database

If the value in the **X** column repeats and we always get the same value in the **Y** column, a Functional Dependency exists

If **X** is the primary key, its value will never repeat!

Y is Functionally dependant on **X**

Can say that **X** is the Determinant of **Y**, **X** Determines **Y**

Note : If **X**→ **Y**, does not imply that **Y**→ **X**

Example1:

PPSNo → Name , PPSNo → Birthdate

PPSNo is Determinant

Name, Birthday functionally dependent on PPS# as PPSNo uniquely identifies person

If PPSNo is the Primary Key, PPSNo will never repeat but <u>Functional Dependency still exist</u>

PPSNo	Name	DOB
123	Mary	8th June 1989
456	Tom	15th Jan 1991
789	John	18th July 1993

If **PPSNo** is a non-key attribute and **PPSNo** repeats in table, so also will the name, address, birthdate repeat (same X value always has same Y value(s))

Mobile No	…	PPSNo	Name	Address	DOB
		123	Mary Kelly	Galway	8th June 19..
		123	Mary Kelly	Galway	8th June 19..
		123	Mary Kelly	Galway	8th June 19..

Example 2:

ChassisNo → Make
ChassisNo → Model
ChassisNo is Determinant

Make, Model, functionally dependent on **ChassisNo** and **ChassisNo** uniquely identifies car

If **ChassisNo** primary key, no problem but if **ChassisNo** not primary key repeating data (same X value same Y value)

PPS#	….	**ChassisNo**	Make	Model	…
		456	Ford	Mondeo	
		456	Ford	Mondeo	
		456	Ford	Mondeo	

Note also Model → Make but Make X→ Model

Example 3:

IMEI_No → Manufacturer
IMEI_No → Model
IMEI_No is **Determinant**

IMEI_No uniquely identifies phone

If **IMEI_No** repeats in relation, so also will the Manufacturer, Model (same X value always has same Y value(s))

Table below : If people share a company mobile phone, same **IMEI_No** shared between different PPS no

PPSNo	….	**IMEI_No**	Manufacturer	Model	…

MultiValued Attributes

Multivalued attribute is attribute that can have more than one value for each instance

Example:
Skill attribute, skills might be reading, writing, arithmetric, etc

Symptom attribute in Doctor table, symptom might be headache, pain, temp, etc

Attribute can have a number of valid values

Repeating Group

Set of two or more attributes that are logically related

Patient Chart		
Patient no:	12345	
Patient Name:	John Burke	
Address:	3, Main St, Galway	
Date Of Visit	**Physician**	**Symptom**
10-1-98	Kelly	Sore Throat
15-2-98	Smith	Flu
23-10-98	Jones	Ear Infection

Date Of Visit, Physician, Symptom combine to make a repeating group

In addition, each attribute is also a multivalued attribute

Patient No	Patient Name	Address	Date of Visit 1	Physician 1	Symptom 1	Date of Visit 2	Physician 2	Symptom 2

Normalisation

Systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.

Normalise data by taking tables through different stages – Forms, Each form is a set of criteria (rules) that data must be adhere to

Both normalised and unnormalised relations equivalent, but normalised table easier to work with

Note

Can only perform Normalisation provided user is familiar with data, impossible to do normalisation using attribute/column names only

Higher the normal form, the more anomalies will be removed from the Database

Performance may be price paid for having no anomalies

All attributes should depend on the key, whole key and nothing but the key.

Normal Forms

1NF

Table is in 1NF form if:
1. **Atomic** (Scalar) values in each cell, no nulls
2. No Repeating Groups / Attributes
3. No Multi-valued Attributes

Course Name	Student	Teacher	Address	Module1	Module2
Computing	Mary	Sean, Kate	Galway	DBMS	Net
Computing	Tom	Sean, Kate	Tralee	DBMS	Net
Computing	Anne	Sean, Kate	Tuam	OOC	Security
Computing	John	Sean, Kate	Westport	OOC	Security

Above table contains a lot of redundancy:
- Teacher is Multi-valued Attribute
- Module1 / Module2 Repeating Group

Course Name	Student	Address
Computing	Mary	Galway
Computing	Tom	Tralee
Computing	Anne	Tuam
Computing	John	Westport

Course Name	Teacher
Computing	Kate
Computing	Sean

2NF

2NF is a superset of 1NF

Table is said to be in 2NF if every non-key attribute is fully functionally dependent on <u>all</u> parts of primary key

Note: If the primary key consists of one attribute, relation is automatically in 2NF.

<u>SID</u>	<u>Activity</u>	Fee
100	Skiing	200
100	Golf	65
150	Swimming	50
175	Squash	50
175	Swimming	50
200	Swimming	50
200	Golf	65

Fee not fully functionally dependent on Primary Key, only dependent on *Activity* part of the primary key

Every time Activity attribute repeats, corresponding Fee value repeats (if X repeats Y repeats)

To create 2NF relations from the activities relation, can divide relations/table in two

SID,Activity → Fee,   Activity → Fee

Student_Activity

SID	Activity
100	Skiing
100	Golf
150	Swimming
175	Squash
200	Swimming
200	Golf

Activity_Fee

Activity	Fees
Skiing	200
Swimming	50
Golf	65
Squash	70

Table is in 2NF form if:

1. Single ATOMIC value dependencies in each cell
2. No Repeating Attributes /groups
3. No Multi-valued Attributes
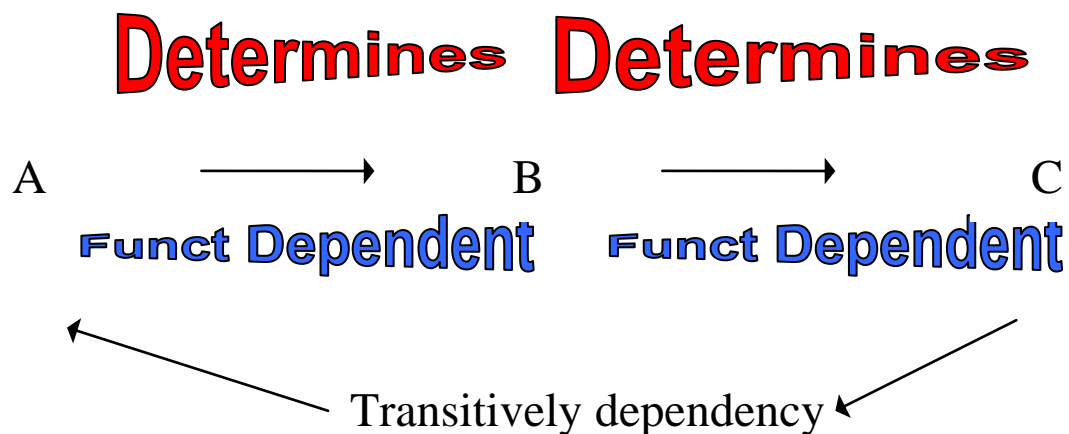4. Each Attribute is **fully** dependent on the primary key.

3NF

3NF is a superset of 2NF

3NF: A table is in second normal form (2NF) and there are **no** Transitive Dependencies.

Transitively Dependency can <u>only exist</u> when there are three columns in a table A,B,C

A,B and C are attributes of a relation such that, if A→B and B→C, then C is transitively dependent on A

Determines   Determines

A ⟶ B ⟶ C

Funct Dependent   Funct Dependent

Transitively dependency

When a <u>non-key</u> attribute determines another <u>non-key</u> attribute

Table is in 3NF if, each attribute is fully dependent on the primary key and all attributes within a table are independent, i.e. no dependencies among non-key attributes

All attributes should depend on the key, whole key and nothing but the key.

Assuming

- Each Salesperson only works in one region
- One Salesperson sells to a customer

CustId	**CustName**	**Salesperson Id**	**Region**	**Salesperson Name**
G100		10	North	
G110		20	South	
G120		20	South	
G130		30	East	
G140		40	West	
G150		40	West	

One <u>non-key</u> attribute (Salesperson) determines another <u>non-key</u> attribute (Region)

CustId $\rightarrow$ Salesperson (Functional Dependency)
Salesperson $\rightarrow$ Region (Functional Dependency)

CustId $\rightarrow$ Salesperson $\rightarrow$ Region (TD)

Boyce-Codd Normal Form (BCNF)

Relation (i.e. table) is in BCNF if every **determinant** is a candidate key

Must look throught all of the data that we have and try to identify any Determinants that are not candidate keys
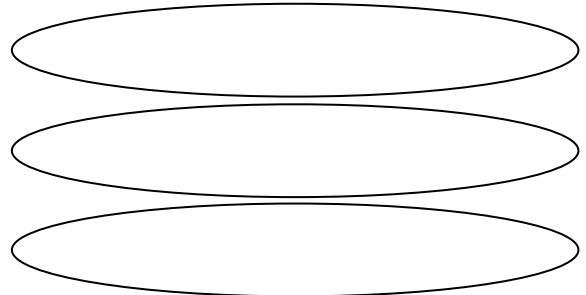
E.g.

xxxx		**Bank_Sort_Code**	**Bank Branch**
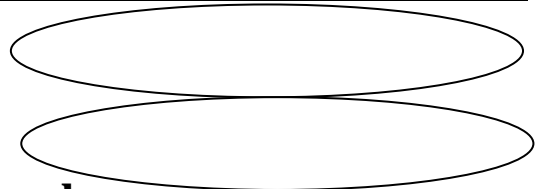
Bank_Sort_Code → Bank_Branch
Bank_Branch → Bank_Sort_Code

xxxx		**Dept_Id**	**Dept_ Name**

Dept_Id → Dept_Name
Dept_Name → Dept_Id

4NF and 5NF

Generally, first three Norm Forms are more commonly used in database modelling.

Often these two additional norm forms can lead to inefficiencies in performance and tend to be used under special circumstances or with complex data structures.

Benefits of Normalisation

Reduced redundant data resulting in a smaller database and smaller memory requirement – cost saving

Faster queries as less data to search through

Better data integrity

Less anomalies due to reduced data duplication

Physical Database Design

Main memory inappropriate for storing Database

Database organised into one or more files, each file consists of one or more records (tuples, rows), each record consists of one or more fields (attributes, columns)

When user requests tuple, DBMS maps (copies) physical record into logical record and the OS copies the physical record into buffers in main memory

Physical record (block) is unit of transfer between disk and main memory – optimised for sequential access

File organisation is **physical** arrangement of data in files i.e. records in blocks on disk

- **Heap**: Unordered, records placed on disk in no particular order
- **Sequential / Ordered / Sorted**: Records ordered by value of specified field
- **Indexed**: Unordered or ordered files
- **Hash**: Records placed on disk according to hash function

Access Method: Steps involved in sorting and retrieving records from file

File organisation
(Aka Storage Engines)

- Heap File: (Pile, Unordered file)

Simplest file organisation as records (rows) inserted into file in same order as they are inserted into table

Data not ordered, New records inserted in last page/block of file on disk

Fields and therefore rows are all of different size, *delimiters* required to separate out fields but leads to much more efficient space allocation

If insufficient space to add new record, new page/block added to file on disk

Access Method:
Insertion very efficient (i.e. fast) but …linear search required in order to access record - slow

Space from deleted records not reused, performance deteriorates, database administrator periodically reclaims unused space

< >	< >	
< >	< >	
< >	< >	
< >	< >	
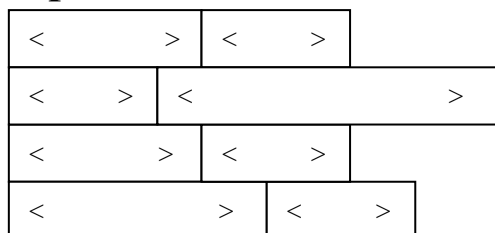
Tuple/row including delimiters for each field

- Sequential File (Ordered):

Store record sequentially in table according to primary key values, **Physically order of blocks on disk matches the logical order of records/rows in table**

Searching for primary **key** field fast as fields ordered sequentialy, binary searching used to locate a particular key field value ….BUT searching for **non**-**key** field very very slow as a linear search is required

Binary search is as follows:

(1)Retrieve mid page of file, check whether required record between first and last record on this page

If yes, required record is on this page

(2) If value of key field in first record of mid-page greater than search value, search value exists on earlier page

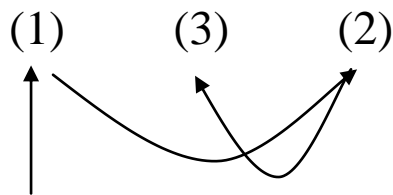Repeat above steps using lower half of file

If value of key field in last record of mid-page less than search value, search value exists on later page

Repeat above steps using top half of file

Example: Looking for value SG37

SA9	SG5	SG14	SG37	SL21	SL41

(1) (3) (2)

Binary search more efficient than linear search

Insertion into ordered file difficult, as physical order must be preserved:

Find correct position
Find space to insert record
If insufficient space, move one or more records onto next page, etc

Big issue with insertions / additions to index and sequential file resulting in Sequential files being used mainly with static databases!!!

- Index Files

Indexing is a trade-off :

Faster Reads, Slower writes

Several types of index files:

1. Primary Index File :
   - Automatically generated
   - Based on primary key value
   - One primary index
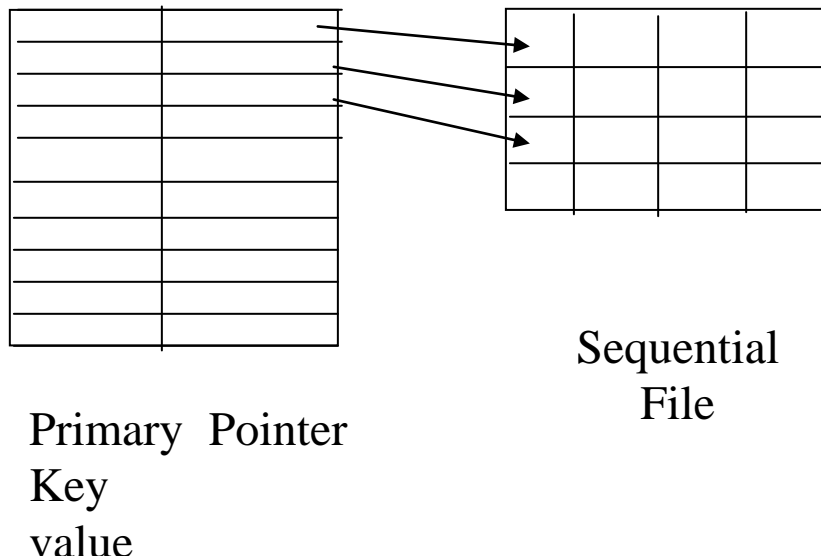
2. Secondary Index File:
   - Non-key attribute
   - Manually generated
   - Several secondary indexes

3. Clustering Index File:
   - Manually generated
   - At most one clustering index

1. Primary Index: Used for very fast retrieval of data

Primary   Pointer
Key
value

Sequential
File

ISAM : Indexed Sequential Access Method, developed by IBM, flavour used by MySQL called MyISAM

Store records/rows sequentially in table according to primary key values, Physically order of records on disk same as logical order of records in table

In addition, a second file called Index File, provides a quick lookup to determine the vicinity of actual record on disk

Index file contains two fields
- Key fields value (same value as in main sequential file)
- Pointer (i.e. address of block on disk) where full record is stored

Index file stored in main memory

Two flavours of Primary Index Files are

- Dense: Each record in main file has one <u>entry</u> in the index file, i.e. Index entry for every search key value

  Results in a very large Primary Index file

  Issues associated with space and maintenance overhead for insertions and deletions.

- Sparse: Index entries for only *some* of the primary key values in main file

  Keep an index entry only for <u>some</u> key values: typically the first key value in each block

  To find specific **key field**, index file in main memory is searched to find highest primary key value <= desired key value
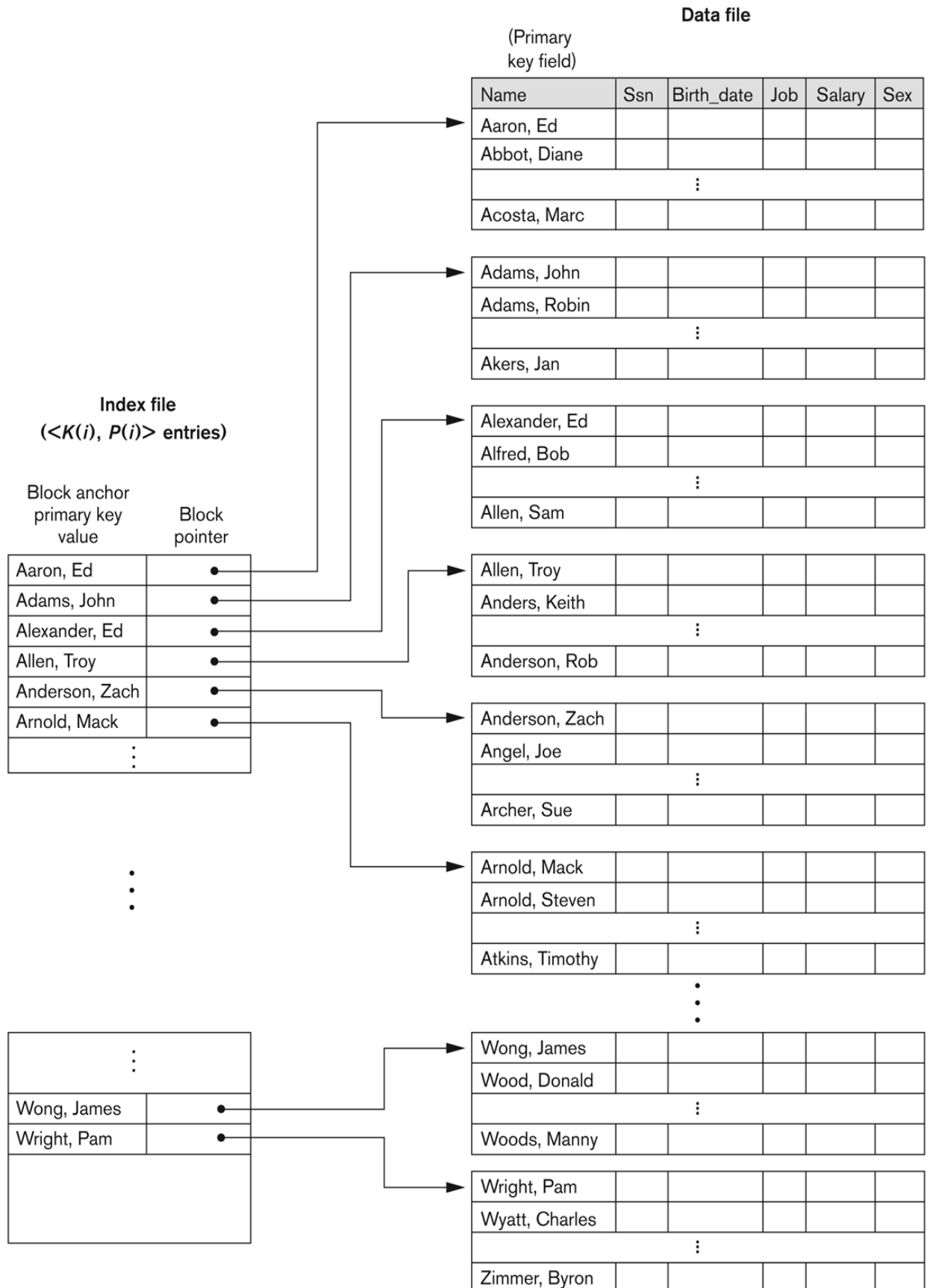
  Continue search in main sequential file at location indicated by pointer in index file

  Index file requires less space and less maintenance overhead for insertions and deletions.

  Generally slower than dense index for locating key records as two searches one in main memory and the second on disk

Sparse **Primary** Index File

Data file

(Primary key field)

Name	Ssn	Birth_date	Job	Salary	Sex
Aaron, Ed					
Abbot, Diane					
⋮					
Acosta, Marc					

Adams, John					
Adams, Robin					
⋮					
Akers, Jan					

Alexander, Ed					
Alfred, Bob					
⋮					
Allen, Sam					

Allen, Troy					
Anders, Keith					
⋮					
Anderson, Rob					

Anderson, Zach					
Angel, Joe					
⋮					
Archer, Sue					

Arnold, Mack					
Arnold, Steven					
⋮					
Atkins, Timothy					

• • •

Wong, James					
Wood, Donald					
⋮					
Woods, Manny					

Wright, Pam					
Wyatt, Charles					
⋮					
Zimmer, Byron					

Index file

($<K(i), P(i)>$ entries)

Block anchor primary key value	Block pointer
Aaron, Ed	●
Adams, John	●
Alexander, Ed	●
Allen, Troy	●
Anderson, Zach	●
Arnold, Mack	●
⋮	

⋮

⋮	
Wong, James	●
Wright, Pam	●

Comparison examples :

Main sequential file contains **1M rows**
– No Index

- **Unordered**: Requires <u>on average</u> 500,000 disk accesses to retrieve particular primary key data from disk

- **Ordered**: Requires at most 20 **disk** accesses (using binary search) to locate particular primary key value – $Log_2 (1,000,000)$ – 19.93

– Create Index

- Create sparse index file One index entry for every $1,000^{th}$ primary key

  Requires at most 10 **memory** accesses (using binary search, $Log_2 (1,000)$ to locate particular primary key value, and then at most another 10 **disk** accesses to retrieve data on disk

- Create dense index file, One index entry per primary key

  Requires at most 20 **memory** accesses (using binary search) to locate particular primary key value, and then **1 disk** access to retrieve data on disk.

Reduced number of disk accesses from 500,000 - using an unordered file - to 1 disk access when using a dense primary index file

1st Access	Eliminate	500,000		1st Access	Eliminate	500
2nd Access		250000		2nd Access		250
3rd Access		125000		3rd Access		125
4th Access		62500		4th Access		62.5
5th Access		31250		5th Access		31.25
6th Access		15625		6th Access		15.625
7th Access		7812.5		7th Access		7.8125
8th Access		3906.25		8th Access		3.90625
9th Access		1953.125		9th Access		1.953125
10th Access		976.5625		10th Access		0.976563
11th Access		488.2813				
12th Access		244.1406				
13th Access		122.0703				
14th Access		61.03516				
15th Access		30.51758				
16th Access		15.25879				
17th Access		7.629395				
18th Access		3.814697				
19th Access		1.907349				
20th Access		0.953674				

2. Secondary Index File

Performing retrievals / queries based on **non-key** attributes

SQL command to generate a secondary index:

Create Index *index_name* **On** *tablename* (*columnname*)

Ex:
    Create Index DeptNameIndex **On** dept(DName)

Sparse Secondary Index file only applicable when records are sequentially ordered based on the index column/key

Dense Index file for **Unique** data values in non-key field in table

A dense secondary index (with block pointers) on a nonordering key field of a file.
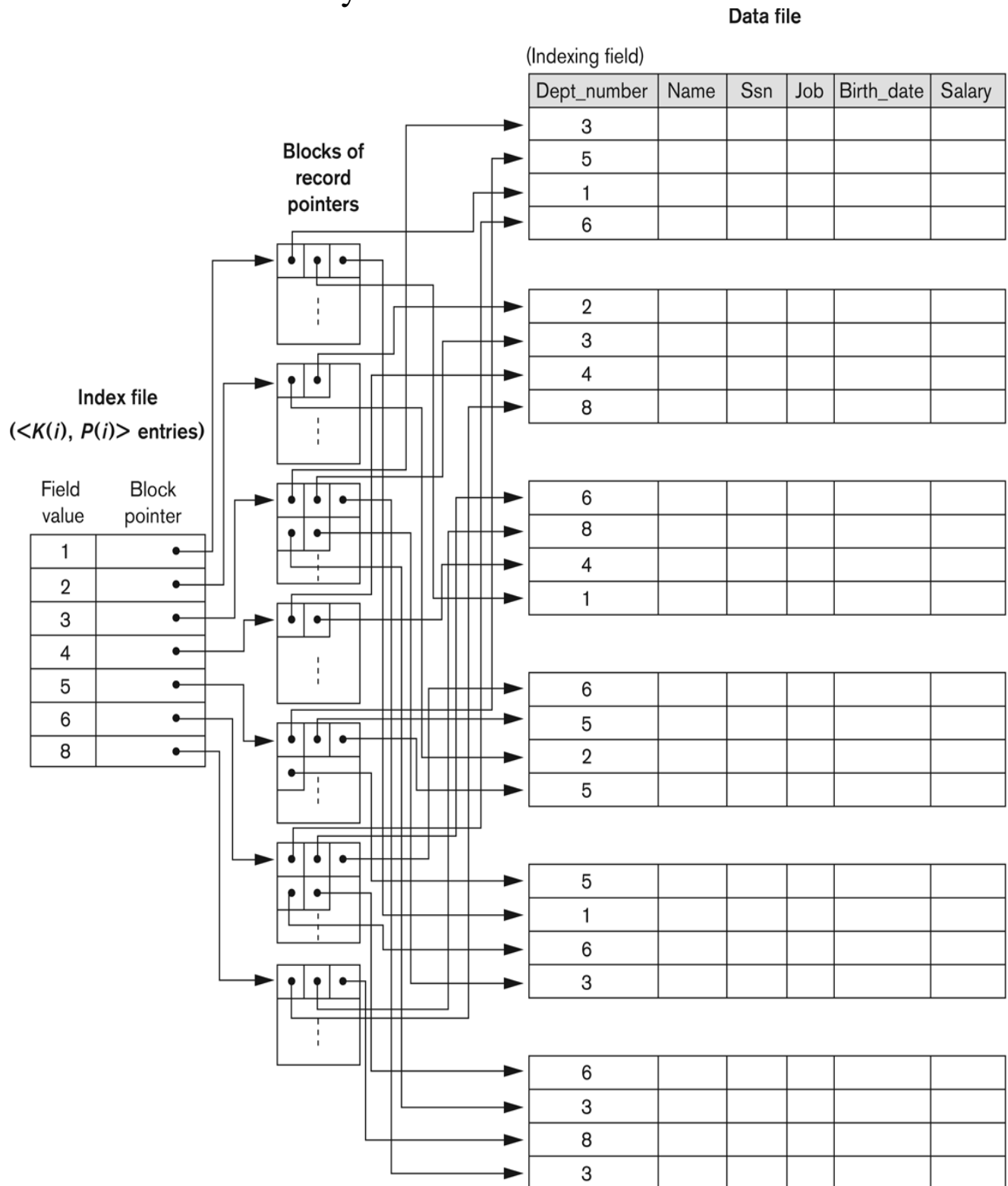
Data file

Index file
(<*K(i)*, *P(i)*> entries)

Indexing field
(secondary
key field)

Index field value	Block pointer
1	•
2	•
3	•
4	•
5	•
6	•
7	•
8	•

9				
5				
13				
8				

9	•
10	•
11	•
12	•
13	•
14	•
15	•
16	•

6				
15				
3				
17				

21				
11				
16				
2				

24				
10				
20				
1				

17	•
18	•
19	•
20	•
21	•
22	•
23	•
24	•

4				
23				
18				
14				

12				
7				
19				
22				
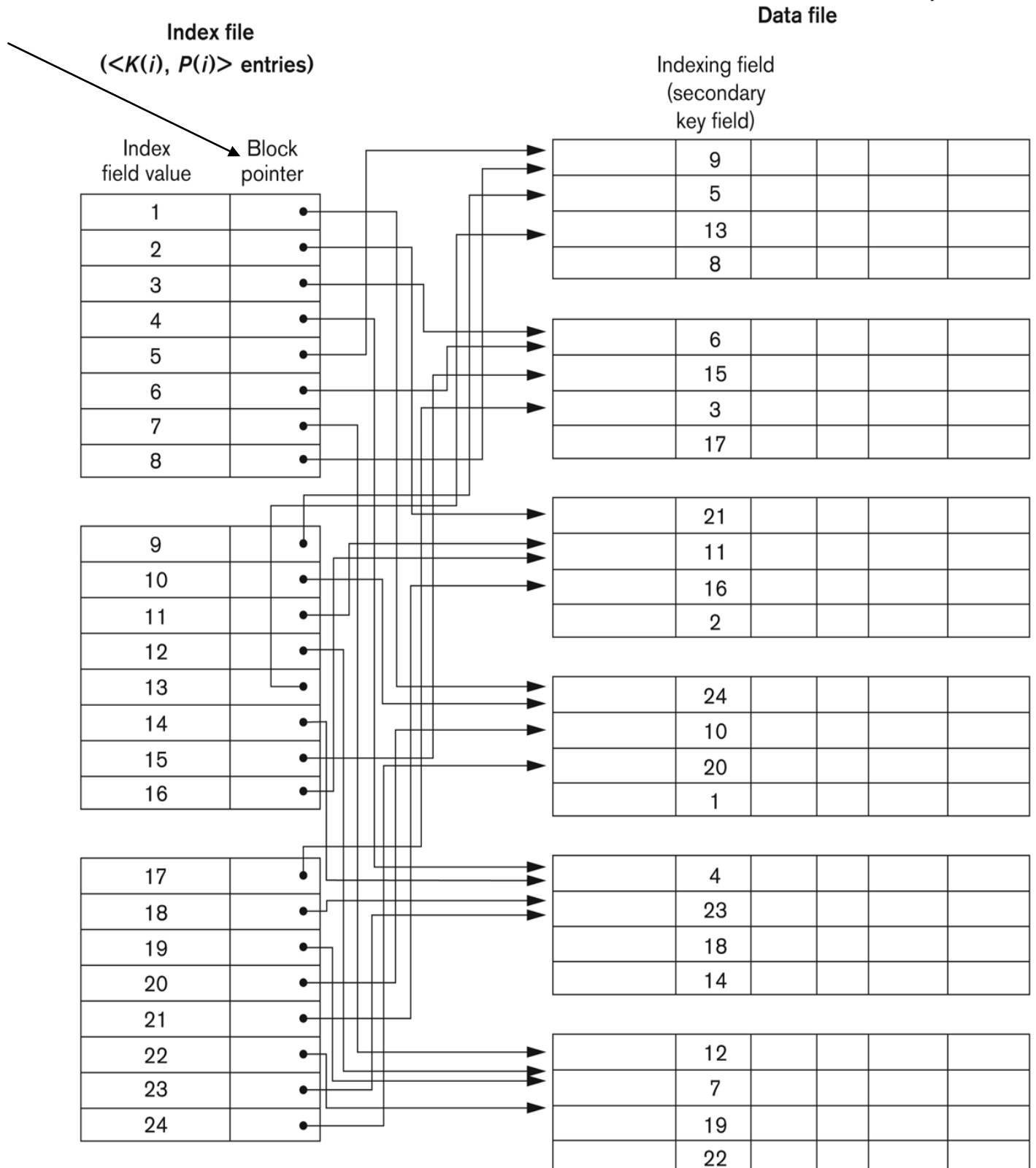
Dense Index file for **Non-Unique** data values for non-key field in table

3. Clustering Index

Clustering attribute is any attribute in a table that is used to cluster (group) together rows <u>on disk</u>

Note: If the clustering attribute is not the primary key, Clustering attribute will probably not have distinct values

Rows are physically clustered together on disk for faster retrieval, i.e. data file sequentially ordered by non-key field rather than primary key

All the tuples with identical key value are stored next to each other (i.e. in the same block on disk,)

SQL command to generate a secondary index:

Create Index *index_name* **On** *tablename* (*columnname*) **CLUSTER**;

E.g., Primary schools store data (cluster data) according to family surnames as they do a lot of searches by surname.

Queries for a particular surname will very quickly retrieve all data for all members of each family with the same surname

How MySQL Uses Indexes

Indexes are used to find records with specific column values quickly.

Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the longer this takes

MySQL indexes are stored in B-Tree (Balanced Tree)

Prefix Index : Indexes created that use only the leading part of column values, e.g.

*Create Index part_of_name ON customer (name(**10**));*

MyISAM : Index file .MY**I**

Innodb : Data files / Index files part contained in innodb file

Indexes are less important for queries on small tables, or big tables where queries process most or all of the rows.

When a query needs to access most of the rows, reading sequentially is faster than working through an index.

Sequential reads minimize disk seeks, even if not all the rows are needed for the query.

Summarise Indexes

Indexing is very important to any database.

Getting the "right" index can make a query run orders of magnitude faster but ……will slow down INSERTs a little

Hash Files

Used a lot as file type for index files

Data records not written sequentially on disk

Identify a column that will be queried most often, call this the **Hash** field / key

Hash **Function** takes as input the Hash field

The output of the **Hash function** is the address of a block on disk where hash field (and record) is stored

Records appear to be randomly distributed in hash file

Goal of hash files is to minimise time required to perform search **for a particular record** containing hash field/s

Possible to access hash record on disk with **ONE** disk access

Techniques for distributing records on disk:
❑ **Division-Remainder**: Uses MOD function and takes hash field value, divides it by some predetermined integer and uses remainder as address of disk page where record stored

❑ **Folding**: Applies arithmetic function (+) to hash field, Resultant sum used as address of disk page where record stored

Example: Hash Key 76123451001214.

Divide key into segments of size 3 digits.

761, 234, 510, 012, and 14.

Address on disk is : 761 + 234 + 510 + 012 + 14 = **1531**.

Issue with hash functions is does not a guarantee unique address - **Collision**

Collision management complicates hash function management and degrades overall performance

Several techniques to manage collisions:

❑ Open address: If collision occurs, system performs linear search to find first available slot in which to insert new record, **Marker indicates collision occured**

Query in collisions record requires iterating through all of hash file

❑ Unchained Overflow: Overflow area maintained for collisions that cannot be placed at particular address, i.e. address already occupied

Collision record is placed in an overflow area

Searching small overflow area has performance improvements over searching all hash file

Hash File
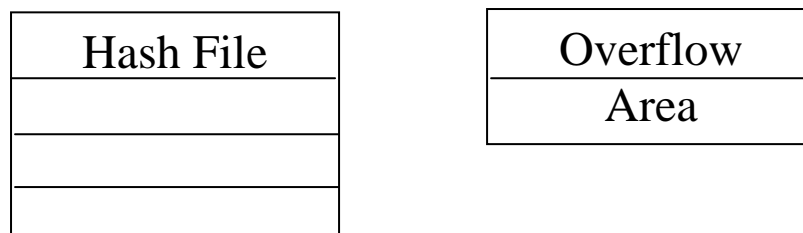
Overflow
Area

❑ Chained Overflow: Overflow area also maintained but where a collision occurs pointer (synonym pointer) within each record points to exact overflow location

Hash File	Overflow Area
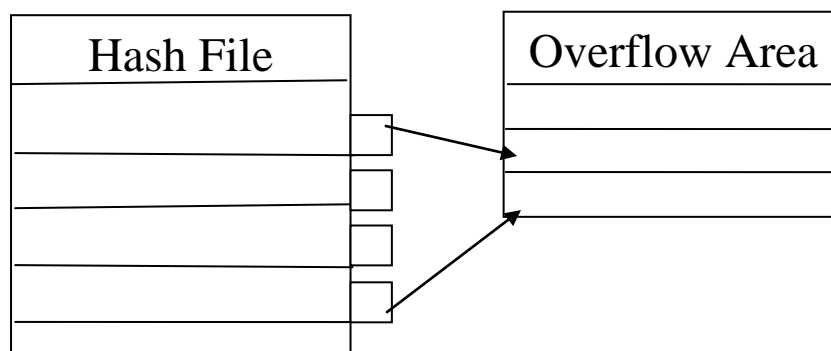
❑ Multiple Hashing: Second hashing function applied if first results is collision, second hash function generally used to place records in overflow area

Above techniques all utilise static hashing in that address space is fixed when hash file created

When space full, file saturated, Database administrator must reorganise hash structure

Alternative approach called dynamic hashing allows file size to change dynamically

Limitations of Hashing

Hashing very <u>inefficient</u> for retrievals based on fields other than key fields

Hashing inappropriate for retrievals based on pattern matching or ranges of values

MySql uses eithet B TREE or a hash file to implement indexes in main memory

Storage Engines (aka File Types) Supported by MySQL

Storage Engine	Description
InnoDB	Transaction-safe tables with row locking and foreign keys. The default storage engine for new tables. See Chapter 14, *The InnoDB Storage Engine*, and in particular Section 14.1.2, "InnoDB as the Default MySQL Storage Engine" if you have MySQL experience but are new to InnoDB.
MyISAM	The binary portable storage engine that is primarily used for read-only or read-mostly workloads. See Section 15.3, "The MyISAM Storage Engine".
MEMORY	The data for this storage engine is stored only in memory. See Section 15.4, "The MEMORY Storage Engine".
CSV	Tables that store rows in comma-separated values format. See Section 15.5, "The CSV Storage Engine".
ARCHIVE	The archiving storage engine. See Section 15.6, "The ARCHIVE Storage Engine".
EXAMPLE	An example engine. See Section 15.10, "The EXAMPLE Storage Engine".
FEDERATED	Storage engine that accesses remote tables. See Section 15.9, "The FEDERATED Storage Engine".
HEAP	This is a synonym for MEMORY.
MERGE	A collection of MyISAM tables used as one table. Also known as MRG_MyISAM. See Section 15.8, "The MERGE Storage Engine".
NDBCLUSTER	Clustered, fault-tolerant, memory-based tables. Also known as NDB. See Chapter 18, *MySQL Cluster NDB 7.2*.

In-memory.

Sometimes referred to as a main-memory database system, an in-memory DBMS relies mostly on memory to store data.

The primary use for the in-memory database is to improve performance.

As data is maintained in memory, as opposed to on a disk, I/O latency is greatly reduced.

Mechanical disk movement, seek time and buffer transfer can be eliminated because the data is immediately accessible.

An in-memory database can also be optimized toaccess data in memory; a traditional DBMS is optimized to access data from a disk.

Comparison of different Storage Engines
(File Types)

1M records in a table in the database

Table below indicates the **number of disk read accesses required** when querying for a particular **primary key** value

Sparse index file points to ever 1000th row

Hash	Heap	Sequential	Sequential	Sequential	Sequential	Sequential
		No Index	Dense **Primary** Index	Sparse **Primary** Index	Dense **Secondary** Index	Sparse **Secondary** Index
1	500,000 On Average	20, Worst Case	1	10, Worst Case	N/A	N/A

1M records in a table in the database

Table below indicates the number of **disk** access required when querying for a **non key** value

Sparse index file points to ever 1000th UNIQUE row

Hash	Heap	Sequential	Sequential	Sequential	Sequential	Sequential
		No Index	Dense Primary Index	Sparse Primary Index	Dense Secondary Index	Sparse Secondary Index
500,000 On Average	500,000 On Average	500,000 On Average	N/A	N/A	1	500,** Worst Case

** Assuming Sparse Secondary Index file only applicable when records are sequentially ordered on disk based on the index column/key

Table Size on disk and in Main Memory

1M records the table
Each row 1000 bytes, Data size = 1,000MB = 1GB
Primary Key 10 bytes, Index file = 10MB
Sparse index file points to ever 1000th row

	Hash	**Heap**	**Sequential**	**Sequential** Dense **Primary** Index	**Sequential** Sparse **Primary** Index	**Sequential** Dense **Secondary** Index	**Sequential** Sparse **Secondary** Index
Main Memory	0	0	0	10M	10K	10M,10M, 10M, etc	10K,10K,10k, etc
Disk	1.5GB	750M	1GB	1GB	1GB	1GB	1GB

Hash file larger to accommodate collisions

Inserting data into a database

Hash	Heap	Sequential	Sequential Dense **Primary** Index	Sequential Sparse **Primary** Index	Sequential Dense **Secondary** Index	Sequential Sparse **Secondary** Index
Slow, Collisions	Super Fast	Slow, Order	Slow, Order	Very Slow, Order and select 1000th data	Slow, Order data	Very Slow, Order and select 1000th data

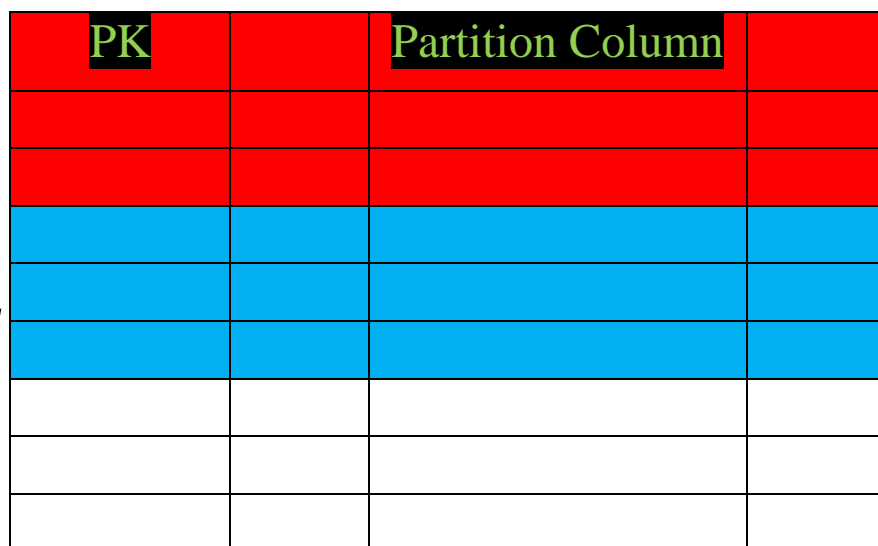http://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Partitioning

Partitioning allows user to <u>split</u> a table across multiple files, and possible multiple drives using a partitioning function.

Partitioning is normally done for manageability, performance or availability reasons.

Partitioning only supported in Mysql by MyISAM , and Innodb storage engines

User viewpoint, One table, insert into one table but physically table partitioned between files / drives

PK		Partition Column	

Select * from table
where colx between 20 and 50

Table partitioned bases on a columns
- Value
- Range
- Hash

This is known as **horizontal partitioning**—different rows of a table may be assigned to different partitions.

MySQL does not support **vertical partitioning**, where different columns of a table are assigned to different partitions.

Once partitioned it is possible to backup complete table or individual Partitions using mysqldump utility

If a table has no primary key, user may use any column or columns for the partitioning expression

When partitioning a table **either**
- Has no PK
- PK must be part of the partition function

Possible to

- **Reorganize partition** –split or merge Partitions
- **Remove partitioning –** reverts to an unpartitioned table

Benefits of Partitioning

- Tables can be assigned to specific folders/drives

- Performance improvement as

  ❖ Queries can be Parallelised

  ❖ Partitions that do not satisfy a certain rule are not scanned, this is called **Partition Pruning.**

- Data easily removed by dropping the partition containing only that data.

- Makes single inserts and selects faster

- Store historical data efficiently

Pruning :

Don't scan partitions where there can be no matching values / Data

Exclude non-matching partitions

```
drop table if exists employees ;

create table employees (
    id int not null,
    fname varchar(30),
    lname varchar(30),
    hired date not null default '2016-01-01',
    job_code int,
    store_id int )

    partition by range ( year(hired) ) (
    partition p0 values less than (2010) data
directory='c:/wamp/p0',
    partition p1 values less than (2012) data
directory='c:/wamp/p1',
    partition p2 values less than (2014) data
directory='c:/wamp/p2',
    partition p3 values less than maxvalue data
directory='c:/wamp/p3'
);

insert into employees  (id,fname,lname,hired)
values(1,Mary,'Kelly',curdate());
```

/*

Partitions allow performance improvements e.g.:

Dropping old data by simply:

Alter Table Employees Drop Partition p0;

Database can speed up a query like this:

Select Count(*) From employees WHERE hired between '2015-01-01' and '2015-12-31'   group by store_id;

Knowing that all data is stored only on p2 partition.

*/

```
Create Table expenses (
     expense_date date not null,
     category varchar(30),
     amount decimal (10,3)
);
Alter table expenses
partition by list columns (category)
     (
     partition p01 values in ( 'Lodging', 'Food'),
     partition p02 values in ( 'Flights', 'Transport'),
     partition p03 values in ( 'Leisure', 'Customer
     Entertainment'),
     partition p04 values in ( 'Communications'),
     partition p05 values in ( 'Fees')
     );

Insert into expenses values (curdate(), 'Food', 200.50);
Insert into expenses values (curdate(), 'Flights', 2000.00);
 Insert into expenses values (curdate(), 'Leisure', 100.00);
```