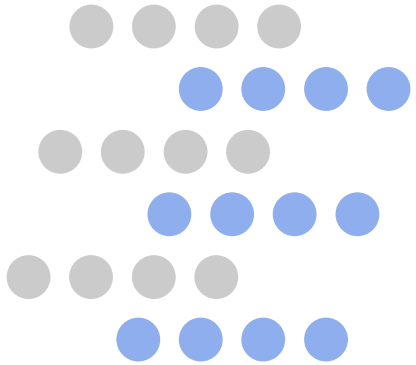
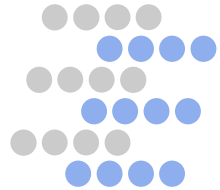
# Introduction to RDBMS

*Credits: lectures notes from J. Hellerstein, Berkeley*

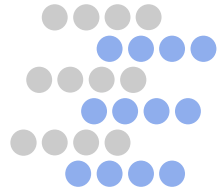


*Prof. P. Michiardi*
Pietro.Michiardi@eurecom.fr
http://www.eurecom.fr/~michiard/



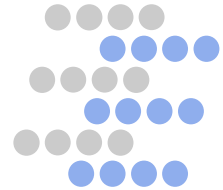
# INTRODUCTION

# So… What is a Database?



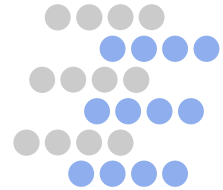
- We will be broad in our interpretation
- A Database:
  - A very large, integrated collection of data.
- Typically models the real-world
  - Entities (e.g., teams, games)
  - Relationships (e.g. The Raiders are not in the Playoffs)
- Might surprise you how flexible this is
  - Web search:
    - Entities: words, documents
    - Relationships: word in document, document links to document.
  - P2P filesharing:
    - Entities: words, filenames, hosts
    - Relationships: word in filename, file available at host

# What is a Database Management System?



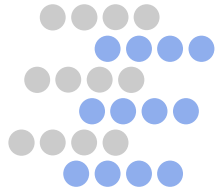
- A Database Management System (DBMS) is:
  - A software system designed to store, manage, and facilitate access to databases.
- Typically this term used narrowly
  - Relational databases with transactions
    - E.g. Oracle, DB2, SQL Server
  - Mostly for historical reasons
    - Also because of technical richness, marketing
  - When we say DBMS in this class we will usually follow this convention
    - But keep an open mind about applying the ideas!

# What: Is the WWW a DBMS?

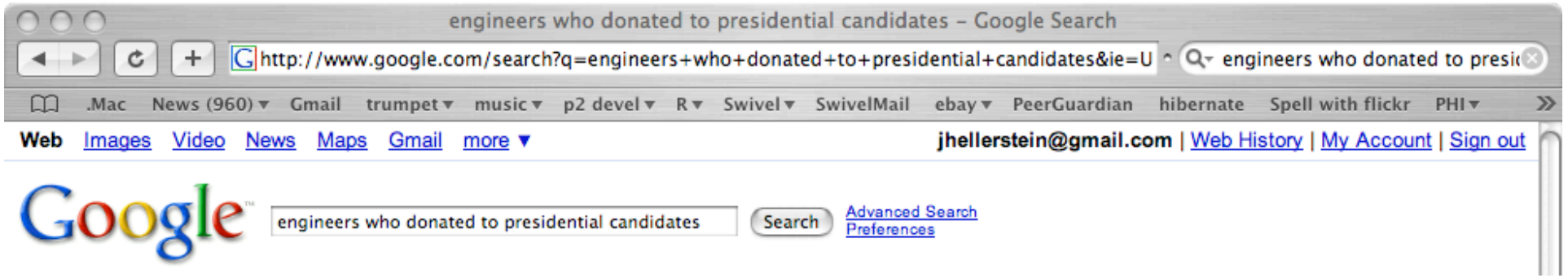


- That's a complicated question!
- The "surface web": docs and search
  - Crawler indexes pages on the web
  - Keyword-based search for pages
- Web-cache SW at Google/Yahoo is a kind of DBMS
- Notes
  - source data is mostly "prose": unstructured and untyped
  - public interface is search only:
    - can't modify the data
    - can't get summaries, complex combinations of data
  - few guarantees provided for freshness of data, consistency across data items, fault tolerance, …

# What: “Search” vs. Query

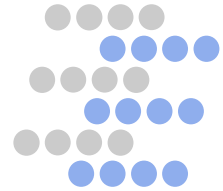


- Try actors who donated to presidential candidates in your favorite search engine.
- Now try engineers who donated to presidential candidates




- If it isn’t “published”, it can’t be searched!

# What: Is a File System a DBMS?



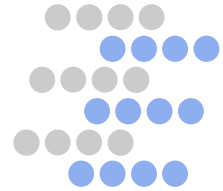
- Thought Experiment 1:
  - You and your project partner are editing the same file.
  - You both save it at the same time.
  - Whose changes survive?

**A) Yours  B) Partner's  C) Both  D) Neither  E) ???**

- Thought Experiment 2:
  - –You're updating a file.
  - –The power goes out.
  - –Which changes survive?

**A) All  B) None  C) All Since Last Save  D) ???**

# What: Is a File System a DBMS?



- Thought Experiment 1:

Q: How do you write programs over a subsystem when it promises you only "???" ?

A: Very, very carefully!!

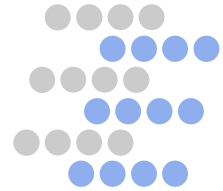
A) Y… E) ???

- …

  –Which changes survive?

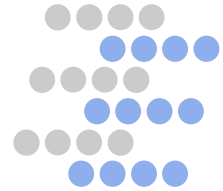
**A) All  B) None  C) All Since Last Save  D) ???**

# OS Support for Data Management



- Data can be stored in RAM
  - this is what every programming language offers!
  - RAM is fast, and random access
  - Isn’t this heaven?
- Every OS includes a File System
  - manages *files* on a magnetic disk
  - allows *open, read, seek, close* on a file
  - allows protections to be set on a file
  - drawbacks relative to RAM?

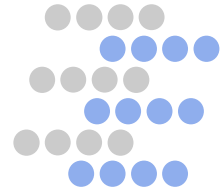
# Database Management Systems



- What more could we want than a file system?
  - Simple, efficient *ad hoc*[1] queries
  - concurrency control
  - recovery
  - benefits of good data modeling

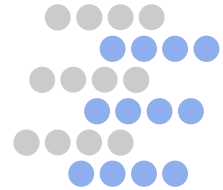
[1] ad hoc: formed or used for specific or immediate problems or needs

# Current Outlook in the Field



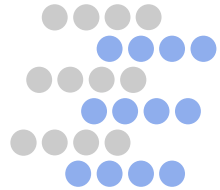
- Relational DBs a major part of the software industry
  - Elephants: Oracle, IBM, Microsoft, Teradata, Sybase, …
  - Startups: Greenplum, Aster, Cloudera, ParAccel, Vertica, …
- Obviously, Search
  - Google, Yahoo, Bing, Ask, …
- Public data services
  - Freebase, Many-Eyes, Swivel, DabbleDB
- Open Source coming on strong
  - Relational: MySQL, PostgreSQL, SQLite, Ingres, …
  - Text & Unstructured: Lucene, Hadoop
  - Key-Value stores: Cassandra, CouchDB, Voldemort, …
- Tons of applications, related industries
  - Alphabet soup!

# What systems will we cover?



- We will be broad and touch upon
  - Relational DBMS (e.g. Oracle, SQL Server, DB2, Postgres)
  - Programmable dataflow engines (e.g. Hadoop MapReduce)
  - Key/value stores (e.g. Dynamo, Cassandra, BigTable, …)
- Not all of this today though…

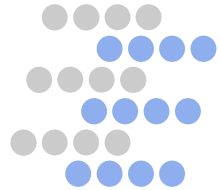
# Why take this lecture?



A. Database systems are at the core of CS
B. They are incredibly important to society
C. The topic is intellectually rich
D. A capstone lecture
E. ~~It isn't that much work~~
F. Looks good on your resume

Let's spend a little time on each of these

# Why take this lecture?



- Shift from computation to information

- Need for DB technology has exploded in the last years
  - Corporate
  - Web
  - Scientific
  - Personal

# Why take this lecture?

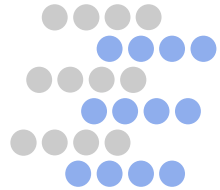
The quest for knowledge used to begin with grand theories.

Now it begins with massive amounts of data.

Welcome to the Petabyte Age.

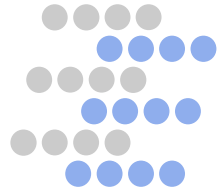
# Why take this lecture?

- representing information
- languages and systems for managing data
- concurrency control for data manipulation
- reliable data storage

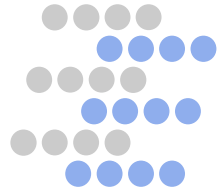


# Why take this lecture?

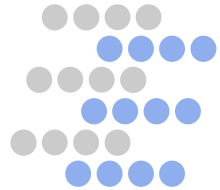
- Algorithms and cost analyses
- System architecture and implementation
- Resource management and scheduling



# Why take this lecture?

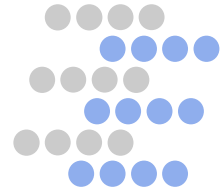


- Yes, but why?  This is not a course for:
  - Oracle administrators
  - SQL Server engine developers
- A course for well-educated computer scientists
  - Database system ideas often used “outside the box”
  - Rich understanding of these issues is basic, unusual skill.
  - If you know how a DB system *works*, you can reuse and rebuild



# QUICK OVERVIEW

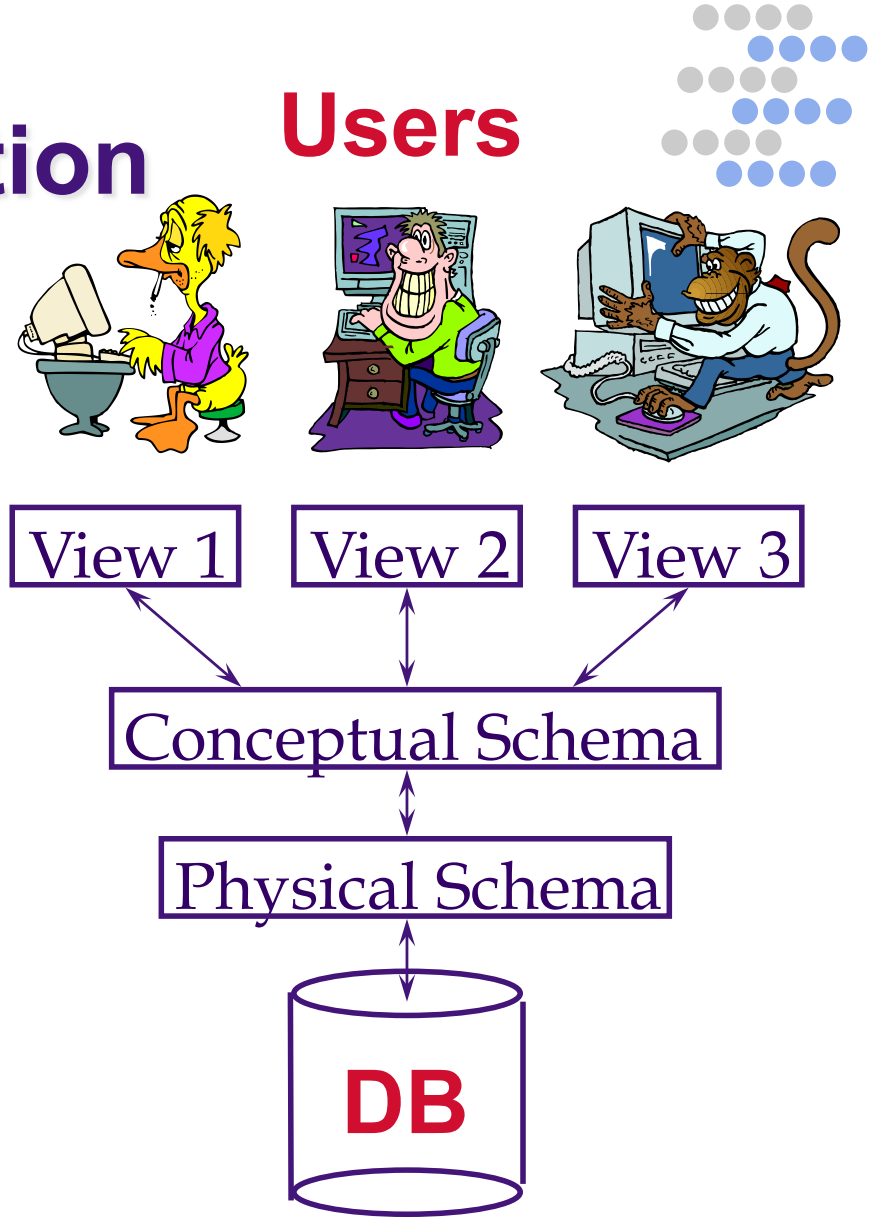
# Describing Data: Data Models



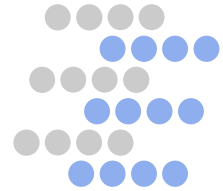
- A *data model* is a collection of concepts for describing data
- A *schema* is a description of a particular collection of data, using a given data model
- The *relational model of data* is the most widely used model today
  - Main concept: *relation*, basically a table with rows and columns
  - Every relation has a *schema*, which describes the columns, or fields

# Levels of Abstraction

- Views describe how users see the data
- Conceptual schema defines logical structure
- Physical schema describes the files and indexes used

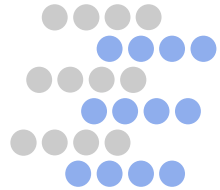


# Example: University Database



- Conceptual schema:
  - `Students(sid text, name text, login text, age integer, gpa float)`
  - `Courses(cid text, cname text, credits integer)`
  - `Enrolled(sid text, cid text, grade text)`
- Physical schema:
  - Relations stored as unordered files.
  - Index on first column of Students.
- External Schema (View):
  - `Course_info(cid text, enrollment integer)`

# Properties DBMS seek to offer



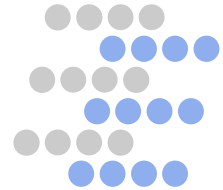
A-tomicity

C-onsistency

I-solation

D-urability

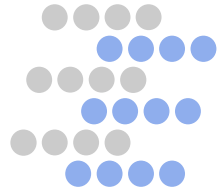
# Data Independence



- Applications insulated from how data is structured and stored
- Logical data independence: Protection from changes in *logical* structure of data
- Physical data independence: Protection from changes in *physical* structure of data

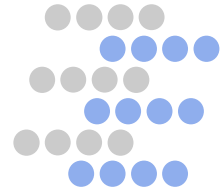
- Q: Why is this particularly important for DBMS? Because databases and their associated applications persist

# Concurrent execution of user programs



- Why?
  - Utilize CPU while waiting for disk I/O
  - Avoid short programs waiting behind long ones
    - e.g. ATM withdrawal while bank manager sums balance across all accounts

# Concurrent execution



- Interleaving actions of different programs: trouble!

  Example:

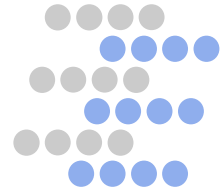
  - Bill transfers $100 from savings to checking

    *Savings –= 100; Checking += 100*
  - Meanwhile, Bill's wife requests account info.

  Bad interleaving:

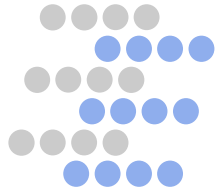
  - Savings –= 100
  - Print balances
  - Checking += 100
- Printout is missing $100 !

# Concurrency Control



- DBMS ensures such problems don't arise
- Users can pretend they are using a single-user system (called "Isolation")
  - Thank goodness!

# Key concept: Transaction

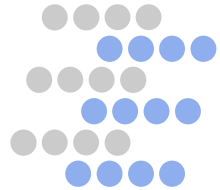


- An atomic sequence of database actions (reads/writes)
- Takes DB from one consistent state to another

# Example

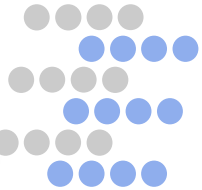

- Here, *consistency* is based on our knowledge of banking "semantics"
- **In general, up to writer of transaction to ensure transaction preserves consistency**
- DBMS provides (limited) automatic enforcement, via integrity constraints
  - e.g., balances must be >= 0

# Concurrent transactions



- Goal: execute xacts {T1, T2, … Tn}, and ensure a consistent outcome
- One option: “serial” schedule (one after another)
- Better: allow interleaving of xact actions, as long as outcome is equivalent to some serial schedule

# Ensuring Transaction Properties



- DBMS ensures:
  - ***atomicity*** **even if xact aborted (due to deadlock, system crash, …)**
  - ***durability*** **of committed xacts, even if system crashes**

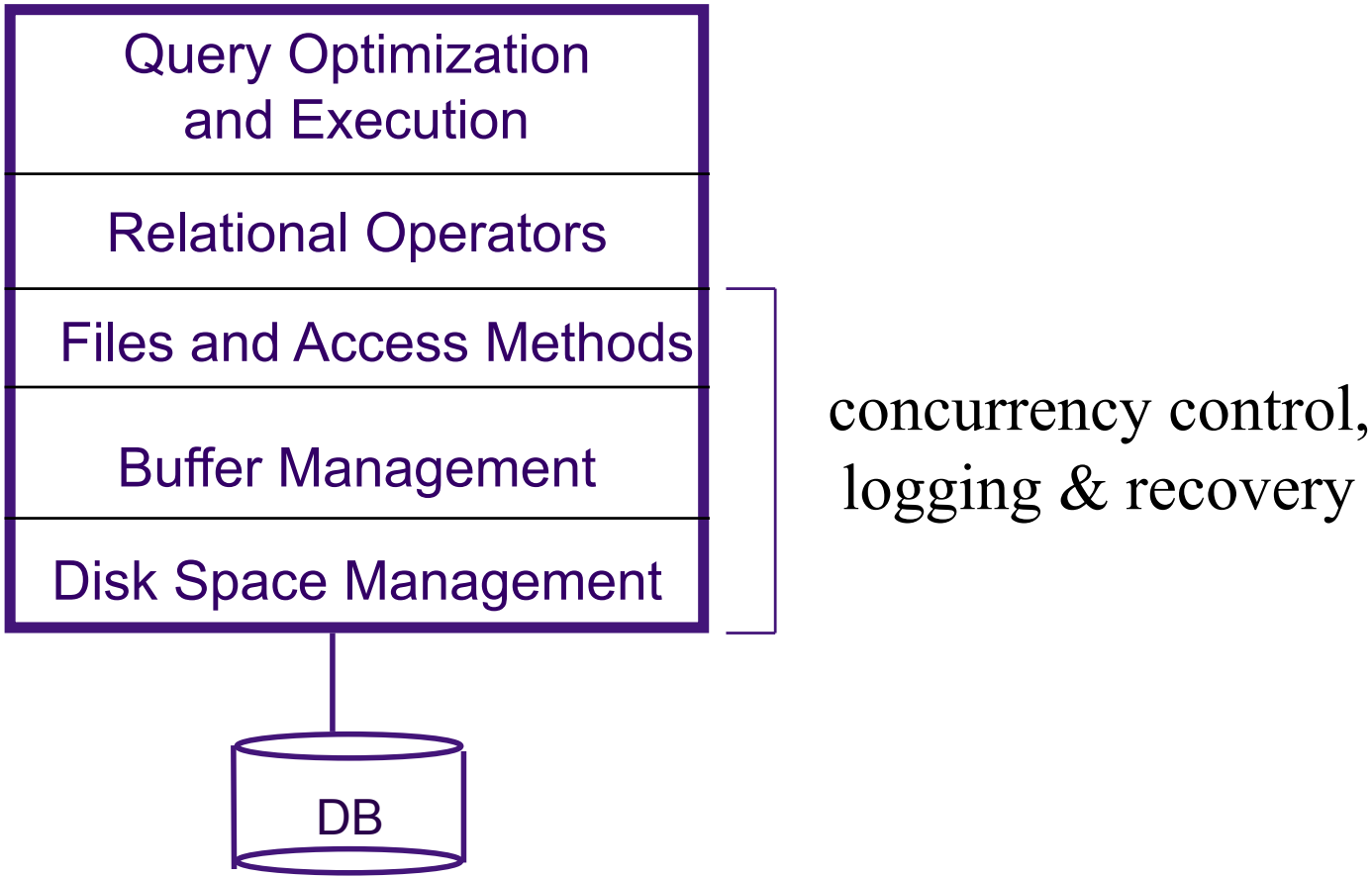
- Idea: Keep a *log* of all actions carried out by the DBMS:
  - Record all DB modifications in log, *before* they are executed
  - To abort a xact, undo logged actions in reverse order
  - If system crashes, must:

    1) **undo** partially executed xacts (ensures **atomicity**)

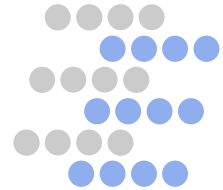
    2) **redo** committed xacts (ensures **durability**)

  - *trickier than it sounds!*

# Typical DBMS architecture



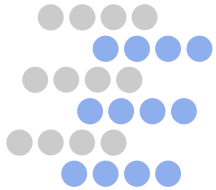
# Advantages of a Traditional DBMS

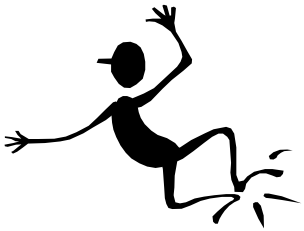


- Data independence
- Efficient data access
- Data integrity & security
- Data administration
- Concurrent access, crash recovery
- Reduced application development time

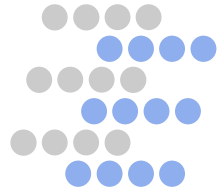
- So why not use them always?
  - Expensive/complicated to set up & maintain
  - Can be difficult to write apps above (improving, though)
  - This cost & complexity must be offset by need
  - General-purpose, not suited for special-purpose tasks (e.g. text search, matrix or time-series data, etc.)

# Databases make these folks happy ...



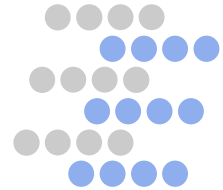


- Web & enterprise app developers
- Computing infrastructure providers
- DBMS vendors, programmers
  - Oracle, IBM, MS …
- Data analysts
- End users in many fields
  - Business, education, science, …
- Database administrators (DBAs)



# ENTITY-RELATIONSHIP DIAGRAMS AND THE RELATIONAL MODEL

# Describing Data: Data Models

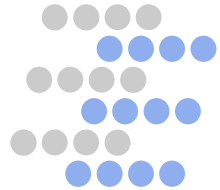


- *Data model* : collection of concepts for describing data

- *Schema:* description of a particular collection of data, using a given data model

- *Relational model of data*
  - Main concept: *relation* (table), rows and columns
  - Every relation has a *schema*
    - describes the columns
    - column names and domains

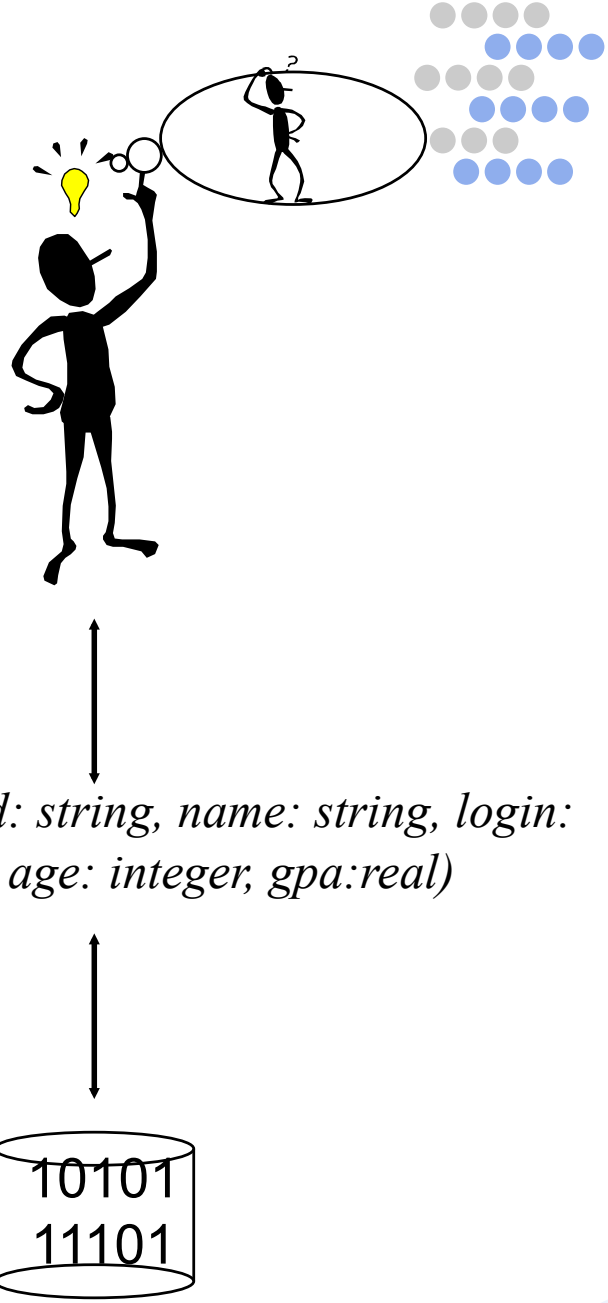
# Some Synonyms



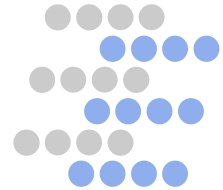
| Formal | Not-so-formal 1 | Not-so-formal 2 |
|---|---|---|
| Relation | Table | |
| Tuple | Row | Record |
| Attribute | Column | Field |
| Domain | Type | |

# Data Models

- Connect concepts to bits!
- Many models exist
- We will ground ourselves in the *Relational* model
  - clean and common
- *Entity-Relationship* model also handy for design
  - Translates down to Relational



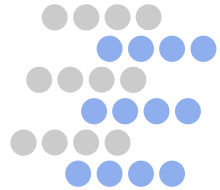
# Why Study the Relational Model?



- Most widely used model

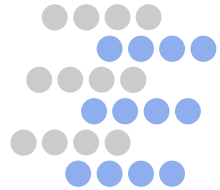
- Other models exist (and co-exist)
  - “Legacy systems” in older models
    - e.g., IBM’s IMS
  - Object-Relational mergers
    - Object-Oriented features provided by DBMS
    - Object-Relational Mapping (ORM) outside the DBMS
      - A la Rails (Ruby), Django (Python), Hibernate (Java)
  - XML
    - XQuery and XSLT languages
    - Most relational engines now “do” XML too

# Steps in Database Design



- **Requirements Analysis**
  - User needs; what must database do?
- **Conceptual Design**
  - High level description (often done w/ER model)
- **Logical Design**
  - Translate ER into DBMS data model
- **Schema Refinement**
  - Consistency, normalization
- **Physical Design** - indexes, disk layout
- **Security Design** - who accesses what, and how

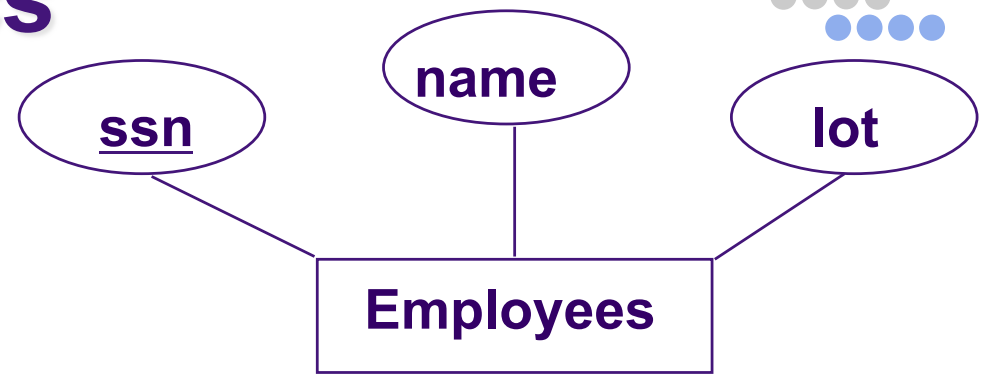
# Conceptual Design



- What are the entities and relationships?
- What info about E's & R's should be in DB?
- What *integrity constraints* (*business rules*) hold?
- *ER diagram* is the `schema'
- Can map an ER diagram into a relational schema

- Conceptual design is where the SW/data engineering *begins*

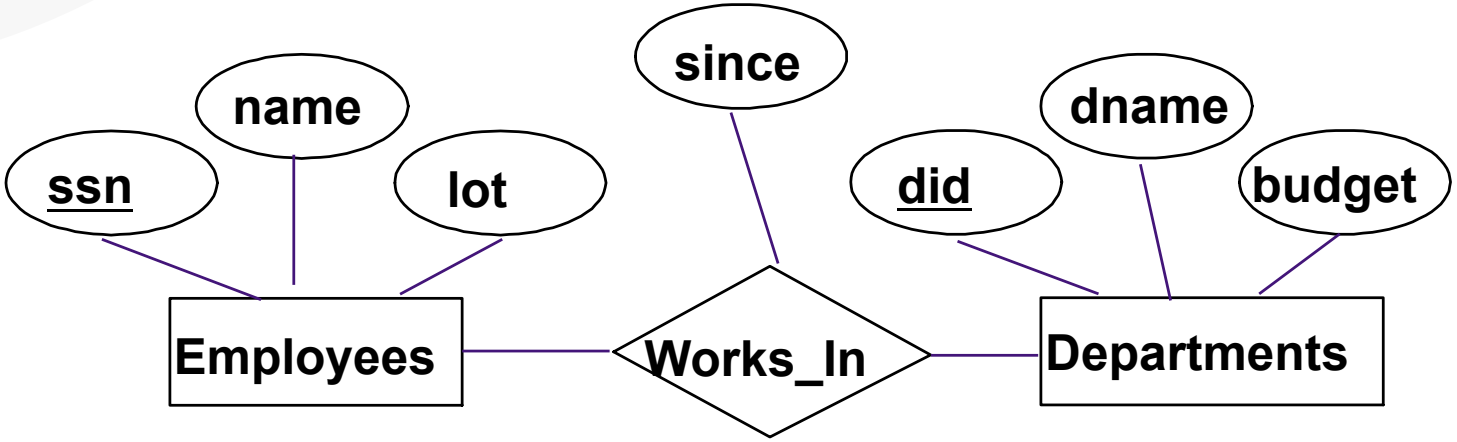
# ER Model Basics




- *Entity:*
  - A real-world object described by a set of *attribute* values.

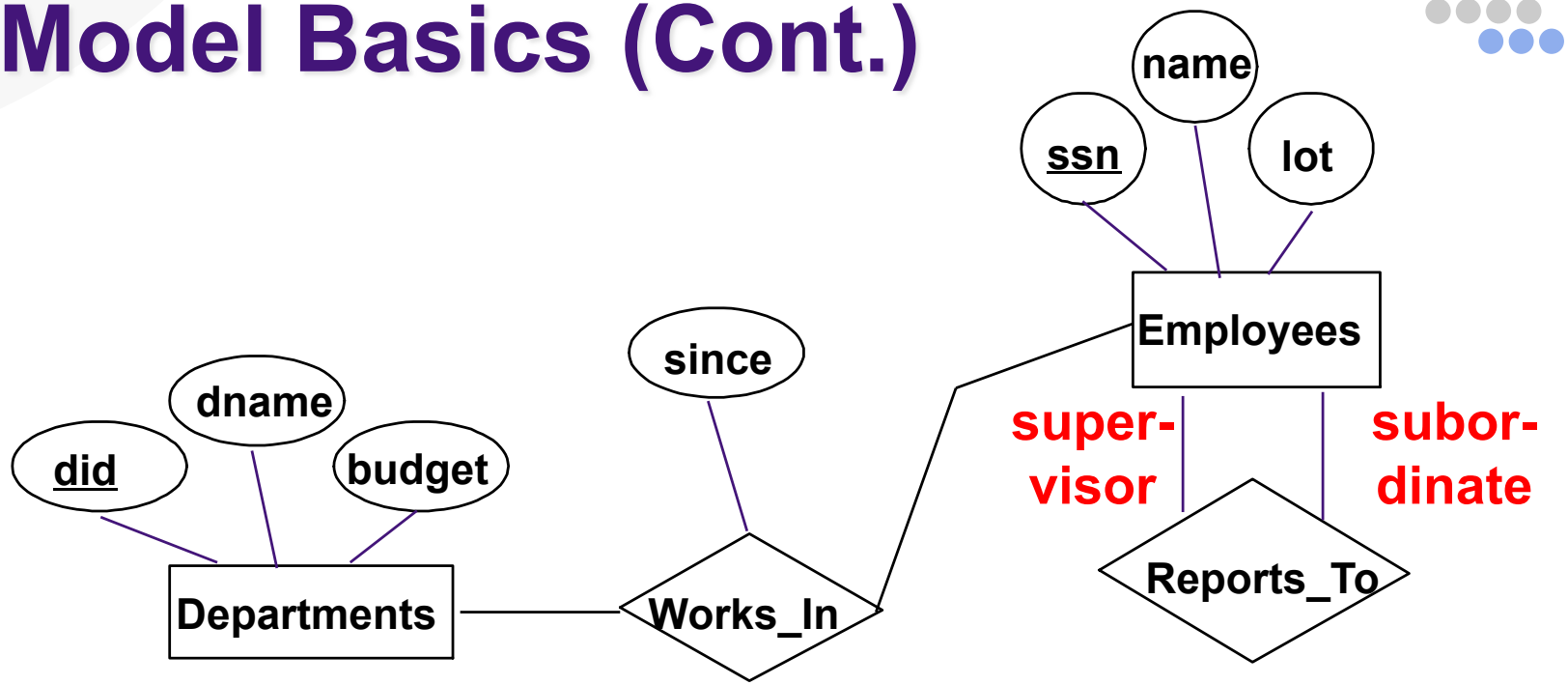
- *Entity Set*: A collection of similar entities.
  - E.g., all employees.
  - All entities in an entity set have the same attributes.
  - Each entity set has a *key* (underlined)
  - Each attribute has a *domain*

# ER Model Basics (Contd.)




- *Relationship*: Association among two or more entities.
  - E.g., Attishoo works in Pharmacy department.
  - relationships can have their own attributes.
- *Relationship Set*: Collection of similar relationships.
  - An *n*-ary relationship set *R* relates *n* entity sets $E_1 \ldots E_n$; each relationship in *R* involves entities $e_1 \in E_1, \ldots, e_n \in E_n$

# ER Model Basics (Cont.)

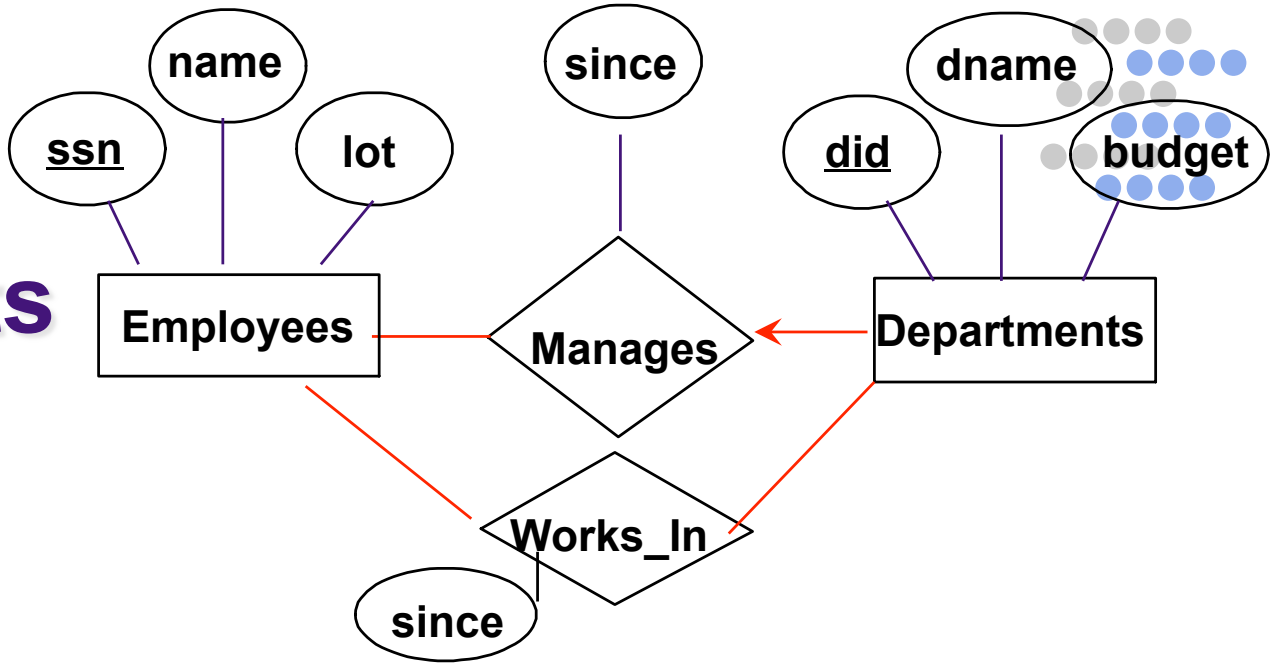


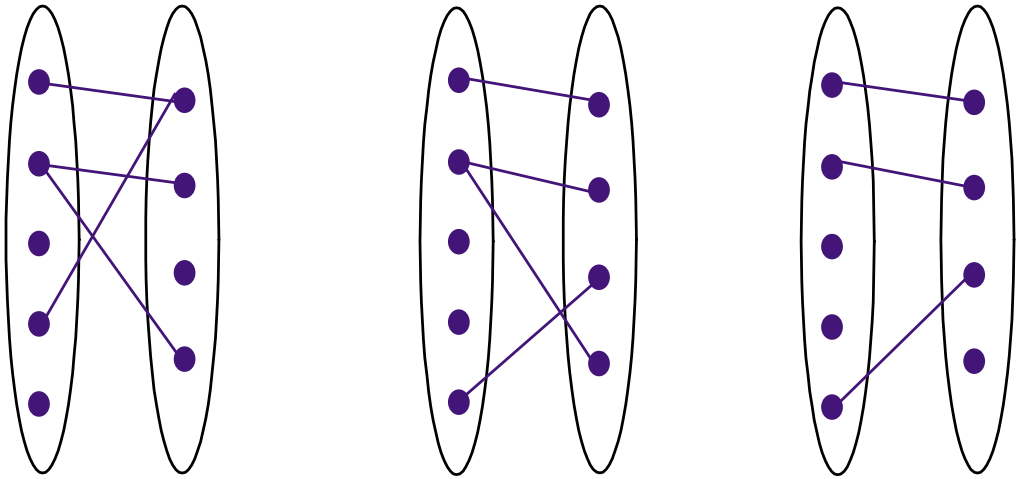

- Same entity set can participate in different relationship sets, or in different "roles" in the same relationship set.

# Key Constraints

- An employee can work in **many** departments; a dept can have **many** employees

- In contrast, each dept has **at most one** manager, according to the *key constraint* on Managers



Many-to-Many

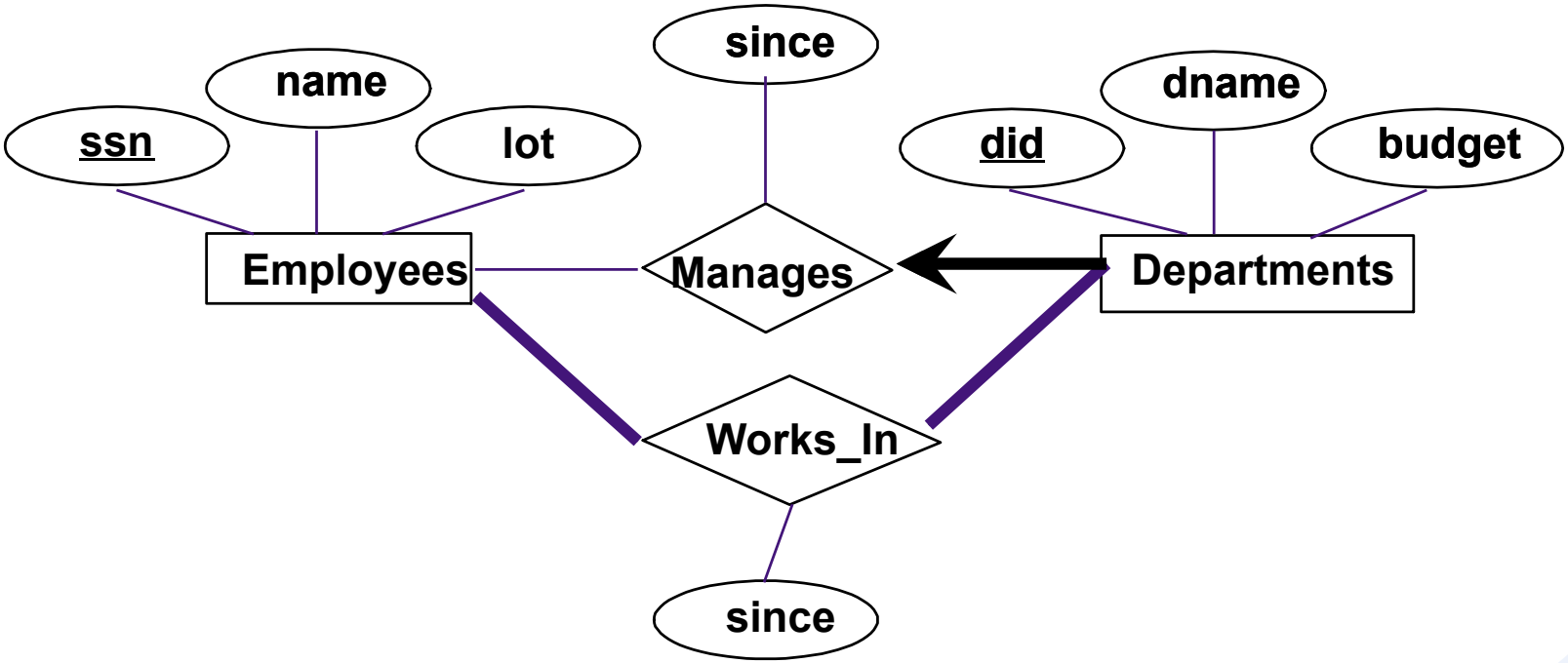
1-to-Many

Many-to-1

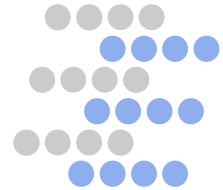
1-to-1

# Participation Constraints

- Does every employee work in a department?
- If so: a *participation constraint*
  - participation of Employees in Works_In is *total* (vs. *partial*)
  - What if every department has an employee working in it?
- Basically means “at least one”

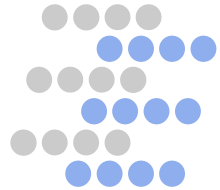


# Summary so far



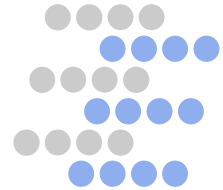
- Entities and Entity Set (boxes)
- Relationships and Relationship sets (diamonds)
- Key constraints (arrows)
- Participation constraints (bold for Total)

These are enough to get started…



# RELATIONAL DATABASES

# Relational Database: Definitions



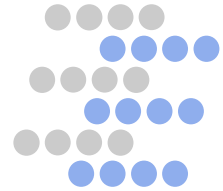
- Relational database: a set of relations
- Relation: made up of 2 parts
  - Schema : specifies name of relation, plus name and type of each column.
    - E.g. Students(sid: string, name: string, login: string, age: integer, gpa: real)
  - Instance : a table, with rows and columns
    - #rows = *cardinality*
    - #fields = *arity* (or *degree*)
- Can think of a relation as a set of rows or tuples.
  - i.e., all rows are distinct

# Ex: Instance of Students Relation

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@eecs | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

- Cardinality = 3, arity = 5, all rows distinct
- Do all values in each column of a relation instance have to be distinct?

# SQL - A language for Relational DBs

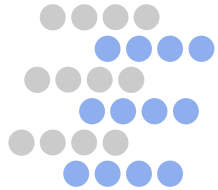


- Say: “ess-cue-ell” or “sequel”
  - But spelled “SQL”

- Data Definition Language (DDL)
  - create, modify, delete relations
  - specify constraints
  - administer users, security, etc.

- Data Manipulation Language (DML)
  - Specify queries to find tuples that satisfy criteria
  - add, modify, remove tuples

# SQL Overview



- `CREATE TABLE <name> ( <field> <domain>, … )`

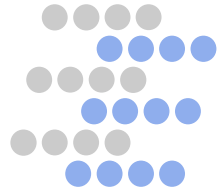
- ```
  INSERT INTO <name> (<field names>)
       VALUES (<field values>)
  ```

- ```
  DELETE FROM <name>
         WHERE <condition>
  ```

- ```
  UPDATE <name>
     SET <field name> = <value>
   WHERE <condition>
  ```

- ```
  SELECT <fields>
    FROM <name>
   WHERE <condition>
  ```

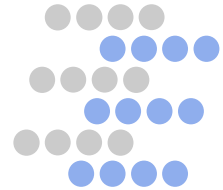
# Creating Relations in SQL



- Creates the Students relation
  - Note: the type (domain) of each field is specified, and enforced by the DBMS whenever tuples are added or modified.

```
CREATE TABLE Students
(sid CHAR(20),
name CHAR(20),
login CHAR(10),
age INTEGER,
gpa FLOAT)
```

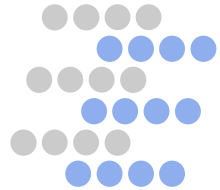
# Table Creation (continued)



- Another example: the Enrolled table holds information about courses  students take

```
CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2))
```

# Adding and Deleting Tuples



- Can insert a single tuple using:

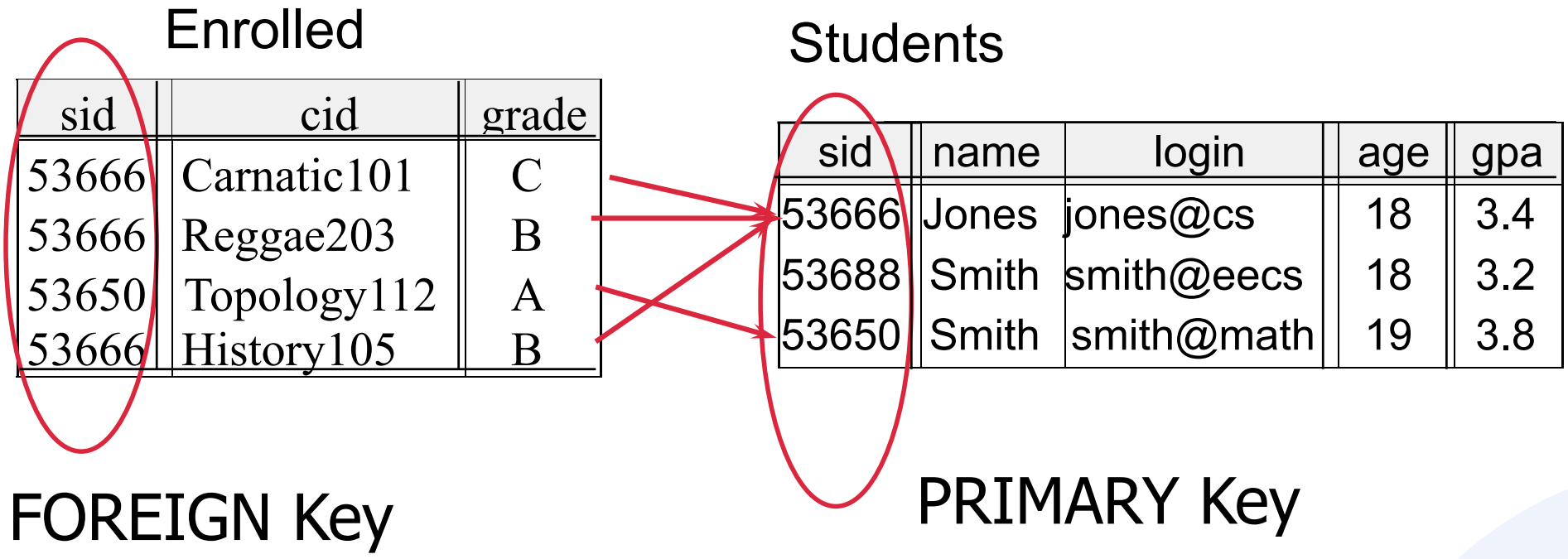
```
INSERT INTO Students (sid, name, login, age, gpa)
 VALUES ('53688', 'Smith', 'smith@ee', 18, 3.2)
```

- Can delete all tuples satisfying some condition (e.g., name = Smith):

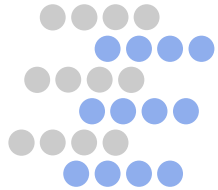
```
DELETE FROM Students S WHERE S.name = 'Smith'
```

# Keys

- Keys are a way to associate tuples in different relations
- Keys are one form of integrity constraint (IC)

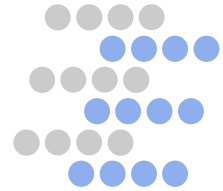


# Primary Keys



- A set of fields is a superkey if:
  - No two distinct tuples can have same values in all key fields
- A set of fields is a key for a relation if :
  - It is a superkey
  - No subset of the fields is a superkey
- what if >1 key for a relation?
  - One of the keys is chosen (by DBA) to be the primary key
  - Other keys are called candidate keys.
- E.g.
  - sid is a key for Students.
  - What about name?
  - The set {sid, gpa} is a superkey.

# Primary and Candidate Keys in SQL



- Possibly many candidate keys (specified using UNIQUE), one of which is chosen as the primary key.
  - Keys must be used carefully!

```
CREATE TABLE Enrolled1
      (sid CHAR(20),
       cid  CHAR(20),
       grade CHAR(2),
  PRIMARY KEY (sid,cid))
```

VS.

```
CREATE TABLE Enrolled2
      (sid CHAR(20),
       cid  CHAR(20),
       grade CHAR(2),
  PRIMARY KEY  (sid),
  UNIQUE (cid, grade))
```

# Primary and Candidate Keys in SQL

```
CREATE TABLE Enrolled1
      (sid CHAR(20),
       cid  CHAR(20),
       grade CHAR(2),
  PRIMARY KEY (sid,cid));
```

VS.

```
CREATE TABLE Enrolled2
      (sid CHAR(20),
       cid  CHAR(20),
       grade CHAR(2),
  PRIMARY KEY  (sid),
  UNIQUE (cid, grade));
```

```
INSERT INTO enrolled1 VALUES ('1234', 'cs186', 'A+');
INSERT INTO enrolled1 VALUES ('1234', 'cs186', 'F');
INSERT INTO enrolled1 VALUES ('1234', 'cs61C', 'A+');
```

```
INSERT INTO enrolled2 VALUES ('1234', 'cs186', 'A+');
INSERT INTO enrolled2 VALUES ('1234', 'cs186', 'F');
INSERT INTO enrolled2 VALUES ('1234', 'cs61C', 'A+');
INSERT INTO enrolled2 VALUES ('4567', 'cs186', 'A+');
```

*"For a given student and course, there is a single grade."*

# Primary and Candidate Keys in SQL

```
CREATE TABLE Enrolled1
      (sid CHAR(20),
       cid  CHAR(20),
       grade CHAR(2),
  PRIMARY KEY (sid,cid));
```

VS.

```
CREATE TABLE Enrolled2
      (sid CHAR(20),
       cid  CHAR(20),
       grade CHAR(2),
  PRIMARY KEY  (sid),
  UNIQUE (cid, grade));
```

```
INSERT INTO enrolled1 VALUES ('1234', 'cs186', 'A+');
INSERT INTO enrolled1 VALUES ('1234', 'cs186', 'F');
INSERT INTO enrolled1 VALUES ('1234', 'cs61C', 'A+');
```

```
INSERT INTO enrolled2 VALUES ('1234', 'cs186', 'A+');
INSERT INTO enrolled2 VALUES ('1234', 'cs186', 'F');
INSERT INTO enrolled2 VALUES ('1234', 'cs61C', 'A+');
INSERT INTO enrolled2 VALUES ('4567', 'cs186', 'A+');
```

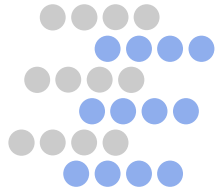
*"Students can take only one course, and no two students in a course receive the same grade."*

# Foreign Keys, Referential Integrity

- *Foreign key*: a “logical pointer”
  - Set of fields in a tuple in one relation that `refer’ to a tuple in another relation
  - Reference to *primary key* of the other relation

- All foreign key constraints enforced?
  - *referential integrity*!
  - i.e., no dangling references

# Foreign Keys in SQL



- E.g. Only students listed in the Students relation should be allowed to enroll for courses.
- sid is a foreign key referring to Students:

```
CREATE TABLE Enrolled
(sid CHAR(20),cid CHAR(20),grade CHAR(2),
  PRIMARY KEY (sid,cid),
  FOREIGN KEY (sid) REFERENCES Students);
```

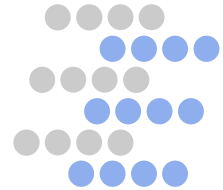
Enrolled

| sid | cid | grade |
|---|---|---|
| 53666 | Carnatic101 | C |
| 53666 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |
| ~~11111~~ | ~~English102~~ | ~~A~~ |

Students

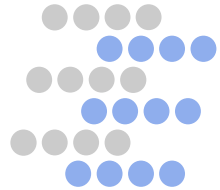
| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@eecs | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

# Enforcing Referential Integrity



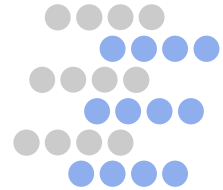
- sid in Enrolled: foreign key referencing Students
- Scenarios:
  - Insert Enrolled tuple with non-existent student id?
  - Delete a Students tuple?
    - Also delete Enrolled tuples that refer to it? (Cascade)
    - Disallow if referred to? (No Action)
    - Set sid in referring Enrolled tuples to a default value? (Set Default)
    - Set sid in referring Enrolled tuples to null, denoting `unknown' or `inapplicable'. (Set NULL)
- Similar issues arise if primary key of Students tuple is updated.

# Integrity Constraints (ICs)



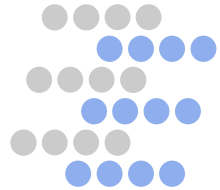
- Condition that must be true for any instance of the database
  - e.g., domain constraints
  - ICs are specified when schema is defined
  - ICs are checked when relations are modified
- A legal instance of a relation is one that satisfies all specified ICs
  - DBMS should not allow illegal instances
- If the DBMS checks ICs, stored data is more faithful to real-world meaning
  - Avoids data entry errors, too!

# Where do ICs Come From?



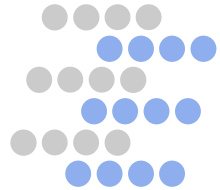
- Semantics of the real world!
- Note:
  - We can check IC violation in a DB instance
  - We can NEVER infer that an IC is true by looking at an instance
    - An IC is a statement about all possible instances!
  - From example, we know name is not a key, but the assertion that sid is a key is given to us
- Key and foreign key ICs are the most common
- More general ICs supported too

# Relational Query Languages



- Feature: Simple, powerful *ad hoc querying*
- Declarative languages
  - Queries precisely specify *what* to return
  - DBMS is responsible for efficient evaluation (*how*).
  - Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change
    - Key to data independence!

# The SQL Query Language



- The most widely used relational query language.
  - Current std is SQL:2008; SQL92 is a basic subset
- To find all 18 year old students, we can write:

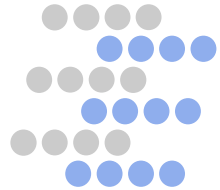
```
SELECT *
FROM Students S
WHERE S.age=18
```

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

- To find just names and logins, replace the first line:

```
SELECT S.name, S.login
```

# Querying Multiple Relations



- What does the following query compute?

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='A'
```

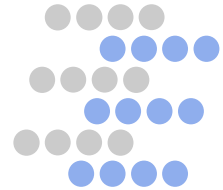
Given the following instance of Enrolled

| sid | cid | grade |
|---|---|---|
| 53831 | Carnatic101 | C |
| 53831 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |

we get:

| S.name | E.cid |
|---|---|
| Smith | Topology112 |

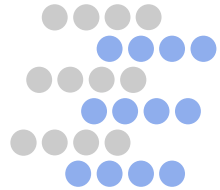
# Semantics of a Query



- A **conceptual evaluation method** for the previous query:
  - 1. do FROM clause: compute cross-product of Students and Enrolled
  - 2. do WHERE clause: Check conditions, discard tuples that fail
  - 3. do SELECT clause: Delete unwanted fields

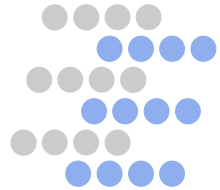
- Remember, this is conceptual. Actual evaluation will be much more efficient, but must produce the same answers.

# Cross-product of Students and Enrolled Instances

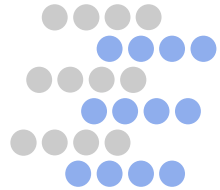


| S.sid | S.name | S.login | S.age | S.gpa | E.sid | E.cid | E.grade |
|---|---|---|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 | 53831 | Carnatic101 | C |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53831 | Reggae203 | B |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53650 | Topology112 | A |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53666 | History105 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Carnatic101 | C |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Reggae203 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53650 | Topology112 | A |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53666 | History105 | B |
| 53650 | Smith | smith@math | 19 | 3.8 | 53831 | Carnatic101 | C |
| 53650 | Smith | smith@math | 19 | 3.8 | 53831 | Reggae203 | B |
| **53650** | Smith | smith@math | 19 | 3.8 | **53650** | Topology112 | **A** |
| 53650 | Smith | smith@math | 19 | 3.8 | 53666 | History105 | B |

# Relational Model: Summary

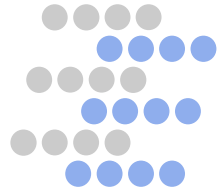


- A tabular representation of data.
- Simple and intuitive, currently the most widely used
  - Object-relational features in most products
  - XML support added in SQL:2003, most systems
- Integrity constraints
  - Specified by the DBA to capture application semantics.
  - DBMS prevents violations.
  - Some important ICs:
    - primary and foreign keys
    - Domain constraints
- Powerful query languages exist.
  - SQL is the standard commercial one
    - DDL - Data Definition Language
    - DML - Data Manipulation Language



# STORING DATA: DISKS AND FILES

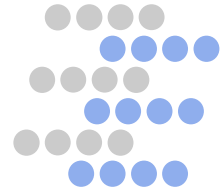
# Disks and Files



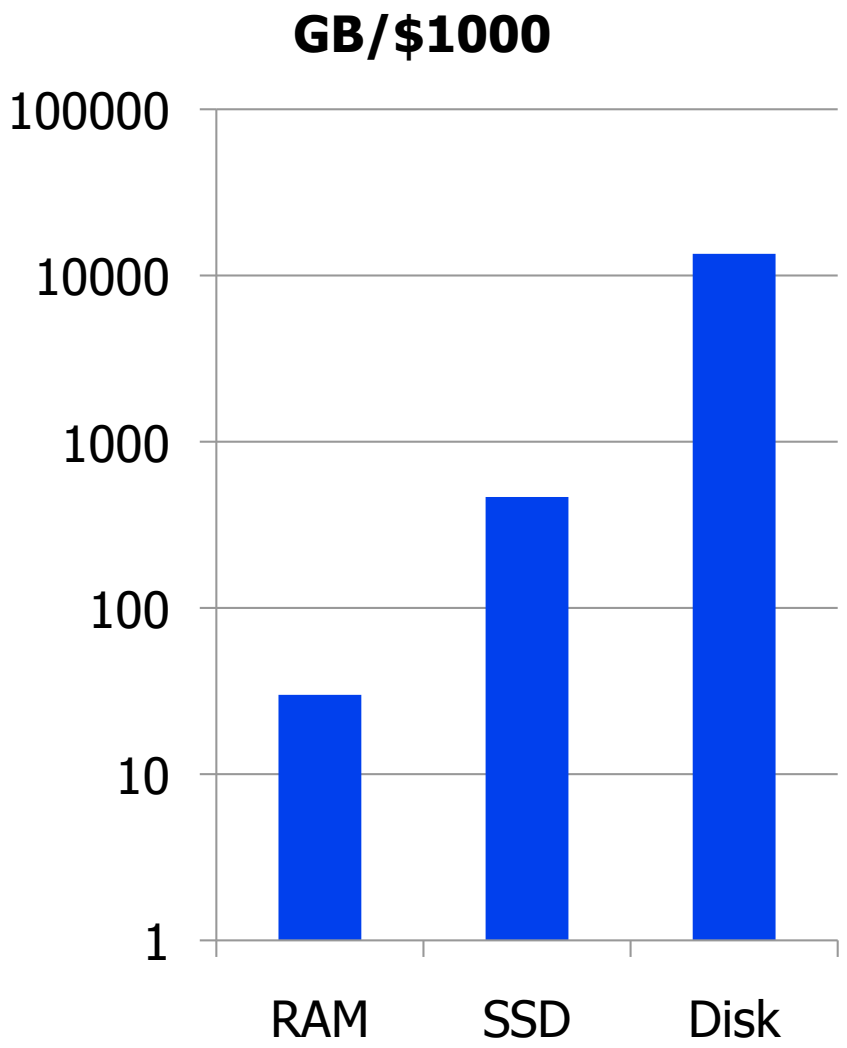
- DBMS stores information on disks
  - Disks are a mechanical anachronism!

- Major implications for DBMS design!
  - READ: transfer data from disk to RAM
  - WRITE: transfer data from RAM to disk
  - Both high-cost relative to memory references
    - Can/should plan carefully!

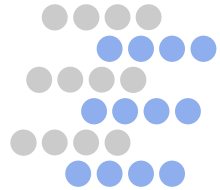
# Why Not Store Everything in Main Memory?



- Costs a lot. For $1000:
  - ~30GB of RAM
  - ~465GB of Solid State Disk
  - ~13.5TB of Magnetic Disk
- Main memory is volatile
  - Want data to persist between runs (Obviously!)
- That said:
  http://www.violin-memory.com/



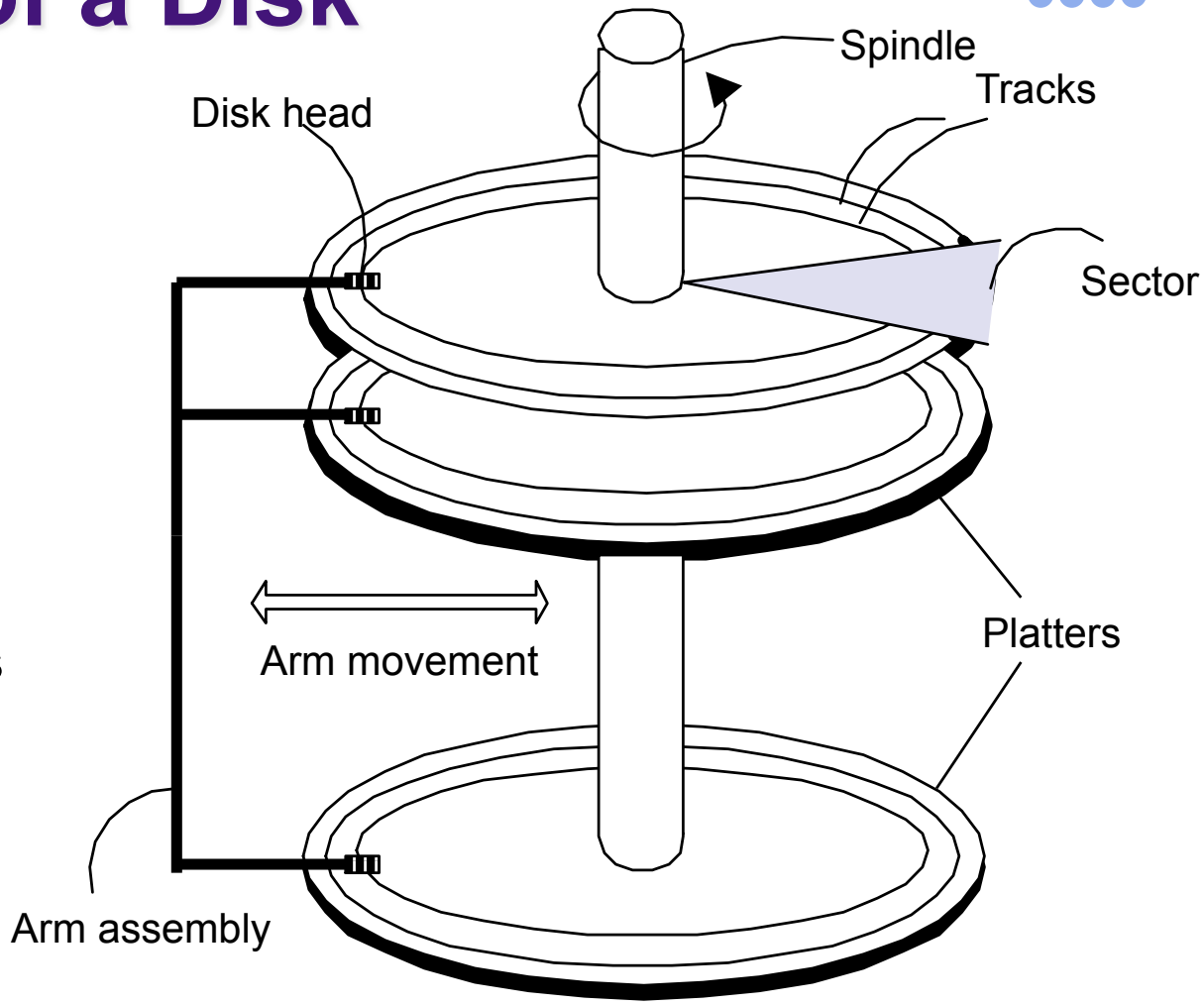
# Disks



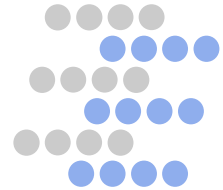
- Still the storage device of choice
- Main advantage over tape (archival):
  - random access vs. sequential
- Fixed unit of transfer
  - Read/write disk blocks or pages (8K)
- Not “random access” (vs. RAM)
  - Time to retrieve a block depends on location
  - Relative placement of blocks on disk has major impact on DBMS performance!

# Components of a Disk

- Platters spin (say 120 rps)
- Arm assembly moved in or out to position a head on a desired track.
  - Tracks under heads make a *cylinder* (imaginary)
- Only one head reads/writes at any one time
- *Block size* is a multiple of (fixed) *sector size*

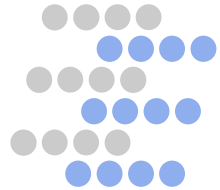


# Accessing a Disk Page



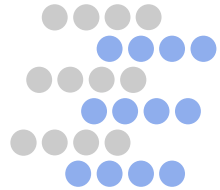
- Time to access (read/write) a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from 0 to 10msec
  - Rotational delay varies from 0 to 3msec
  - Transfer rate around .02msec per 8K block
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?

# Notes on Flash



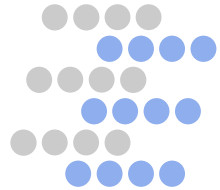
- Various technologies, we focus on NAND
  - suited for volume data storage
  - alternative: NOR Flash
- Read is random access and fast
  - E.g. 512B at a time
- Write is coarser grained and slower
  - E.g. 16KB at a time. Can slow down over time
- Some concern about write endurance
  - 100K cycle lifetimes?
- Still changing quickly

# Storage Pragmatics & Trends



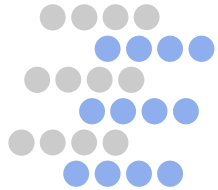
- Many significant DBs are not that big, e.g.
  - Daily weather, round the globe, 1929-2009: 20GB
  - 2000 US Census: 200GB
  - All of Wikipedia: 500GB
- Meanwhile, data sizes are growing faster than Moore's Law
- What is the role of disk, flash, RAM?
  - The subject of much debate/concern!

# Bottom Line (for now!)



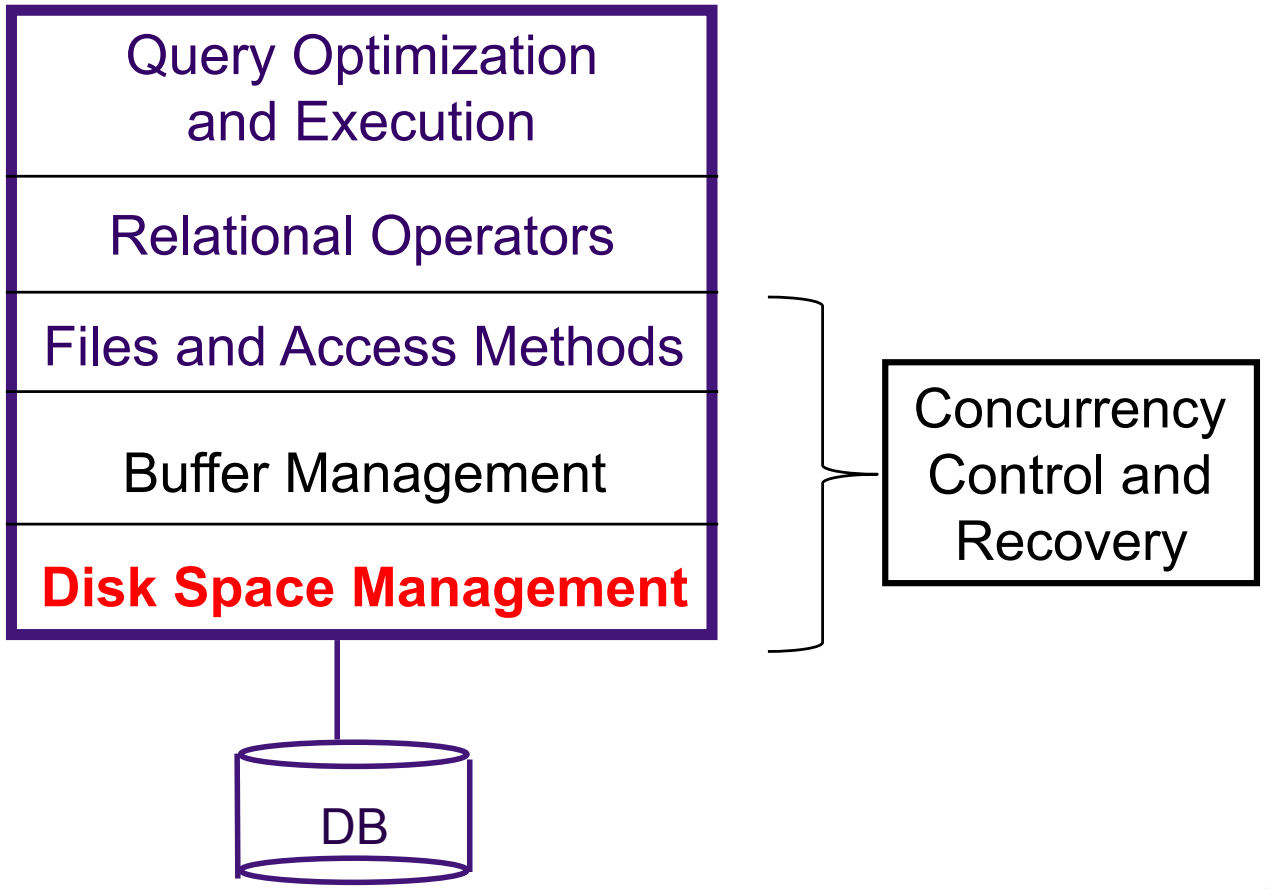
- Really big DBs look relatively traditional
  - Disk is still the best cost/MB by orders of magnitude
  - Flash role: a limited size but “random access” disk for special use
- Smaller DB story is changing quickly
  - Entry cost for disk is not cheap, so flash wins at the low end
  - Many interesting databases fit in RAM
- Lots of uncertainty on the HW side
  - If disk/flash/RAM improve at the same rates, status quo is fine
  - But more uncertainty today on all 3 than at any time in the past!
- Lots of uncertainty on the SW/usage side
  - Everybody can generate and archive data cheaply and easily
    - So the high end is ubiquitous?
  - Lots of incredibly rich data sets are pretty small
    - So the low end is ubiquitous?

# Arranging Pages on Disk

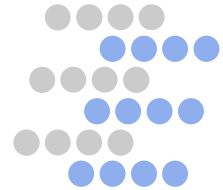


- `*Next*' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Arrange file blocks sequentially on disk
  - minimize seek and rotational delay.
- For a sequential scan, *pre-fetch*
  - several pages at a time!

# Block diagram of a DBMS

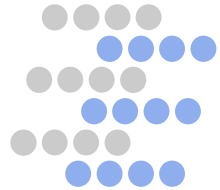


# Disk Space Management



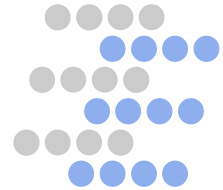
- Lowest layer of DBMS, manages space on disk
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a *sequence* of pages best satisfied by pages stored sequentially on disk!
  - Responsibility of disk space manager.
  - Higher levels don't know how this is done, or how free space is managed.
  - Though they may make performance assumptions!
    - Hence disk space manager should do a decent job.

# Files of Records



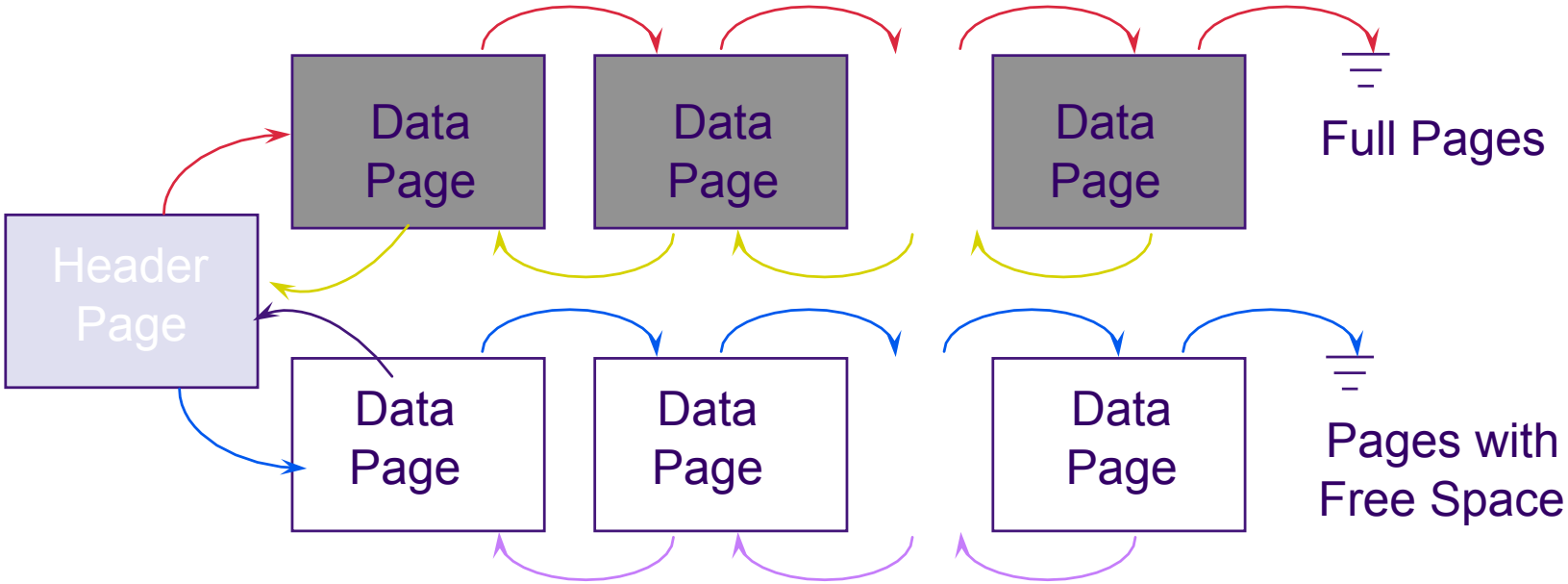
- Blocks are the interface for I/O, but…
- Higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - fetch a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)
- Typically implemented as multiple OS “files”
  - Or “raw” disk space

# Unordered (Heap) Files



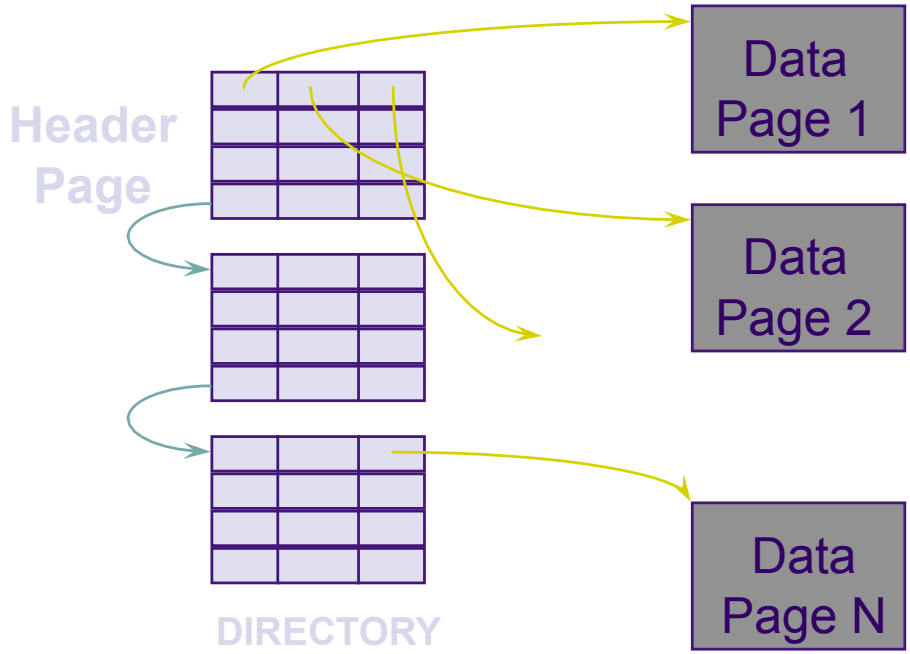
- Collection of records in no particular order.
- As file shrinks/grows, disk pages (de)allocated
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this.
  - We'll consider 2

# Heap File Implemented as a List



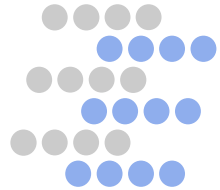

- The header page id and Heap file name must be stored someplace.
  - Database “catalog”
- Each page contains 2 `pointers’ plus data.

# Better: Use a Page Directory



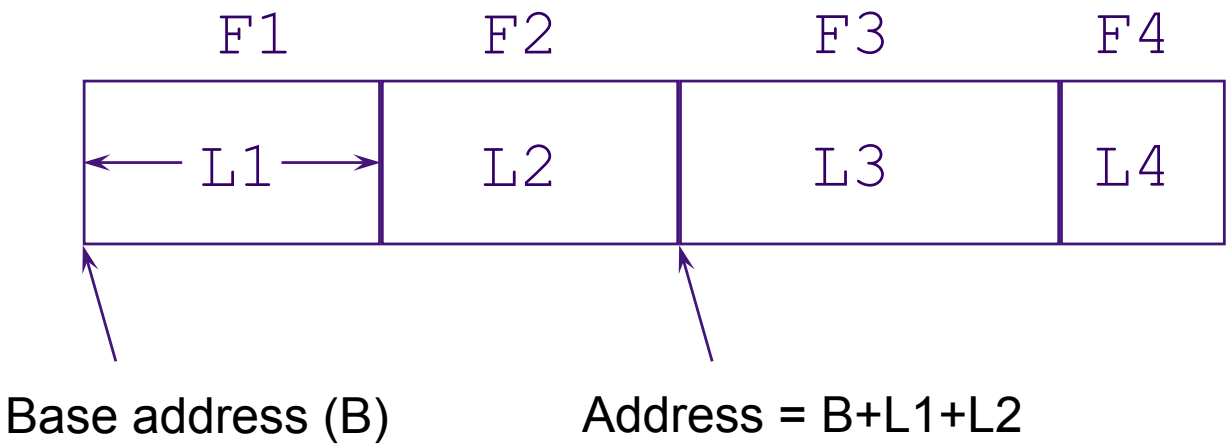

- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.
  - *Much smaller than linked list of all HF pages*!

# Indexes (sneak preview)



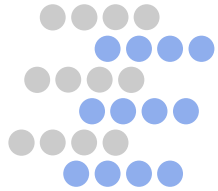
- A Heap file allows us to retrieve records:
  - by specifying the *rid*, or
  - by scanning all records sequentially
- Nice to fetch records *by value*, e.g.,
  - Find all students in the “CS” department
  - Find all students with a gpa > 3 AND blue hair
- Indexes: file structures for efficient value-based queries

# Record Formats: Fixed Length

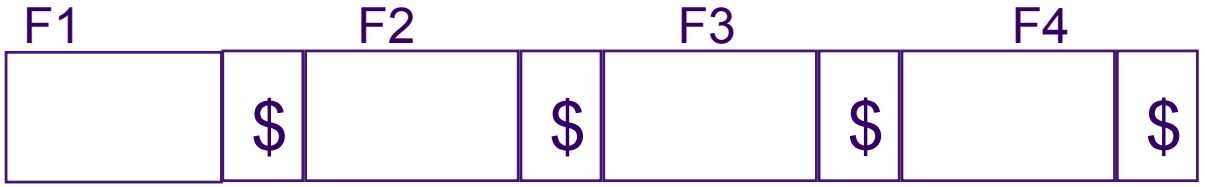



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field done via arithmetic.

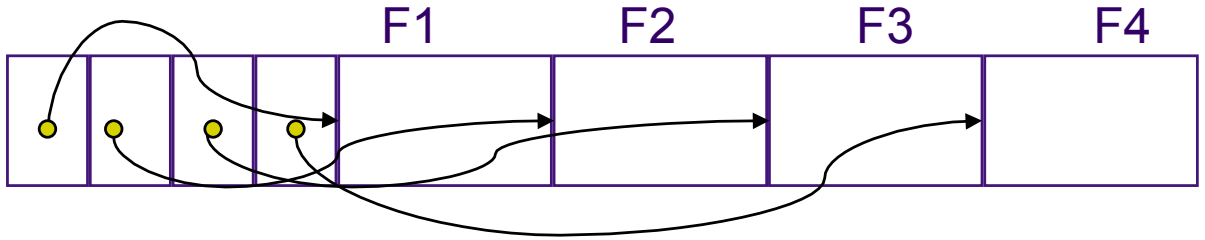
# Record Formats: Variable Length



- Two alternative formats (# fields is fixed):



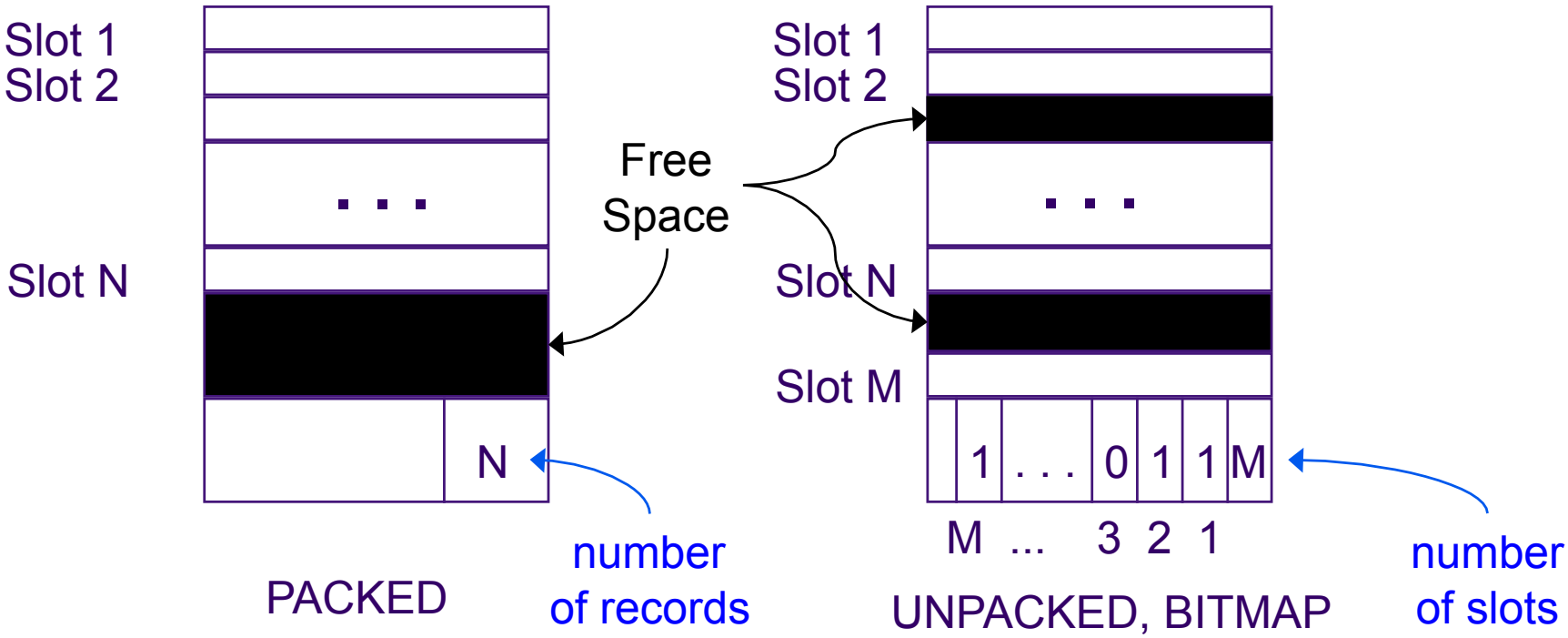

Fields Delimited by Special Symbols




Array of Field Offsets

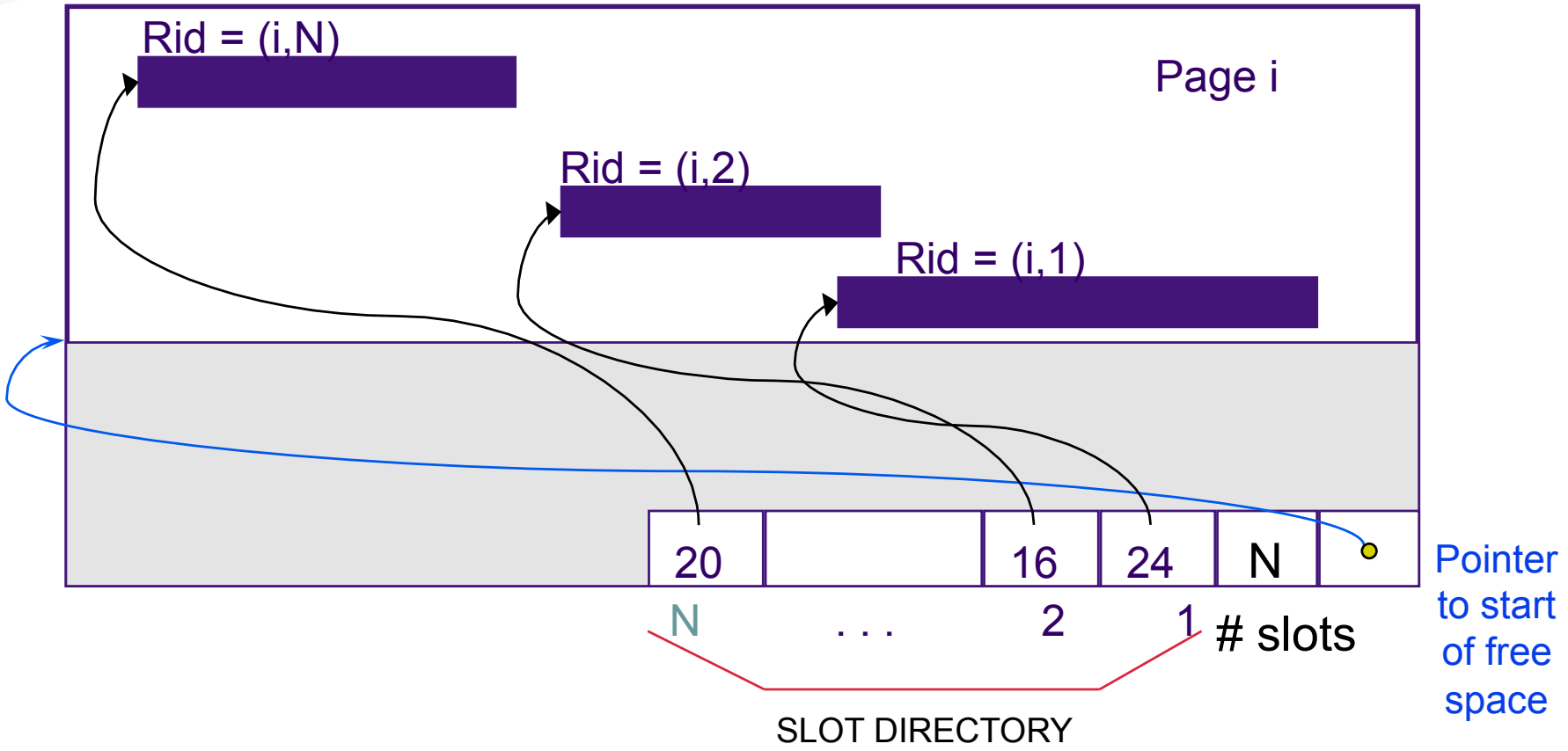
Second offers direct access to i'th field, efficient storage of *nulls* (special *don't know* value); small directory overhead.

# Page Formats: Fixed Length Records



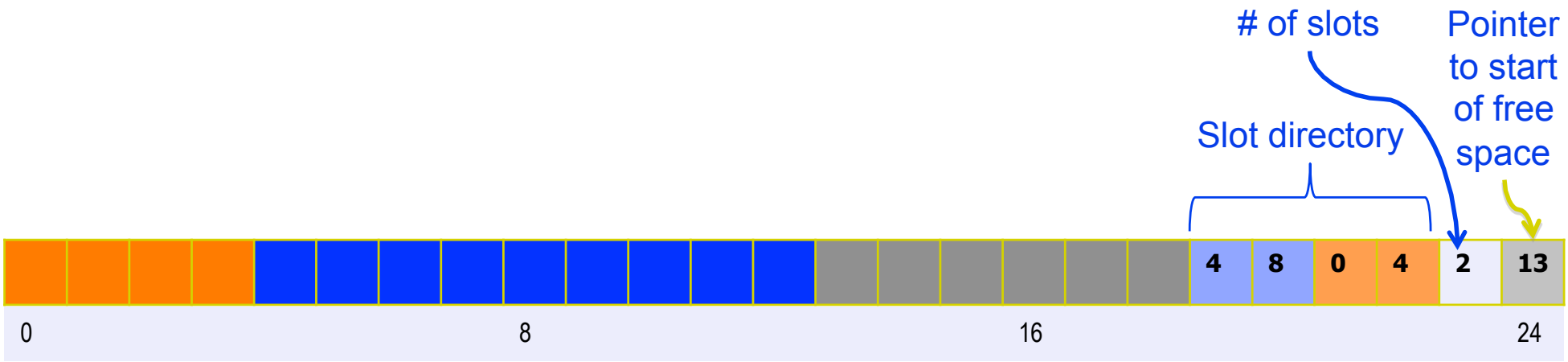

*Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

# Page Formats: Variable Length Records



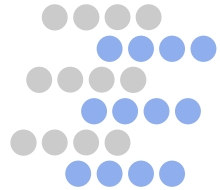

*Can move records on page without changing rid; so, attractive for fixed-length records too.*

# Slotted page: a detailed view



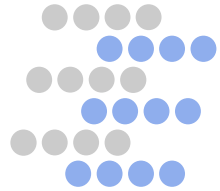

- What's the biggest tuple you can add?
  - Needs 2 bytes of slot space
  - *x* bytes of storage

# System Catalogs



- For each relation:
  - name, file location, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

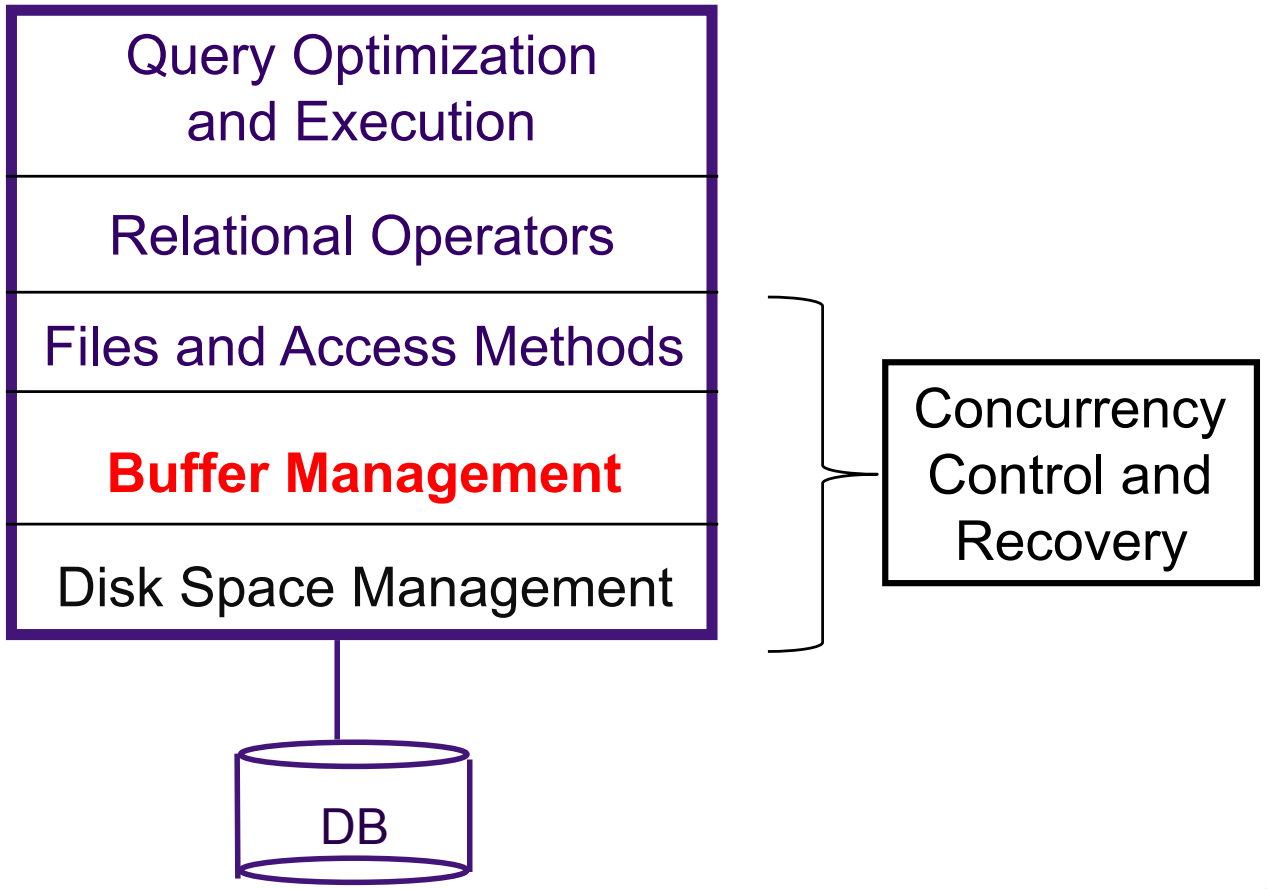
☛ *Catalogs are themselves stored as relations*!



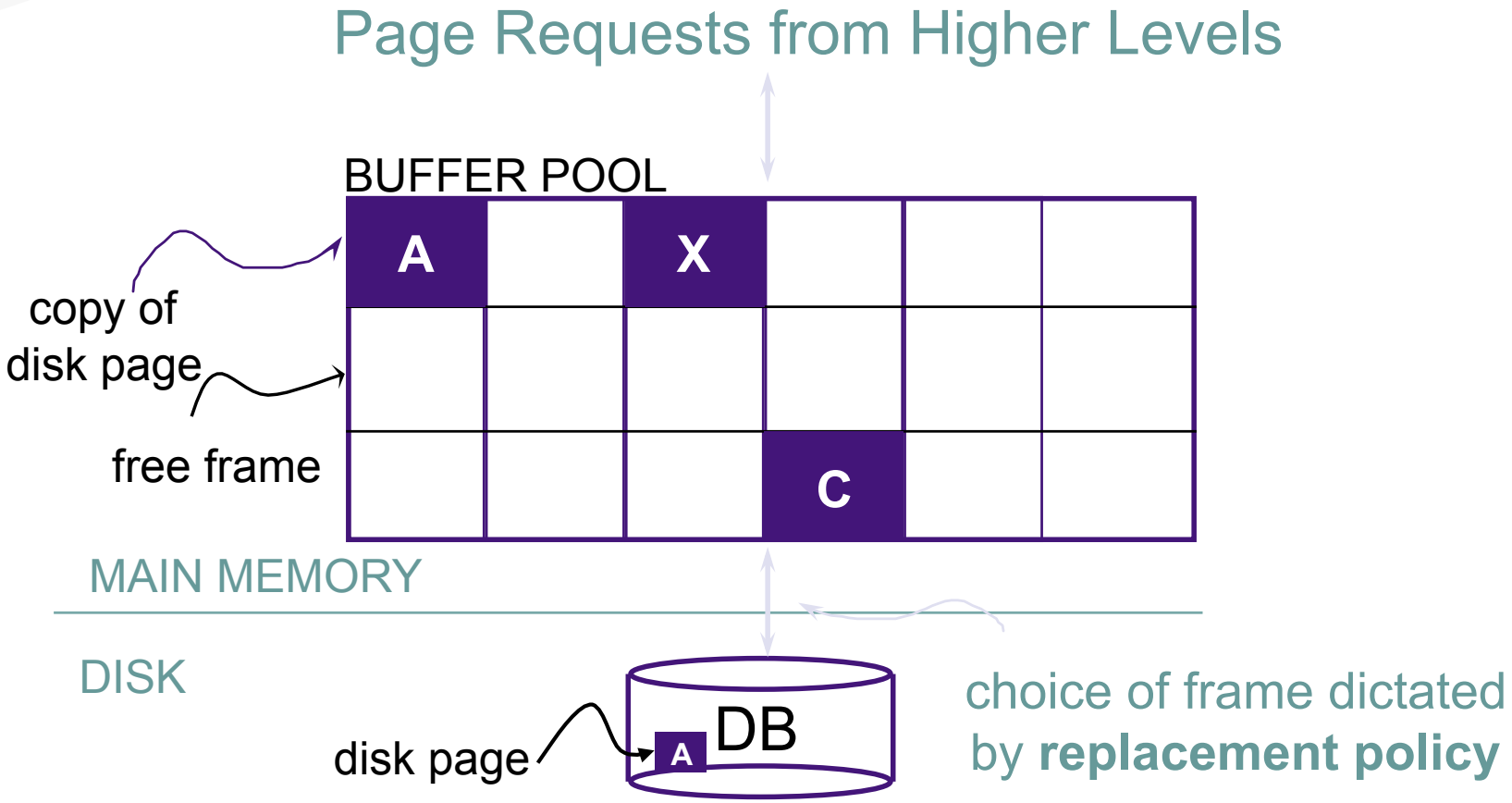
# Attr_Cat(attr_name, rel_name, type, position)

| attr_name | rel_name | type | position |
|---|---|---|---|
| attr_name | Attribute_Cat | string | 1 |
| rel_name | Attribute_Cat | string | 2 |
| type | Attribute_Cat | string | 3 |
| position | Attribute_Cat | integer | 4 |
| sid | Students | string | 1 |
| name | Students | string | 2 |
| login | Students | string | 3 |
| age | Students | integer | 4 |
| gpa | Students | real | 5 |
| fid | Faculty | string | 1 |
| fname | Faculty | string | 2 |
| sal | Faculty | real | 3 |

# Block diagram of a DBMS

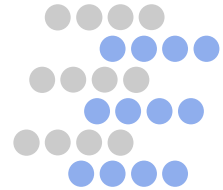


# Buffer Management in a DBMS




- *Data must be in RAM for DBMS to operate on it!*
- *BufMgr hides the fact that not all data is in RAM*

# When a Page is Requested ...

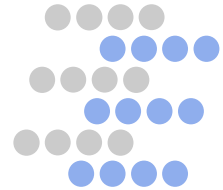


- Buffer pool information table contains:
  <frame#, pageid, pin_count, dirty>

1. If requested page is not in pool:
   a. Choose a frame for *replacement.*
      *Only "un-pinned" pages are candidates!*
   b. If frame "dirty", write current page to disk
   c. Read requested page into frame
2. *Pin* the page and return its address.

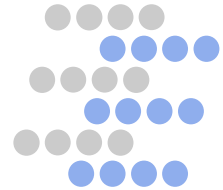
*If requests can be predicted (e.g., sequential scans)*
*pages can be pre-fetched several pages at a time!*

# More on Buffer Management



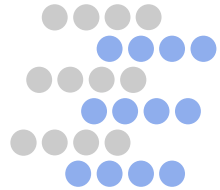
- Requestor of page must eventually:
  1. *unpin* it
  2. indicate whether page was modified via *dirty* bit.
- Page in pool may be requested many times,
  - a *pin count* is used.
  - To pin a page: pin_count++
  - A page is a candidate for replacement iff *pin count* == 0 (“unpinned”)
- CC & recovery may do additional I/Os upon replacement.
  - *Write-Ahead Log* protocol; more later!

# Buffer Replacement Policy



- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), MRU, Clock, …
- Policy can have big impact on #I/O's;
  - Depends on the *access pattern*.

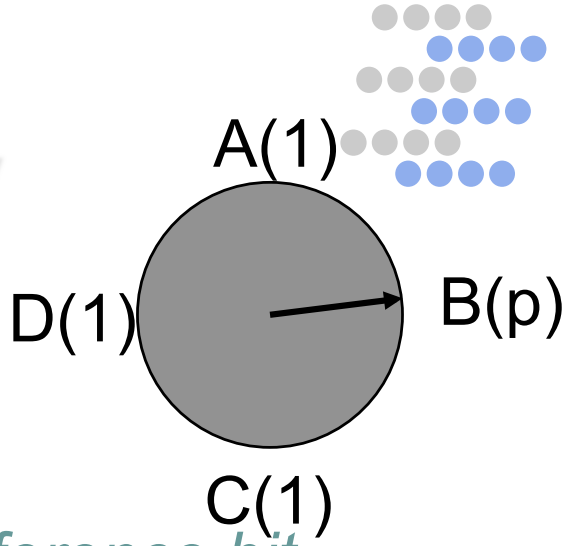
# LRU Replacement Policy



- *Least Recently Used (LRU)*
  - (Frame pinned: “in use”, not available to replace)
  - track time each frame last *unpinned* (end of use)
  - replace the frame which has the earliest unpinned time
- Very common policy: intuitive and simple
  - Works well for repeated accesses to popular pages

- *Problem: Sequential flooding*
  - LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O.
  - Idea: MRU better in this scenario?

# “Clock” Replacement Policy

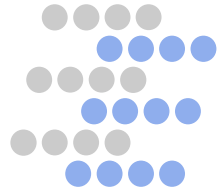



- An approximation of LRU
- Arrange frames into a cycle, store one *reference bit per frame*
  - Can think of this as the 2nd chance bit
- When pin count reduces to 0, turn on ref. bit

- When replacement necessary:

```
do for each frame in cycle {
        if (pincount == 0 && ref bit is on)
                turn off ref bit; // 2nd chance
        else if (pincount == 0 && ref bit is off)
                choose this page for replacement;
} until a page is chosen;
```

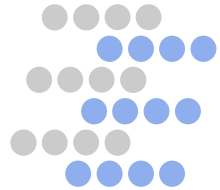
# DBMS vs. OS File System



OS does disk space & buffer mgmt: why not let OS manage these tasks?

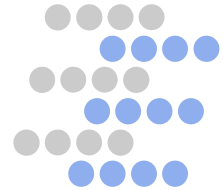
- Buffer management in DBMS requires ability to:
  - pin page in buffer pool, force page to disk, order writes
    - important for implementing CC & recovery
  - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.
- I/O typically done via lower-level OS interfaces
  - Avoid OS “file cache”
  - Control write timing, prefetching

# Summary



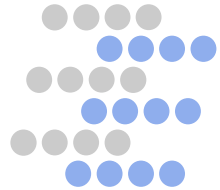
- Disks provide cheap, non-volatile storage.
  - Better random access than tape, worse than RAM
  - Arrange data to minimize *seek* and *rotation* delays.
    - Depends on workload!
- Buffer manager brings pages into RAM.
  - Page pinned in RAM until released by requestor.
  - Dirty pages written to disk when frame replaced (sometime after requestor unpins the page).
  - Choice of frame to replace based on *replacement policy.*
  - Tries to *pre-fetch* several pages at a time.

# Summary (Contd.)

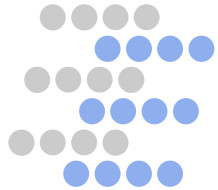


- DBMS vs. OS File Support
  - DBMS needs non-default features
  - Careful timing of writes, control over prefetch
- Variable length record format
  - Direct access to i’th field and null values.
- Slotted page format
  - Variable length records and intra-page reorg

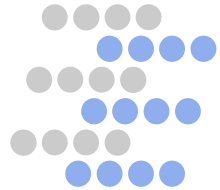
# Summary (Contd.)



- DBMS “File” tracks collection of pages, records within each.
  - Pages with free space identified using linked list or directory structure
- Indexes support efficient retrieval of records based on the values in some fields.
- Catalog relations store information about relations, indexes and views.

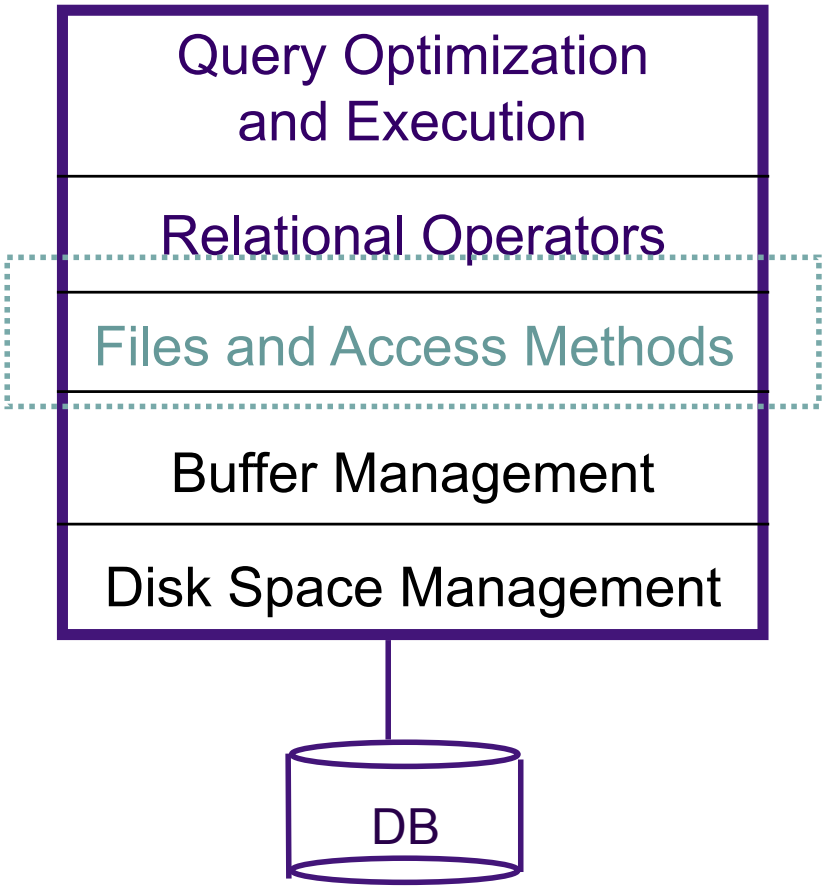


# TIME TO GET A BREAK

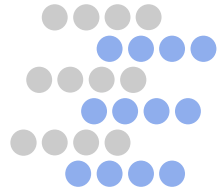


# FILE ORGANIZATIONS AND INDEXING

# Context


Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
DB

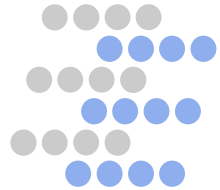
# Multiple File Organizations



Many alternatives exist, *each good for some situations, and not so good in others:*

- Heap files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best for retrieval in *search key* order, or only a `range' of records is needed.
- Clustered Files (with Indexes)

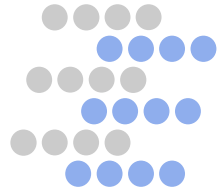
# Cost Model for Analysis



- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** (Average) time to read or write disk block
- *Average-case* analyses for *uniform random* workloads
- We will ignore:
  - Sequential vs. Random I/O
  - Pre-fetching
  - Any in-memory costs

☛ ***Good enough to show the overall trends!***

# More Assumptions



- Single record insert and delete.
- Equality selection - exactly one match
- For Heap Files:
  - Insert always appends to end of file.
- For Sorted Files:
  - Files compacted after deletions.
  - Selections on search key.

- Question all these assumptions and rework
  - As an exercise to study for tests, generate ideas

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File | Clustered File |
|---|---|---|---|
| **Scan all records** | | | |
| **Equality Search** | | | |
| **Range Search** | | | |
| **Insert** | | | |
| **Delete** | | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File | Clustered File |
|---|---|---|---|
| **Scan all records** | BD | BD | |
| **Equality Search** | | | |
| **Range Search** | | | |
| **Insert** | | | |
| **Delete** | | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File | Clustered File |
|---|---|---|---|
| **Scan all records** | BD | BD | |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ | |
| **Range Search** | | | |
| **Insert** | | | |
| **Delete** | | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

| | Heap File | Sorted File | Clustered File |
|---|---|---|---|
| **Scan all records** | BD | BD | |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ | |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ | |
| **Insert** | | | |
| **Delete** | | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

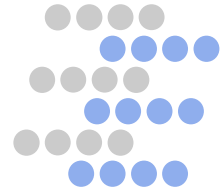
| | Heap File | Sorted File | Clustered File |
|---|---|---|---|
| **Scan all records** | BD | BD | |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ | |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ | |
| **Insert** | 2D | $((\log_2 B)+B)D$ | |
| **Delete** | | | |

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

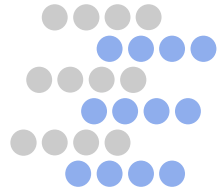
| | Heap File | Sorted File | Clustered File |
|---|---|---|---|
| **Scan all records** | BD | BD | |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ | |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ | |
| **Insert** | 2D | $((\log_2 B)+B)D$ | |
| **Delete** | 0.5BD + D | $((\log_2 B)+B)D$ | |

# Review: Files, Pages, Records



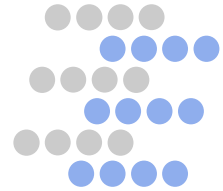
- Abstraction of stored data is “files” with “pages” of “records”
  - Records live on pages
  - Physical Record ID (RID) = <page#, slot#>
- Variable length data requires more sophisticated structures for records and pages (why?)
  - Fields in Records: offset array in header
  - Records on Pages: Slotted pages w/internal offsets & free space area
- Often best to be “lazy” about issues such as free space management, exact ordering, etc. (why?)
- Files can be unordered (heap), sorted, or kinda sorted (i.e., “clustered”) on a search key
  - Tradeoffs are update/maintenance cost vs. speed of accesses via the search key.
  - Files can be clustered (sorted) at most one way.
- Indexes can be used to speed up many kinds of accesses. (i.e., “access paths”)

# Indexes



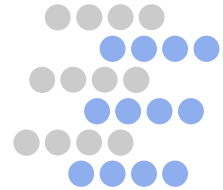
- Allow record retrieval *by value* in ≥1 field, e.g.,
  - Find all students in the “CS” department
  - Find all students with a gpa > 3
- *Index* : disk-based data structure for fast lookup by value
  - *Search key:* any subset of columns in the relation.
  - *Search key* may not be a *key* of the relation
    - Can have multiple items matching a lookup
- Index contains a collection of *data entries*
  - Items associated with each search key value ***k***
  - Data entries come in various forms, as we’ll see

# 1st Question to Ask About Indexes



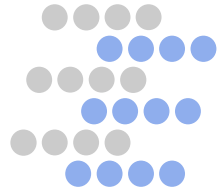
- What kinds of selections (lookups) do they support?
  - Selection: <key> <op> <constant>
  - Equality selections (op is =)?
  - Range selections (op is one of <, >, <=, >=, BETWEEN)?
  - More exotic selections?
    - 2-dimensional ranges (“east of Berkeley and west of Truckee and North of Fresno and South of Eureka”)
      - Or n-dimensional
    - 2-dimensional radii (“within 2 miles of Soda Hall”)
      - Or n-dimensional
    - Ranking queries (“10 restaurants closest to Berkeley”)
    - Regular expression matches, genome string matches, etc.
    - One common n-dimensional index: R-tree
- See http://gist.cs.berkeley.edu for research on this topic

# Single-level ordered indexes



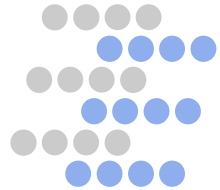
- Here we assume file records to be **physically ordered** on disks (no heap)
  - Records are ordered based on an ordering key field
- There are several types of ordered indexes:
  - Primary index
  - Clustering index
  - Secondary index
- Indexes can be
  - **Dense**: one entry for every record of the data file
  - **Sparse**: not all records are indexed

# Primary indexes (1)



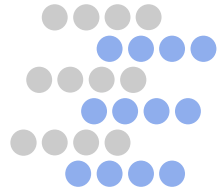
- Ordered file whose records are of fixed length with two fields <K(i),P(i)>
  - K(i) is of the same data type as the ordering key filed (the **primary key**) of the *data file*
    - Has a distinct value for each record
  - P(i) is a pointer to a **disk block**
- There is one index entry for each block in the data file
- Each index entry has the value of the primary key field for the first record in a block (called the **block anchor**)

# Primary indexes (2)



- The total number of entries in the index is the same as the number of disk blocks in the ordered data file
- Primary indexes are **sparse**
- Major problem with primary indexes is insertion and deletion of records
  - If we insert a record in the correct position of the data file (ordered) we have to move records and index entries

# Primary indexes: example (1)



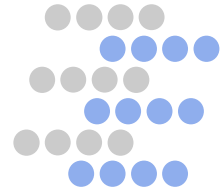
- Ordered file with r=30,000 records
- Block size B=1024 bytes
- Fixed records of length R=100 bytes

➔ Blocking factor = 1024/100 ~ 10 records per block

➔ Total # of blocks = 30,000/10 = 3000 blocks

➔ Binary search on data file requires log2(3000) ~ 12 block accesses

# Primary indexes: example (2)



- Suppose K(i) is V=9 bytes and P(i) is P=6 bytes long
- The size of each primary index entry is 9+6 = 15 bytes

➔ The blocking factor is 1024/15 ~ 68 entries

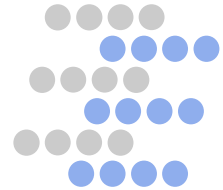
➔ Total number of entries = number of blocks = 3000

➔ The number of index blocks is 3000/68 = 45 blocks

➔ A binary search on the index is log2(45) ~ 6 block accesses

➔ Need an additional block access to the data file for a total of 7 block accesses

# Clustering indexes (1)



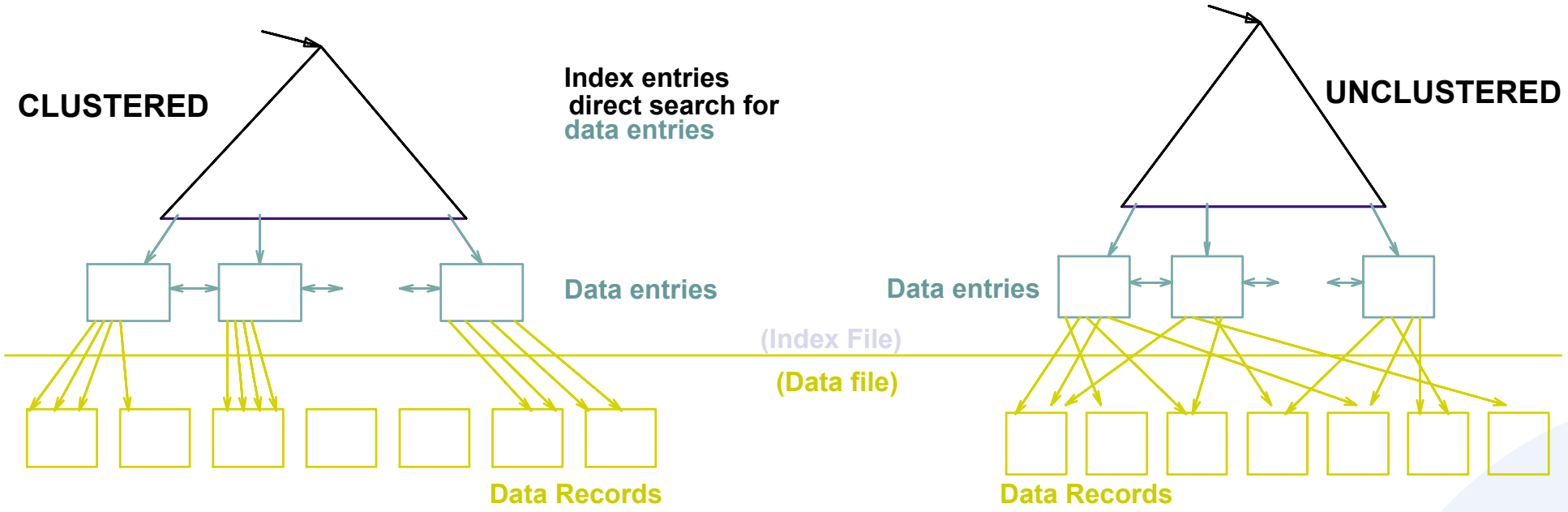
- When records are physically ordered on a **nonkey field** (no distinct value for each record), the file is called clustered

- Ordered file with two fields
  - First field: same data type of the clustering field
  - Second field: block pointer

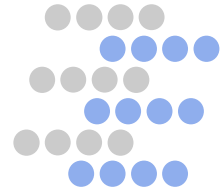
- One value for each distinct clustering field and a pointer to the first block in the data file that has a record with that value for the clustering field

# Clustered vs. Unclustered Index

- Data records in a Heap file
  - To build clustered index, first sort the Heap file
    - with some free space on each block for future inserts
  - Overflow blocks may be needed for inserts
    - Thus, order of data recs is `close to', but not identical to, the sort order.



# Unclustered vs. Clustered Indexes



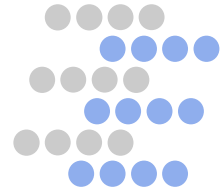
- Clustered Pros
  - Efficient for range searches
  - Supports some types of compression
  - Possible locality benefits
    - Disk scheduling, prefetching, etc.
- Clustered Cons
  - More expensive to maintain
    - on the fly or “sloppily” via reorgs
    - Heap file usually only packed to 2/3 to accommodate inserts

# Cost of Operations

**B:** The number of data pages
**R:** Number of records per page
**D:** (Average) time to read or write disk page

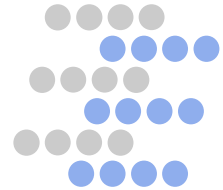
| | Heap File | Sorted File | Clustered File |
|---|---|---|---|
| **Scan all records** | BD | BD | 1.5 BD |
| **Equality Search** | 0.5 BD | $(\log_2 B) * D$ | $(\log_F 1.5B+1) * D$ |
| **Range Search** | BD | $[(\log_2 B) +$ #match pg$]*D$ | $[(\log_F 1.5B) +$ *#match pg*$]*D$ |
| **Insert** | 2D | $((\log_2 B)+B)D$ | $((\log_F 1.5B)+2) * D$ |
| **Delete** | 0.5BD + D | $((\log_2 B)+B)D$ *(because R,W 0.5)* | $((\log_F 1.5B)+2) * D$ |

# Secondary indexes (1)



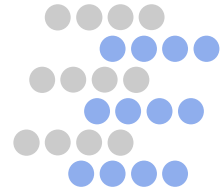
- Provide a secondary means of accessing a file for which some primary access already exists
- The secondary index may be on a field which is a candidate key or a nonkey
- Ordered file with two fields <K(i),P(i)>
  - K(i) is of the same data type as the non-ordering key filed of the *data file*
    - If the key has a distinct value for each record and is sorted, it's called a secondary key
  - P(i) is a pointer to a **disk block** or a **record**

# Secondary indexes (2)



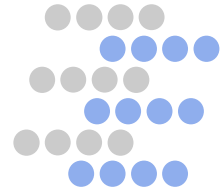
- Let's consider a secondary index access structure on a key field that has a distinct value for every record
- There will be one entry in the index for every record
- Entries are ordered by value of K(i) so we can perform a binary search
- Since records of the data file are not physically ordered by values of the secondary key field we cannot use block anchors
- The index is **dense**

# Secondary indexes: example (1)



- Continue from the example before: r=30,000 fixed-length records of size R=100 bytes stored on disk with block size B=1024 bytes
- Hence, the file has b=3000 blocks
- To do a linear search we require 3000/2 = 1500 block access on average
  - Linear search since the file is not ordered by the key we are considering

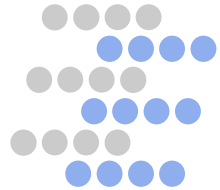
# Secondary indexes: example (2)



- By using a secondary index on a non-ordering key filed of the file V=9 bytes long with a block pointer P=6 bytes long ➔ each entry is 15 bytes and the blocking factor is 1024/15 ~ 68 entries per block
- In a dense secondary index such as this, the total number of entries is equal to the number of records, i.e., 30,000
- The number of blocks required for the index is 30,000/68 ~ 442 blocks
  - A binary search on the index costs log2(442) ~ 9 blocks accesses + 1 block access ➔ 10 block access

➔ Vast improvement over linear search!

# Secondary indexes



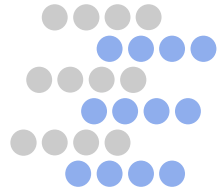
- We can also create a secondary index for a non-key field of a file

1. Include several index entries with the same K(i) value, one for each record

2. Have a variable-length record for the index entries, with a repeating field for the pointer.
   Keep a list of pointers for K(i).

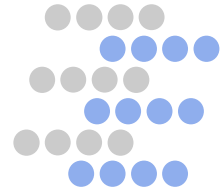
3. Keep the index entries of fixed length and have a single entry for each index field value. Create an extra level of indirection to handle multiple pointers
   - P(i) points to a block of record pointers
   - Each record pointer in that block points to one of the data file records with value K(i)
   - Cluster records if a single K(i) occurs in too many records

# Index Breakdown



- What selections does the index support
- Representation of data entries in index
  - i.e., what kind of info is the index actually storing?
  - 3 alternatives here
- Clustered vs. Unclustered Indexes
- Single Key vs. Composite Indexes
- Tree-based, hash-based, other

# Alternatives for Data Entry k* in Index



- Three alternatives:
  1. Actual data record (with key value **k**)
  2. <**k**, rid of matching data record>
  3. <**k**, list of rids of matching data records>

- Choice is orthogonal to the indexing technique.
  - B+ trees, hash-based structures, R trees, GiSTs, …

- Can have multiple (different) indexes per file.
  - E.g. file sorted by *age*, with a hash index on *salary* and a B+tree index on *name*.

# Alternatives for Data Entries (Contd.)

- Alternative 1:
  Actual data record (with key value ***k***)

  - Index as a file organization for records
    - Alongside Heap files or sorted files
  - At most one Alt. 1 index per relation
  - No “pointer lookups” to get data records

# Alternatives for Data Entries (Contd.)

Alternative 2

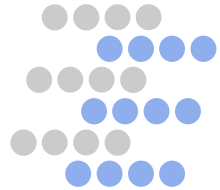
<**k**, rid of matching data record>

and Alternative 3

<**k**, list of rids of matching data records>

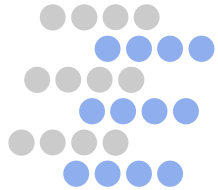
- Must use Alts. 2 or 3 to support >1 index per relation.
- Alt. 3 more compact than Alt. 2, but *variable sized data entries*
  - even if search keys are of fixed length.
- For large rid lists, data entry spans multiple blocks!

# Index Classification



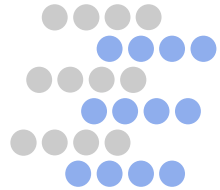
- *Clustered* vs. *Unclustered:*
- Clustered index:
  - order of data records the same as, or `close to', order of index data entries
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
  - Alternative 1 implies clustered, *but not vice-versa*.
- Note: another definition of "clustering"
  - Data mining, AI, stat

# Summary



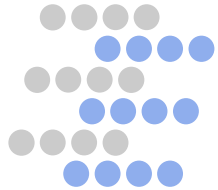
- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure.
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; log F N cost.
  - High fanout (F) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.
  - Typically, 67% occupancy on average.
  - Usually preferable to ISAM; adjusts to growth gracefully.
  - If data entries are data records, splits can change rids!

# Summary (Contd.)



- Key compression increases fanout, reduces height.
- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- B+ tree widely used because of its versatility.
  - One of the most optimized components of a DBMS.

# Multilevel indexes



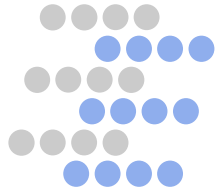
- In summary: we've seen already indexes, but the goal now is to optimize search speed for very large data files

- Binary search shrinks the search space by a factor of 2
  - Can we do better?
  - Yes, shrink the search space by a larger factor

➔ Use trees, which reduces the search space by their fan-out

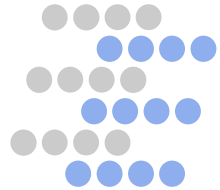
➔ Search takes approximately $\log_{fo}(b)$

# Tree-Structured Indexes



- Tree-structured indexing techniques support both range selections and equality selections
  - Selections of form field <op> constant
  - Equality selections (op is =)
    - Either “tree” or “hash” indexes help here
  - Range selections (op is one of <, >, <=, >=, BETWEEN)
    - “Hash” indexes don’t work for these.
  - More complex selections (e.g. spatial containment)
    - There are fancier trees that can do this…
- ISAM: static structure; early index technology
  - ISAM =Indexed Sequential Access Method
- B+ tree: dynamic, adjusts gracefully under inserts and deletes.

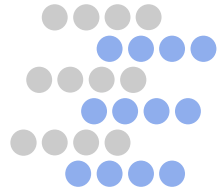
# A Note of Caution



- ISAM is an old-fashioned idea
  - B+-trees are usually better (not always)
  - Simpler than B+-tree, many of the same ideas

- Upshot
  - Don’t brag about ISAM on your resume
  - Do understand ISAM, and tradeoffs with B+-trees

# B+ Tree: The Most Widely Used

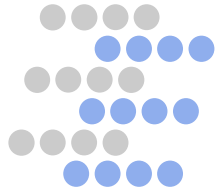


- Insert/delete function of F, N
  - F = fan-out, N = # leaf pages
  - Keep tree height-balanced

- Tree is not width-balanced: different nodes have different fan-outs
  - BUT: minimum 50% node occupancy (except for root)
  - Each node contains m entries where d <= m <= 2d entries
  - “d” is called the order of the tree

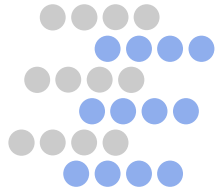
- Supports equality and range-searches efficiently
- As in ISAM, all searches go from root to leaves, but structure is dynamic

# B+ Trees in Practice



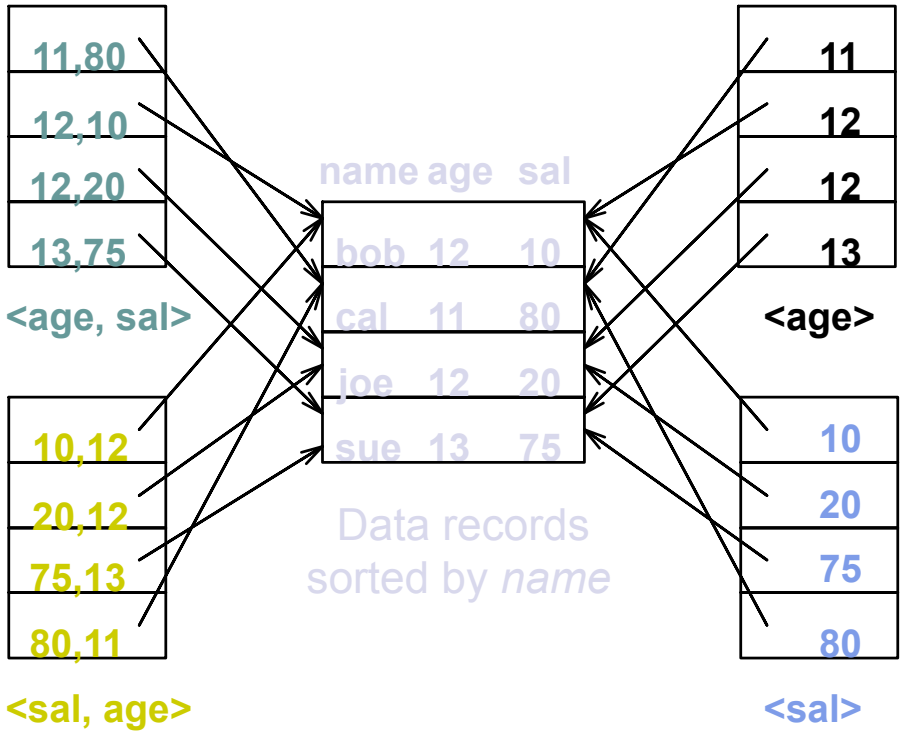
- Typical order: 100.  Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 2: $133^3$ =  2,352,637 entries
  - Height 3: $133^4$ = 312,900,700 entries
- Can often hold top levels in buffer pool:
  - Level 1 =  1 page  =  8 Kbytes
  - Level 2 =  133 pages =  1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

# Composite Search Keys

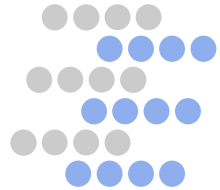


- Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <age,sal> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age > 20; or age=20 and sal > 10
- Data entries in index can be sorted by search key to support range queries.
  - Lexicographic order
  - Like the dictionary, but on fields, not letters!

Examples of composite key indexes using lexicographic order.

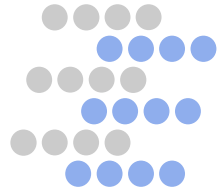


# Summary (1)



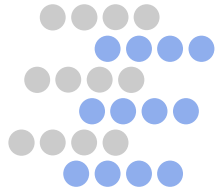
- File Layer manages access to records in pages
  - Record and page formats depend on fixed vs. variable-length
  - Free space management an **important issue**
  - Slotted page format supports variable length records and allows records to move on page
- Many alternative file organizations exist, each appropriate in some situation
- If selection queries are frequent, sorting the file or building an index is important
  - Hash-based indexes only good for equality search
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values

# Summary (2)

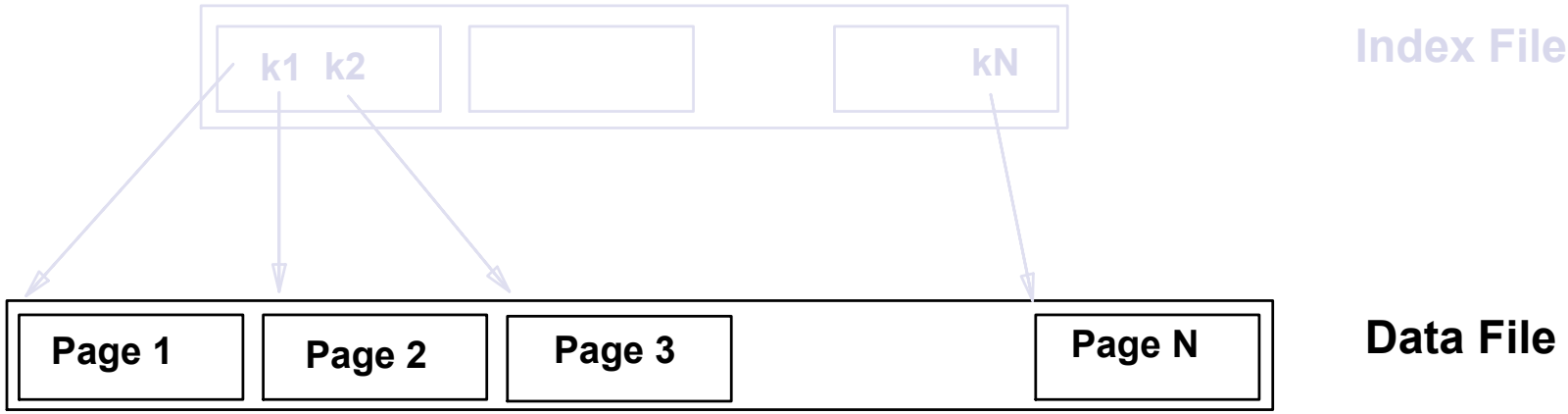


- Data entries in index can be1 of 3 alternatives: (1) actual data records, (2) <key, rid> pairs, or (3) <key, rid-list> pairs
  - Choice orthogonal to indexing structure (i.e., tree, hash, etc.)
- Usually have several indexes on a given file of data records, each with a different search key
- Indexes can be classified as clustered vs. un-clustered
- Differences have important consequences for utility/ performance

# Range Searches

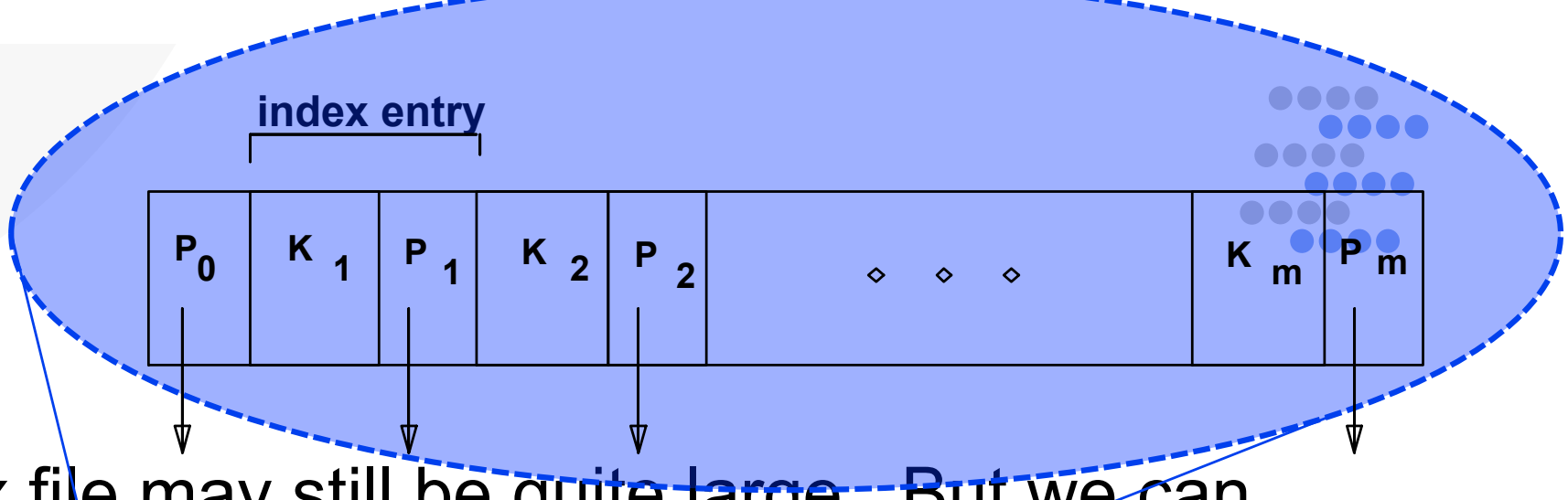


- ``*Find all students with gpa > 3.0*''
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search in a database can be quite high. Q: Why???
- Simple idea:  Create an `index' file.

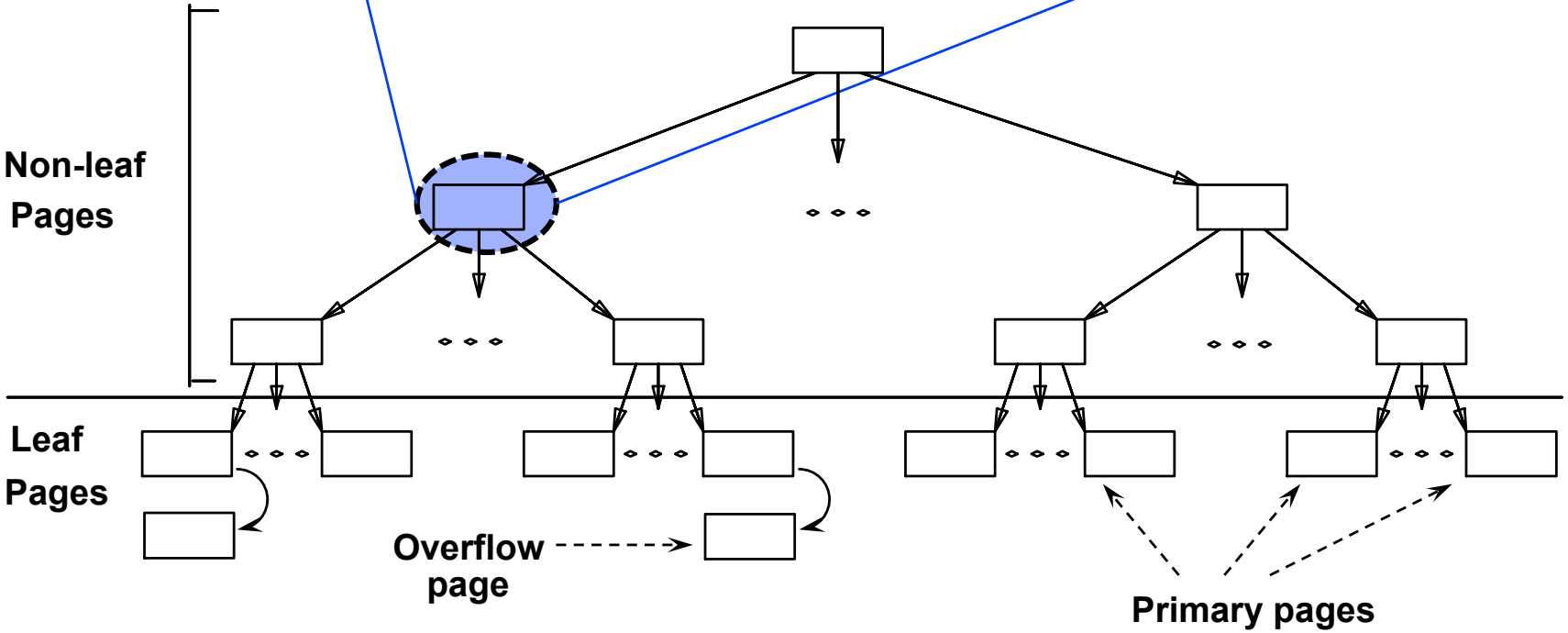



☛ *Can do binary search on (smaller) index file!*

# ISAM



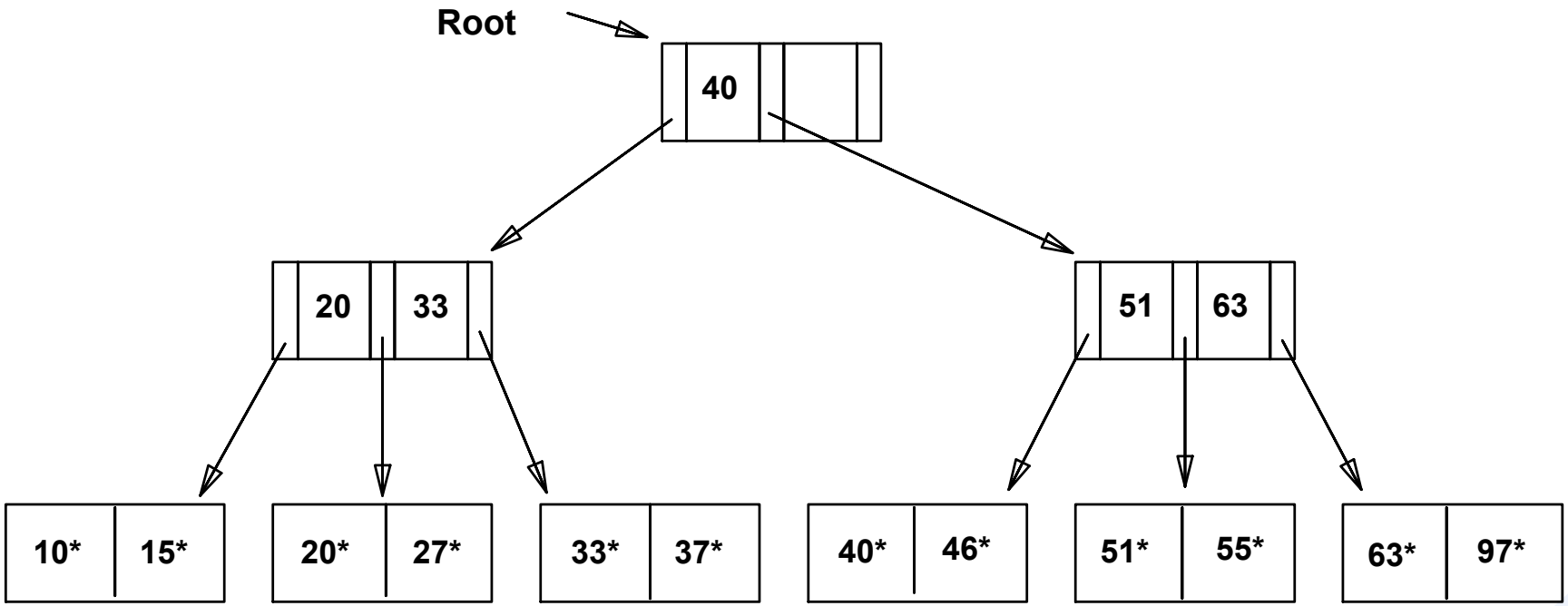

- Index file may still be quite large. But we can apply the idea repeatedly!



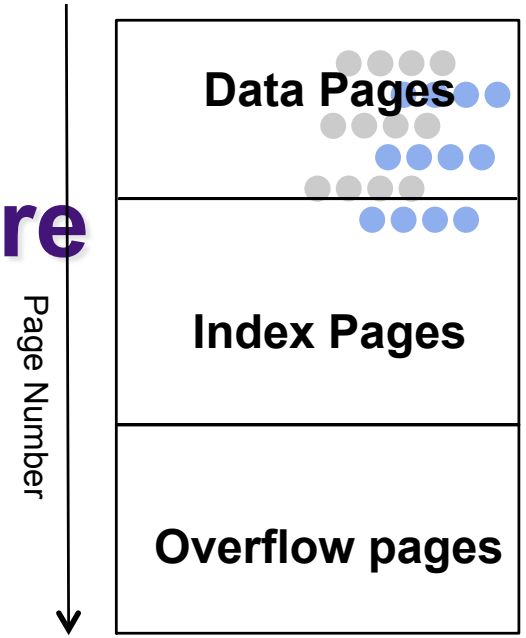

☛ *Leaf pages contain data entries.*

# Example ISAM Tree

- *Index entries*:<search key value, page id> they direct search for data entries *in leaves.*
- Example where each node can hold 2 entries;



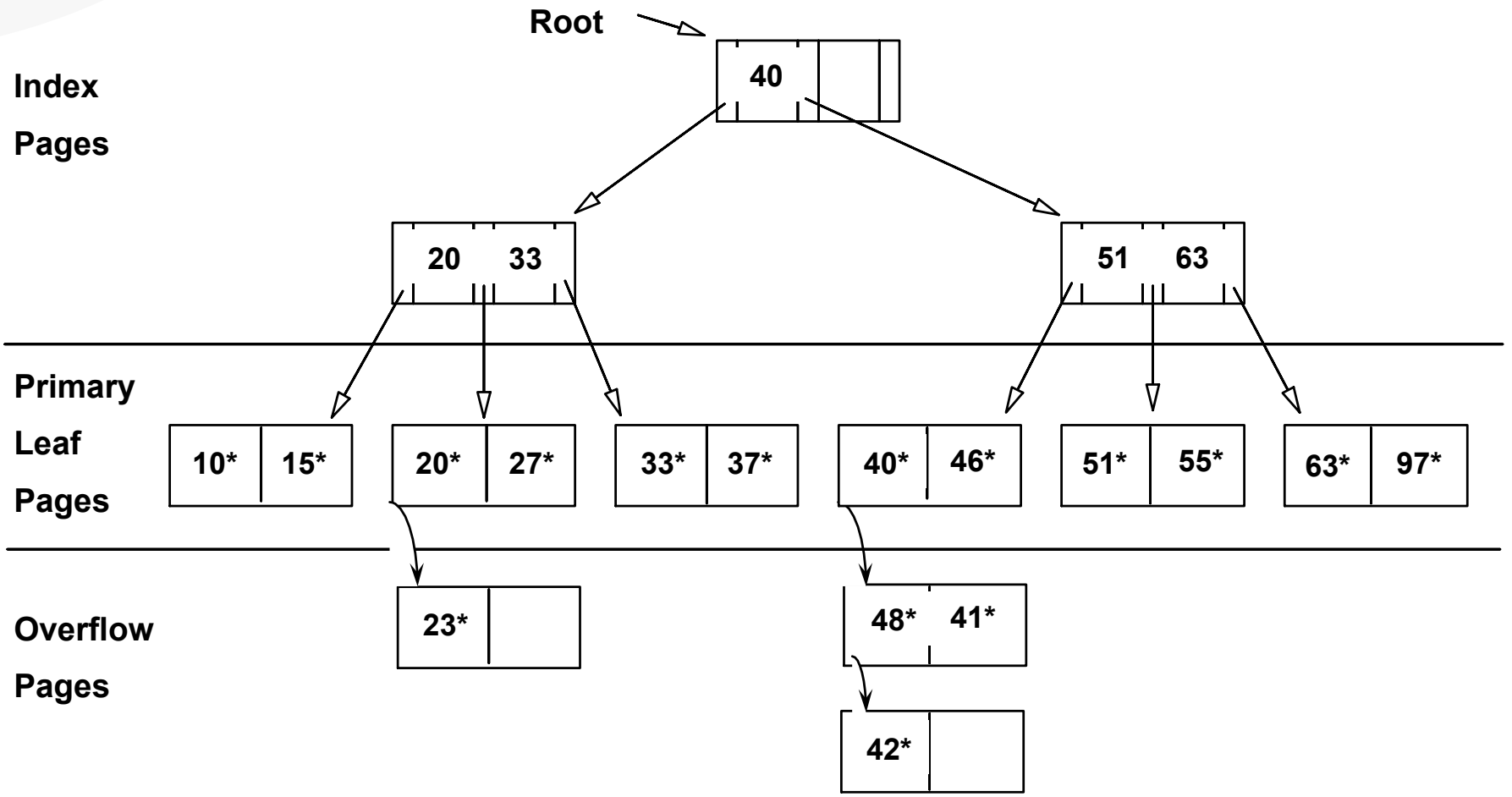
# ISAM is a STATIC Structure



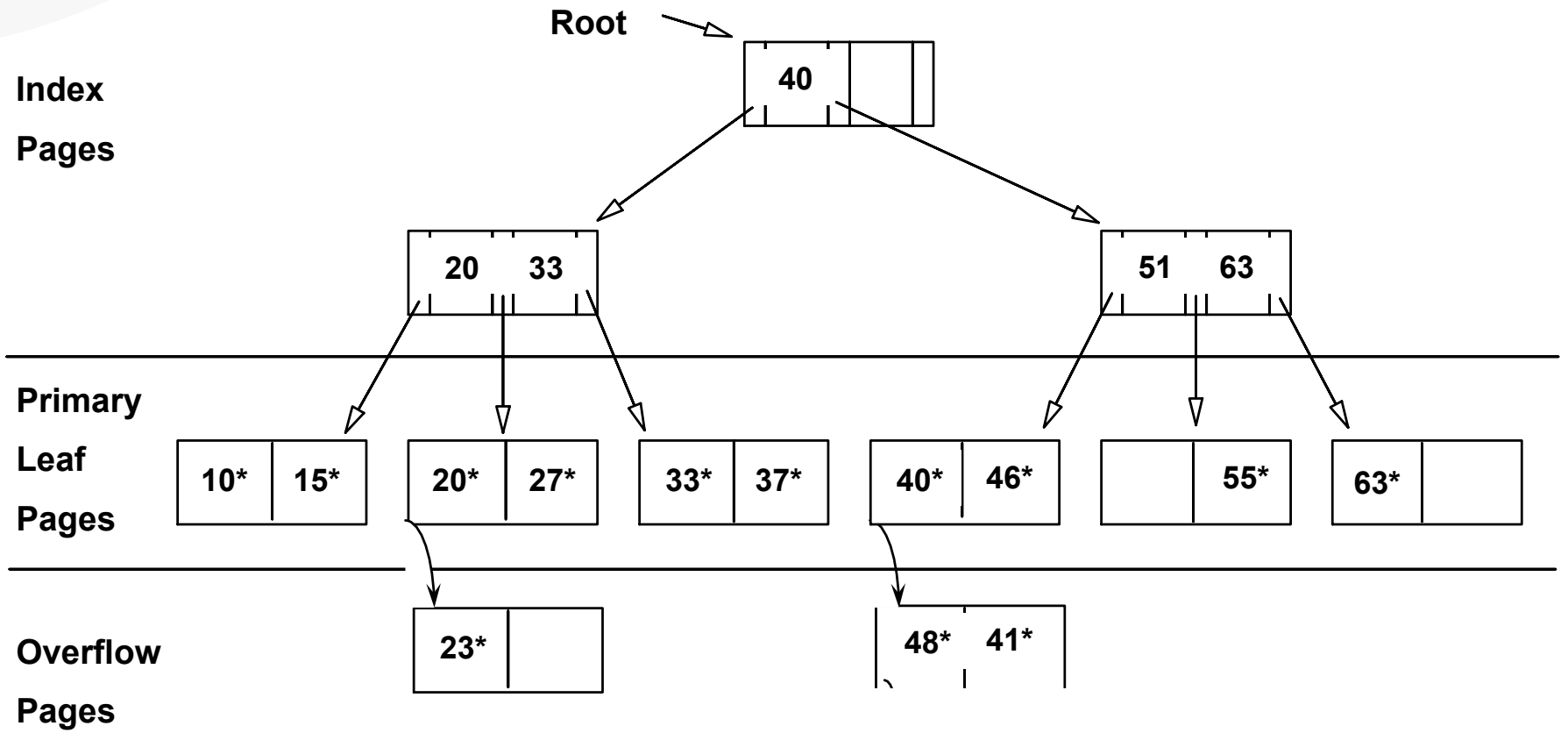

- *File creation*:
  - Leaf (data) pages allocated sequentially, sorted by search key
  - then index pages
  - then overflow pgs.
- *Search*: Start at root; use key comparisons to go to leaf.
- Cost = $\log_F N$
  - F = # entries/pg (i.e., fanout)
  - N = # leaf pgs
  - no need for `next-leaf-page' pointers. (Why?)
- *Insert*: Find leaf that data entry belongs to, and put it there. Overflow page if necessary.
- *Delete*: Seek and destroy! If deleting a tuple empties an overflow page, de-allocate it and remove from linked-list.

**Static tree structure**: *inserts/deletes affect only leaf pages.*

# Example: Insert 23*, 48*, 41*, 42*



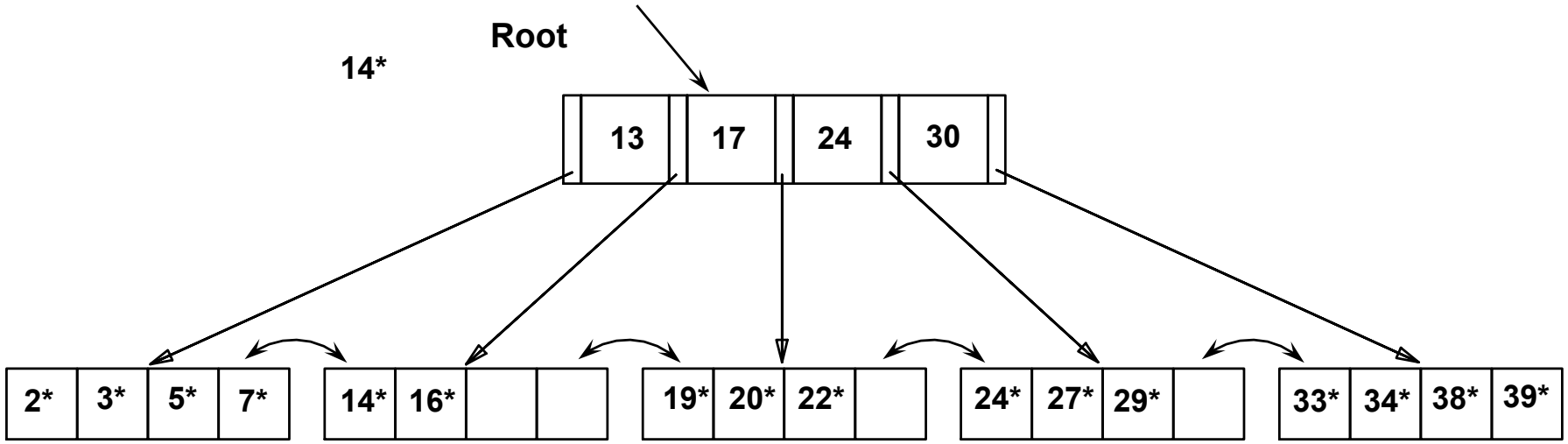
# ... then Deleting 42*, 51*, 97*




☛ *Note that 51* appears in index levels, but not in leaf!*

# Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...

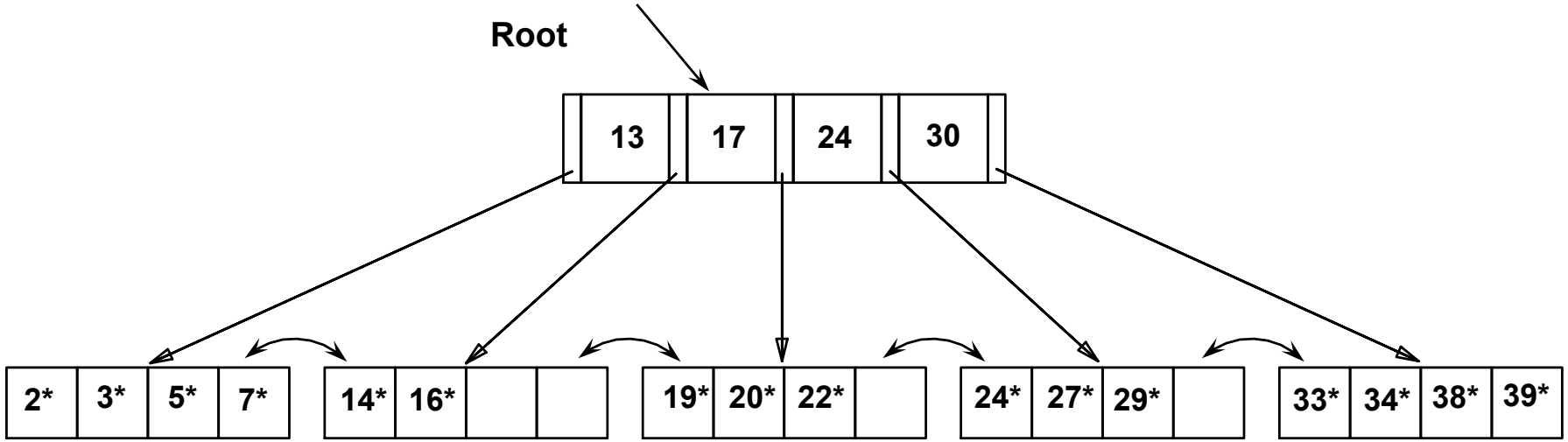



☛ *Based on the search for 15*, we know it is not in the tree!*

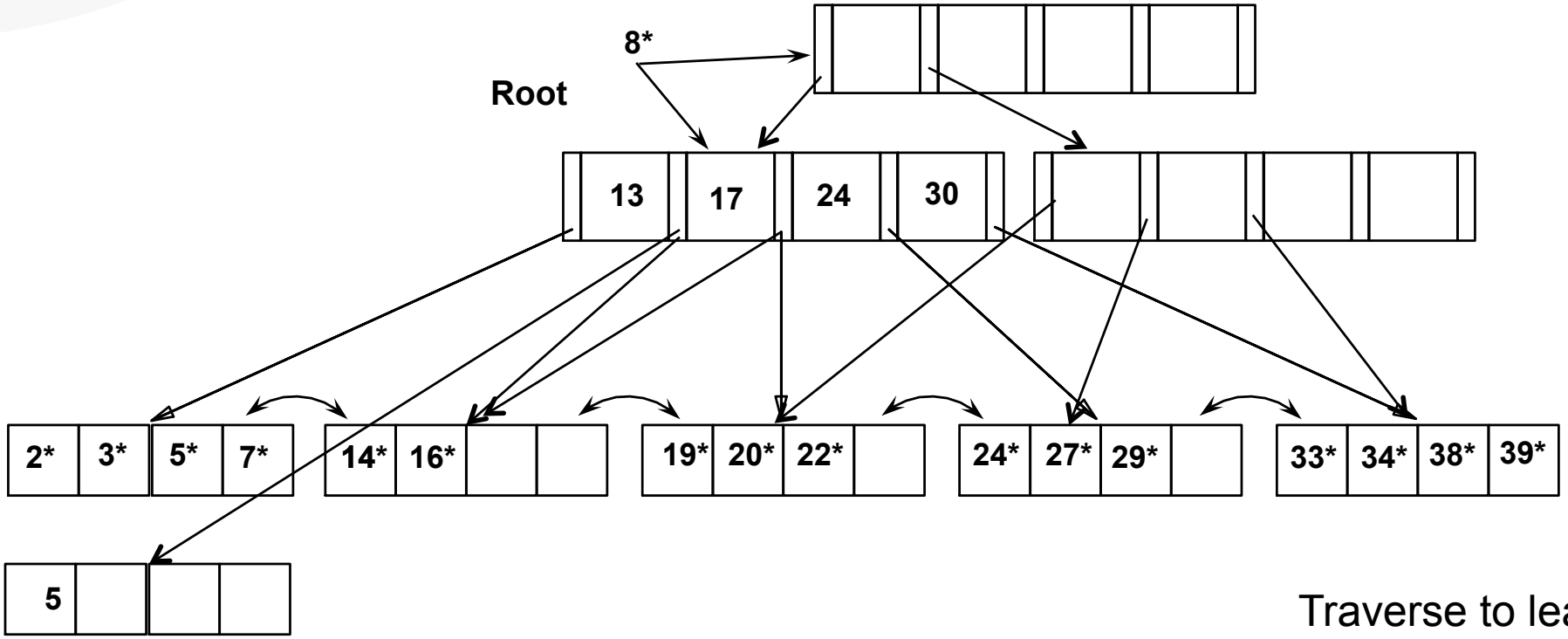
# Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

# Example B+ Tree – Inserting 8*



# Animation: Insert 8*




Traverse to leaf

Split Leaf

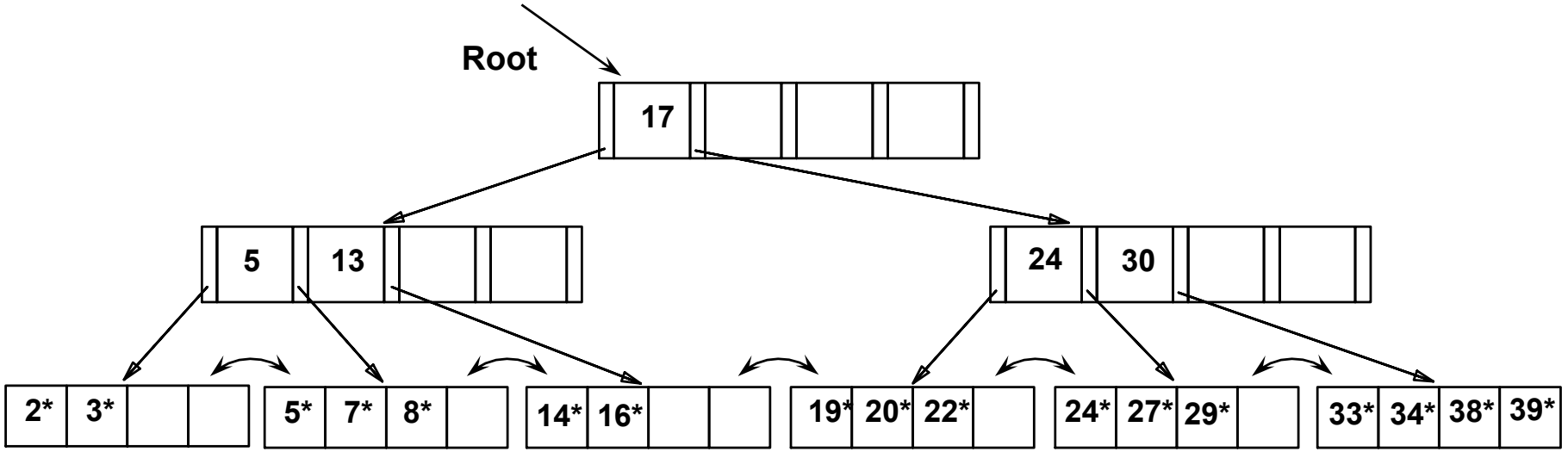
***Copy up*** middle key

Split Parent

New Root

***Push up*** middle key

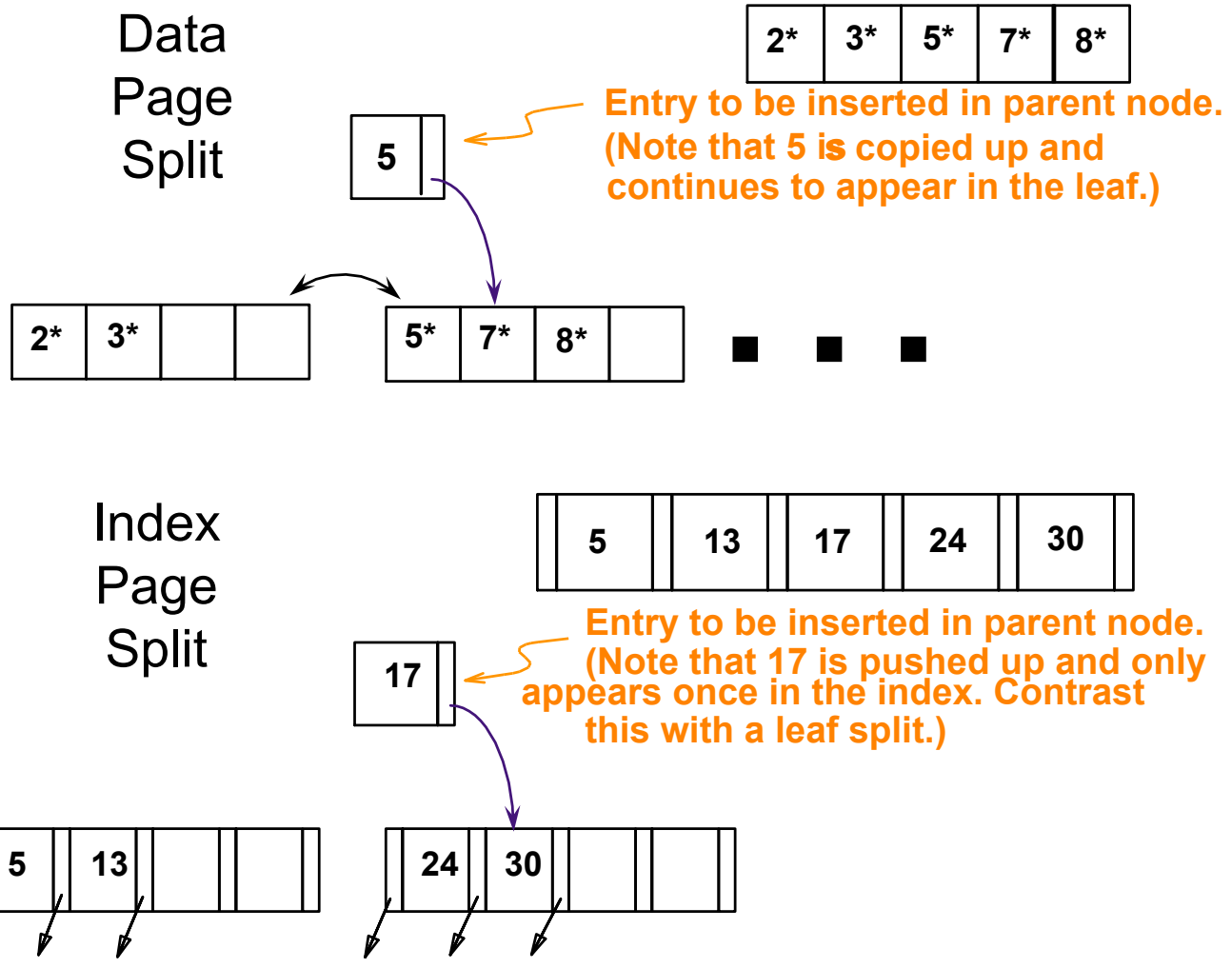
# Final B+ Tree - Inserting 8*




- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

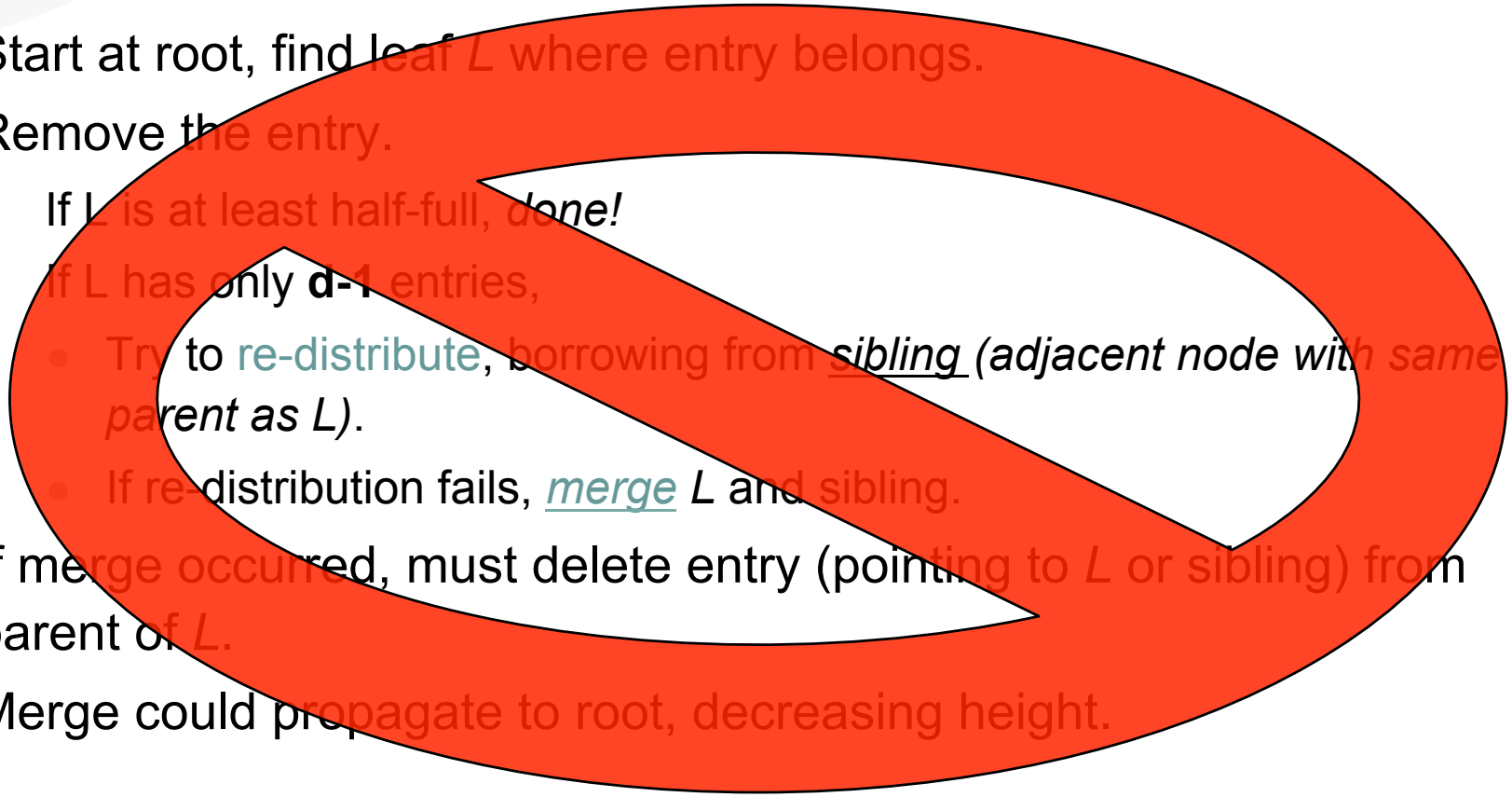
# Data vs. Index Page Split
## (from previous example of inserting "8*")

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.



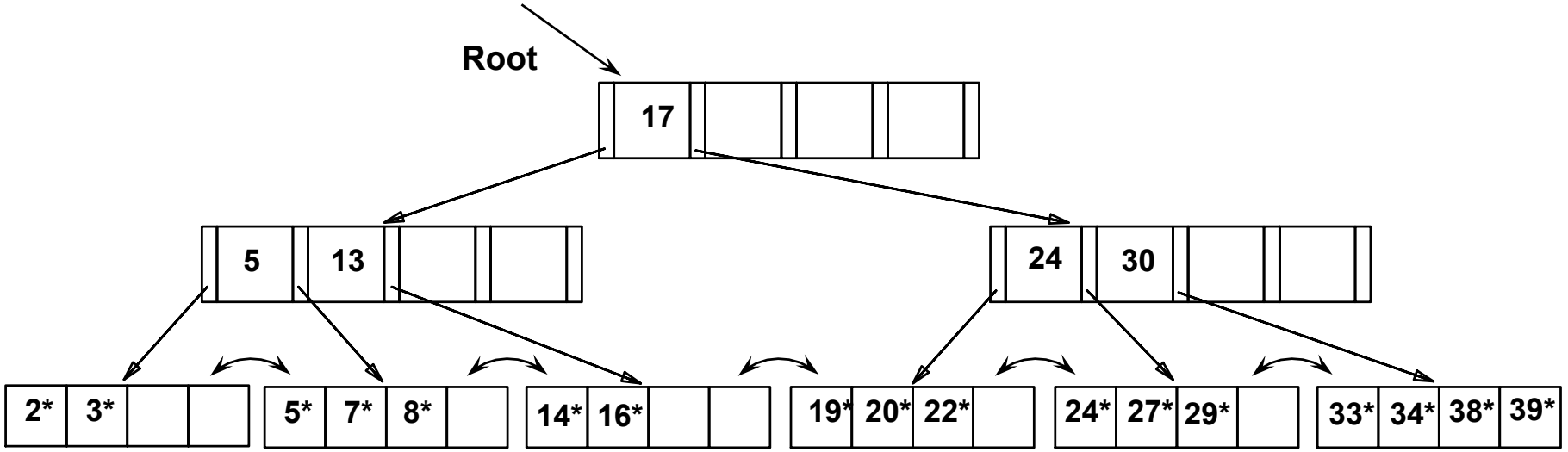
# Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling* (*adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

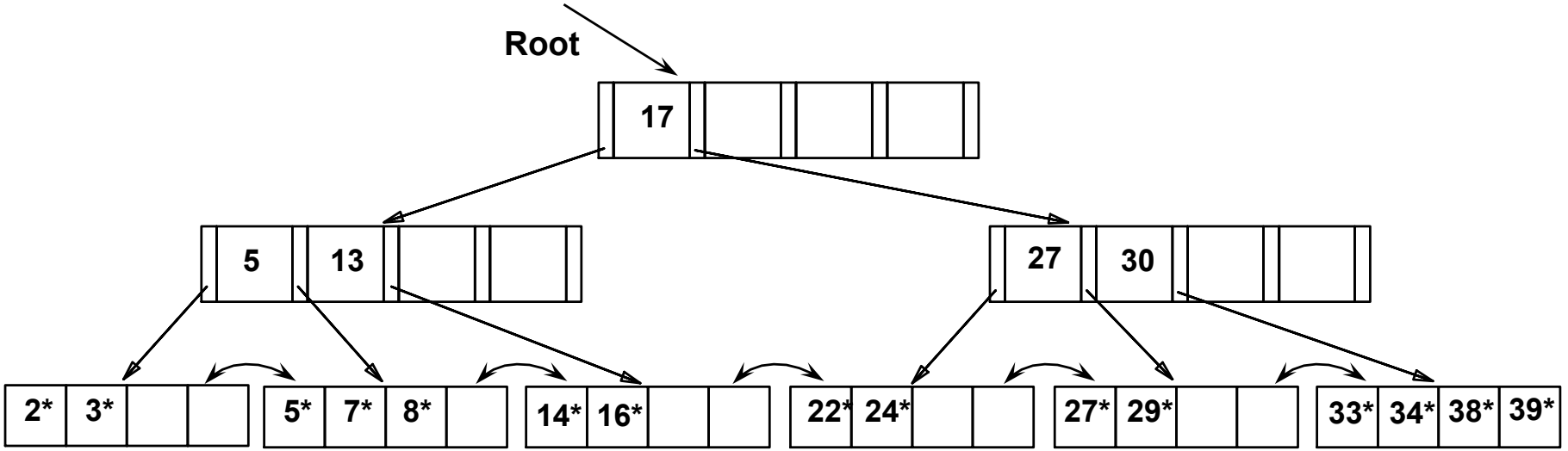


In practice, many systems do not worry about ensuring half-full pages. Just let page slowly go empty; if it's truly empty, just delete from tree and leave unbalanced.

# Example Tree (including 8*)
# Delete 19* and 20* ...



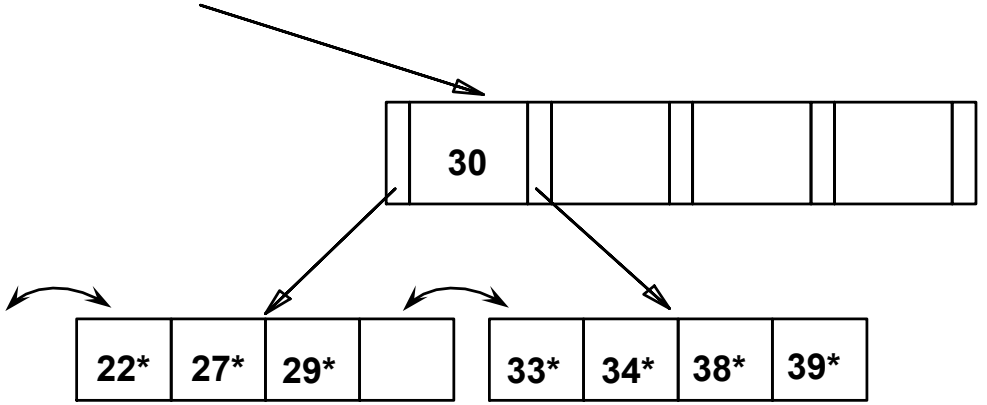
# Example Tree (including 8*)
# Delete 19* and 20* ...

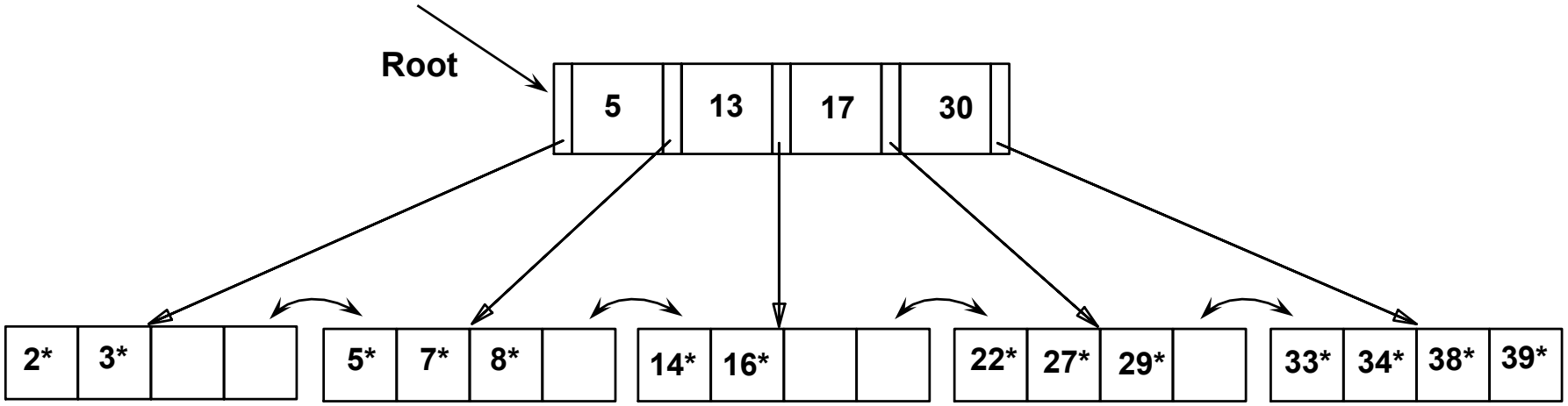



- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

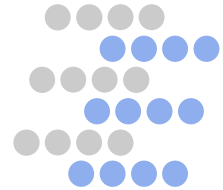
# ... And Then Deleting 24*

- Must merge.
- Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).

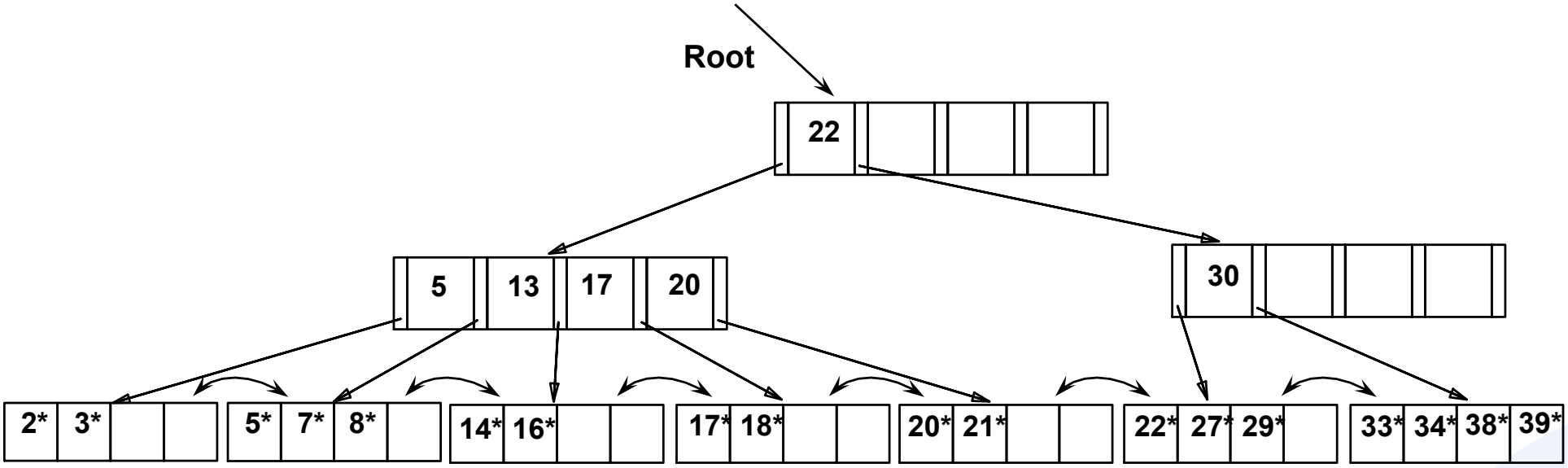


# Example of Non-leaf Re-distribution

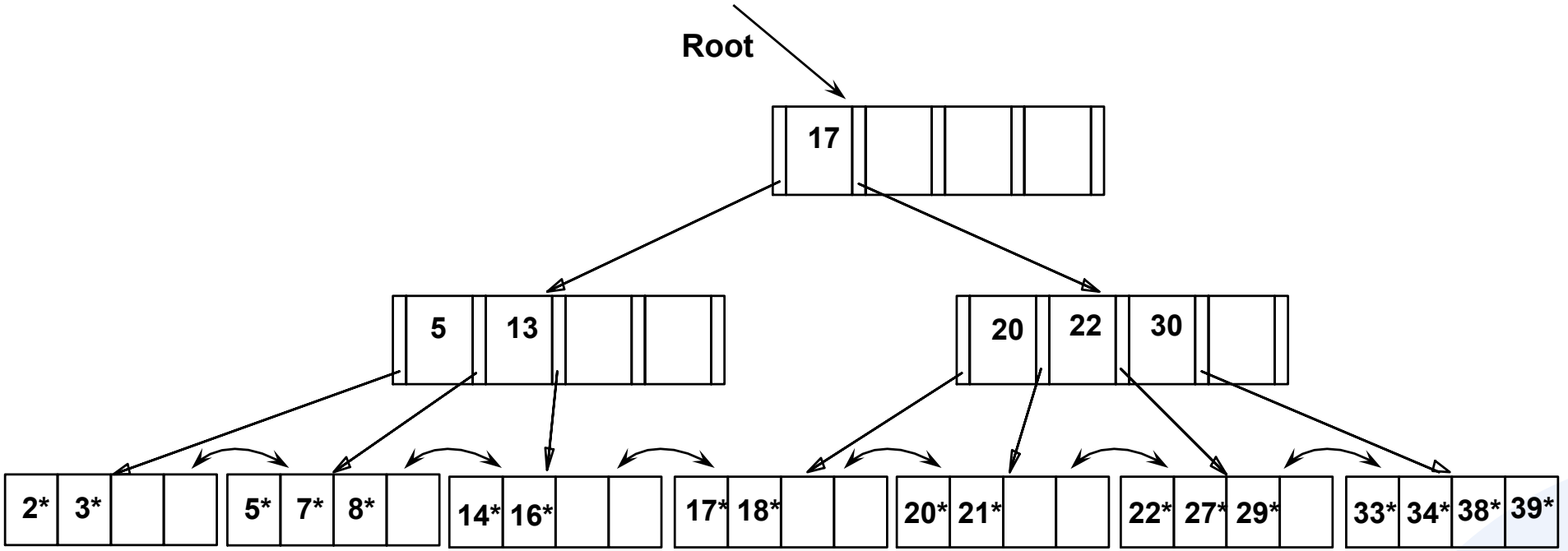


- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.

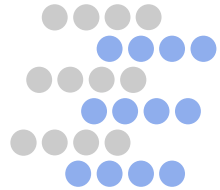


# After Re-distribution

- Intuitively, entries are re-distributed by `*pushing through*' the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.

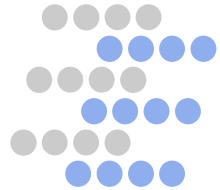


# Prefix Key Compression



- Important to increase fan-out. (Why?)
- Key values in index entries just`direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? It depends on the leaves. What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any descendant leaf) to its left.
- Insert/delete must be suitably modified.

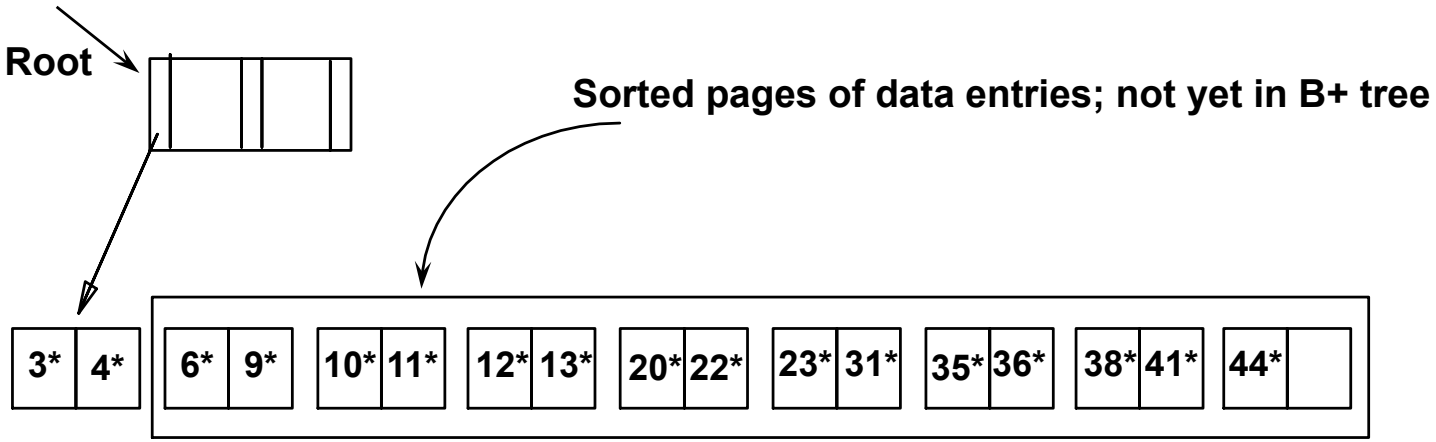
# Suffix Key Compression



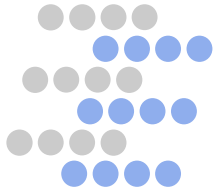
- If many index entries share a common prefix
  - E.g. MacDonald, MacEnroe, MacFeeley
  - Store the common prefix “Mac” at a well known location on the page, use suffixes as split keys
- Particularly useful for composite keys
  - Why?

# Bulk Loading of a B+ Tree

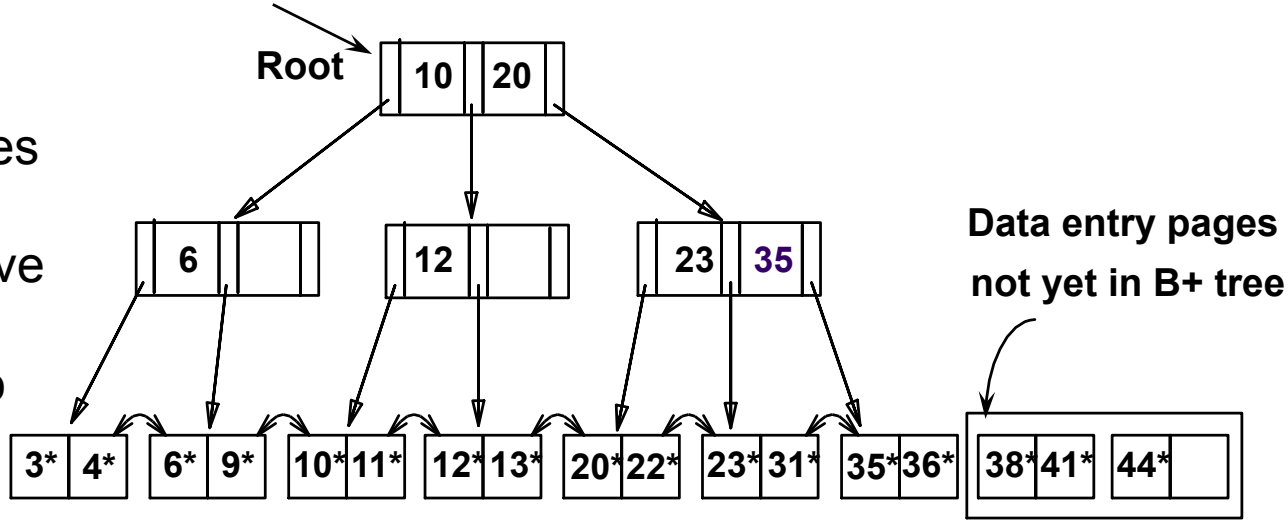
- Given: large collection of records
- Desire: B+ tree on some field
- Bad idea: repeatedly insert records
  - Slow, and poor leaf space utilization . Why?
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



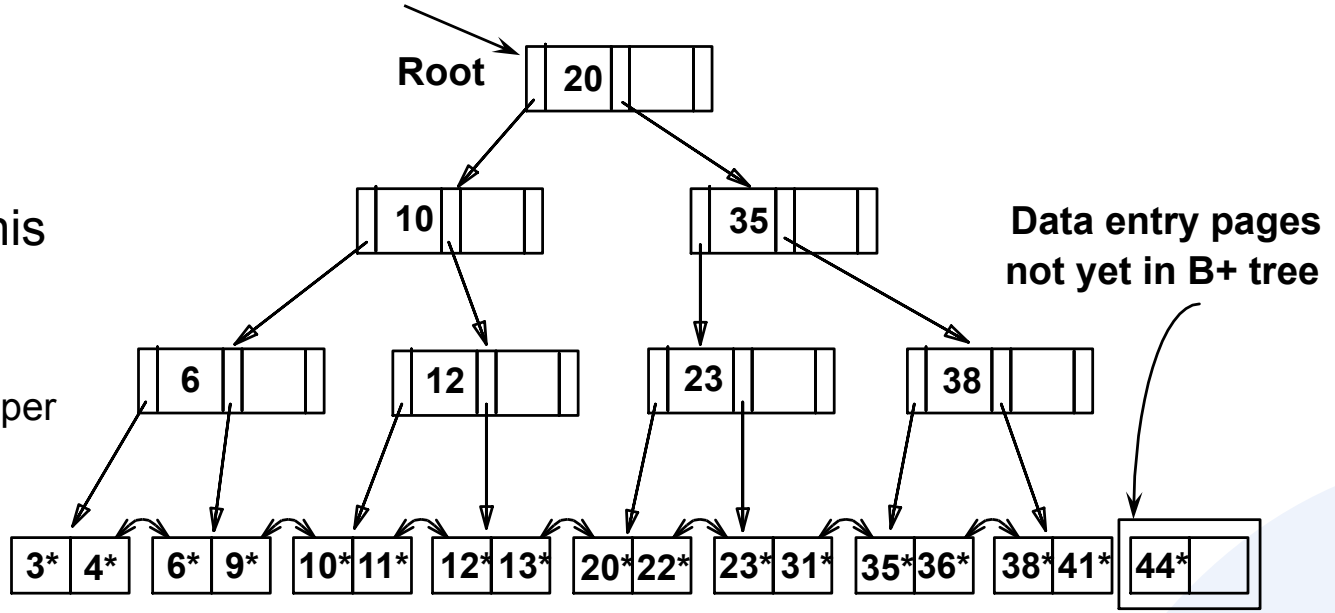
# Bulk Loading (Contd.)



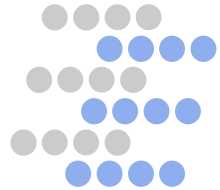
- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts.




- Exercise: what kind of buffer pool hit rate will this give you for different policies?
  - Q1: how many references per page?
  - Q1: how often are they re-referenced?



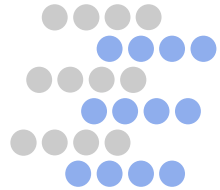
# Summary of Bulk Loading



- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control “fill factor” on pages.

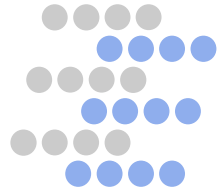
# A Note on `Order'

- *Order* (**d**) makes little sense with variable-length entries
- Use a physical criterion in practice (`*at least half-full*').
  - Index pages often hold many more entries than leaf pages.
  - Variable sized records and search keys:
    - different nodes have different numbers of entries.
  - Even with fixed length fields, Alternative (3) gives variable length
- Many real systems are even sloppier than this --- only reclaim space when a page is *completely* empty.

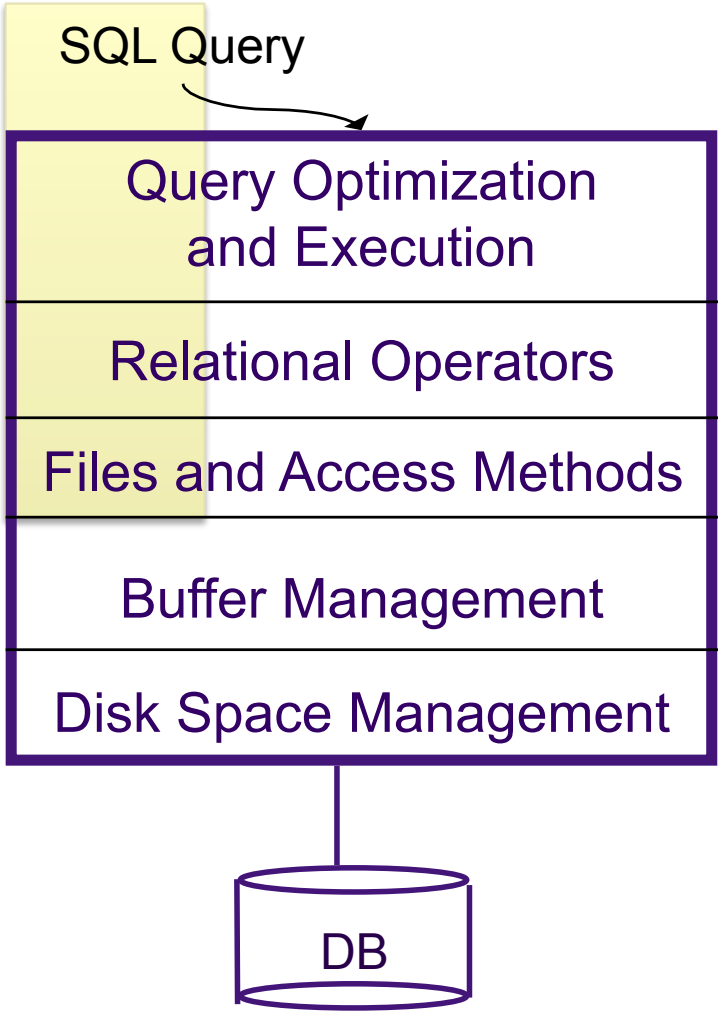


# QUERY PROCESSING: SINGLE-TABLE QUERIES

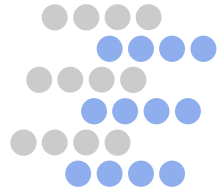
# A “Slice” Through Query Processing



- Start with single-table queries
  - SQL details
  - Query Executor Architecture
  - Simple Query “Optimization”

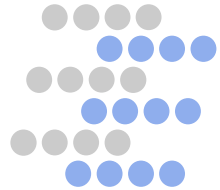


# Basic Single-Table Queries



- SELECT [DISTINCT] *<column expression list>*
  FROM *<single table>*
  [WHERE *<predicate>*]
  [GROUP BY *<column list>*
  [HAVING *<predicate>*] ]
  [ORDER BY *<column list>*]

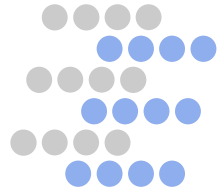
# Basic Single-Table Queries



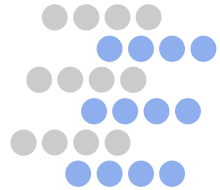
- SELECT [DISTINCT] *<column expression list>*
  FROM *<single table>*
  [WHERE *<predicate>*]
  [GROUP BY *<column list>*
  [HAVING *<predicate>*] ]
  [ORDER BY *<column list>*]

- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
    - Expression can be a column reference, or an arithmetic expression over column refs

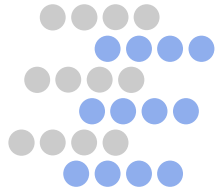
# Basic Single-Table Queries



- ```
  SELECT S.name, S.gpa
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>]
  ```

- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
    - Expression can be a column reference, or an arithmetic expression over column refs
  ```

# SELECT DISTINCT



- ```
  SELECT  DISTINCT  S.name, S.gpa
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>]
  ```

- DISTINCT flag specifies removal of duplicates before output
  ```

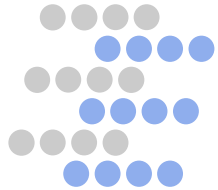
# ORDER BY



```
SELECT  DISTINCT  S.name, S.gpa, S.age*2 AS a2
  FROM Students S
 WHERE S.dept = 'CS'
[GROUP BY <column list>
 [HAVING <predicate>] ]
 ORDER BY S.gpa, S.name, a2;
```

- ORDER BY clause specifies output to be sorted
  - Lexicographic ordering again!
- Obviously must refer to columns in the output
  - Note the AS clause for naming output columns!

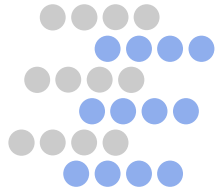
# ORDER BY



```
SELECT  DISTINCT  S.name, S.gpa
  FROM Students S
 WHERE S.dept = 'CS'
[GROUP BY <column list>
 [HAVING <predicate>] ]
 ORDER BY S.gpa DESC, S.name ASC;
```

- Ascending order by default, but can be overriden
  - DESC flag for descending, ASC for ascending
  - Can mix and match, lexicographically

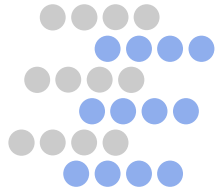
# Aggregates



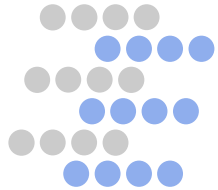
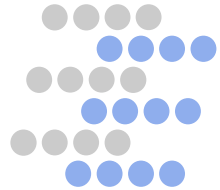
- SELECT [DISTINCT] AVERAGE(S.gpa)
  FROM Students S
  WHERE S.dept = 'CS'
  [GROUP BY *<column list>*
  [HAVING *<predicate>*] ]
  [ORDER BY *<column list>*]

- Before producing output, compute a summary (a.k.a. an *aggregate*) of some arithmetic expression
- Produces 1 row of output
  - with one column in this case
- Other aggregates: SUM, COUNT, MAX, MIN
- Note: can use DISTINCT *inside* the agg function
  - SELECT COUNT(DISTINCT S.name) FROM Students S
  - vs. SELECT DISTINCT COUNT (S.name) FROM Students S;

# GROUP BY

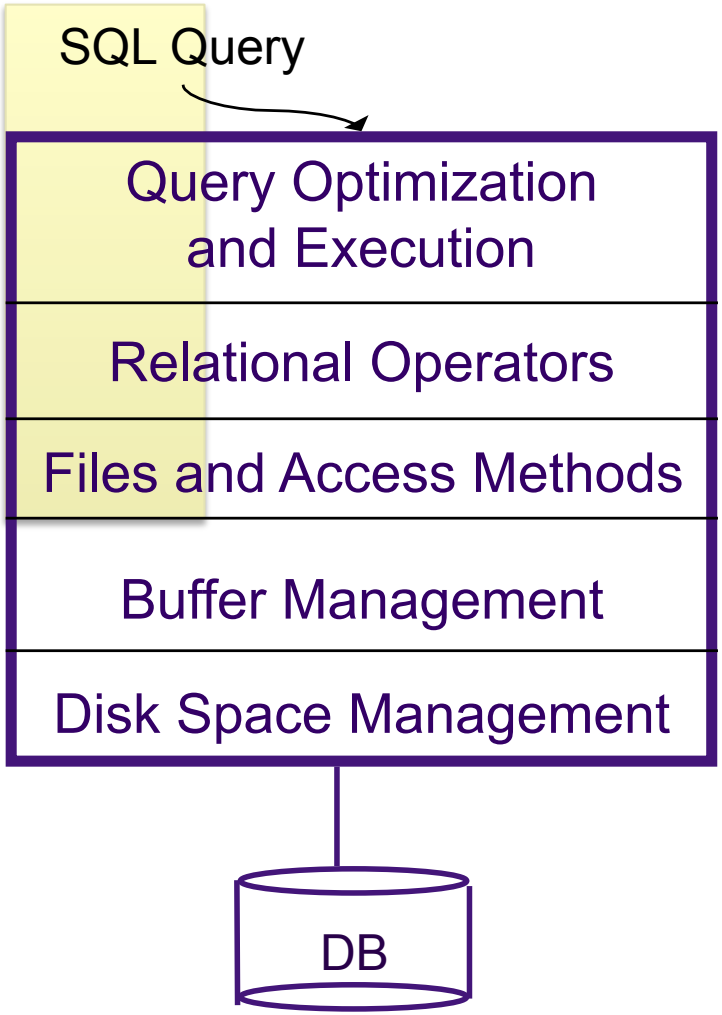


- ```
  SELECT [DISTINCT] AVERAGE(S.gpa), S.dept
    FROM Students S
  [WHERE <predicate>]
   GROUP BY S.dept
    [HAVING <predicate>]
  [ORDER BY <column list>]
  ```

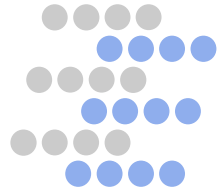
- Partition table into groups with same GROUP BY column values
  - Can group by a list of columns
- Produce an aggregate result per group
  - Cardinality of output = # of distinct group values
- Note: can put grouping columns in SELECT list
  - For aggregate queries, SELECT list can contain aggs and GROUP BY columns only!
  - What would it mean if we said SELECT S.name, AVERAGE(S.gpa) above??
  ```

# HAVING



- SELECT [DISTINCT] AVERAGE(S.gpa), S.dept
  FROM Students S
  [WHERE *<predicate>*]
  GROUP BY S.dept
  HAVING COUNT(*) > 5
  [ORDER BY *<column list>*]

- The HAVING predicate is applied *after* grouping and aggregation
  - Hence can contain anything that could go in the SELECT list
  - I.e. aggs or GROUP BY columns
- HAVING can only be used in aggregate queries
- It's an optional clause

# Putting it all together



- ```
  SELECT S.dept, AVERAGE(S.gpa), COUNT(*)
    FROM Students S
   WHERE S.gender = “F”
   GROUP BY S.dept
    HAVING COUNT(*) > 5
   ORDER BY S.dept;
  ```
  ```

# Context

- We looked at SQL
- Now shift gears and look at Query Processing

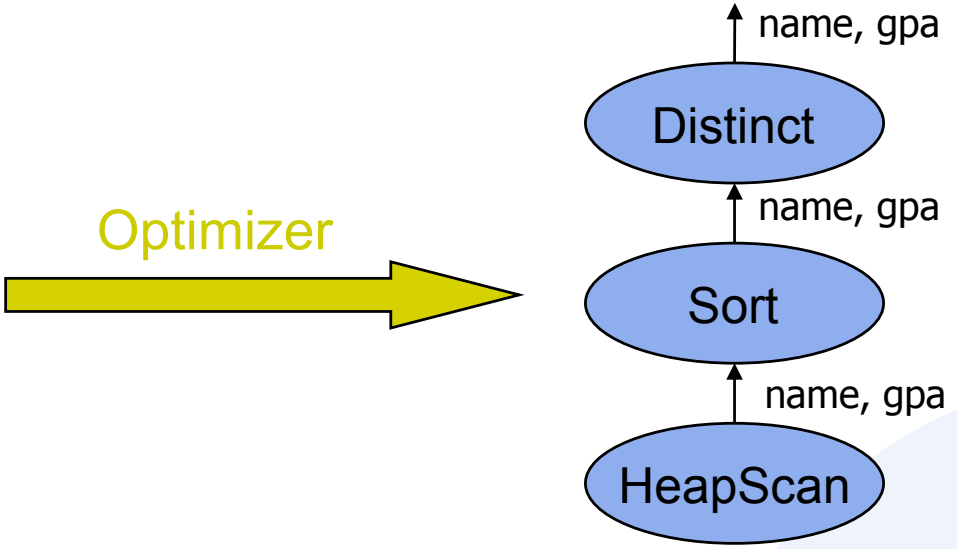


# Query Processing Overview

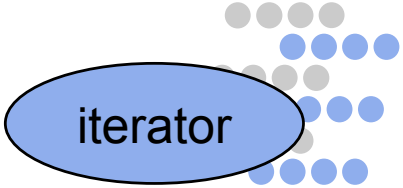


- The *query optimizer* translates SQL to a special internal "language"
  - Query Plans
- The *query executor* is an *interpreter* for query plans
- Think of query plans as "blobs-and-arrows" *dataflow* diagrams
  - Each blob implements a *relational operator*
  - Edges represent a flow of tuples (columns as specified)
  - For single-table queries, these diagrams are straight-line graphs

```
SELECT DISTINCT name, gpa
  FROM Students
```



# Iterators




- The relational operators are all subclasses of the class `iterator`:

```
class iterator {
   void init();
   tuple next();
   void close();
   iterator &inputs[];
   // additional state goes here
}
```

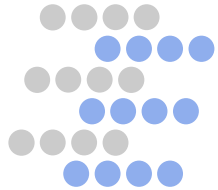
- Note:
  - Edges in the graph are specified by inputs (max 2, usually)
  - Encapsulation: any iterator can be input to any other!
  - When subclassing, different iterators will keep different kinds of state information

# Example: Sort

```
class Sort extends iterator {
        void init();
        tuple next();
        void close();
    iterator &inputs[1];
      int numberOfRuns;
      DiskBlock runs[];
       RID nextRID[];
            }
```

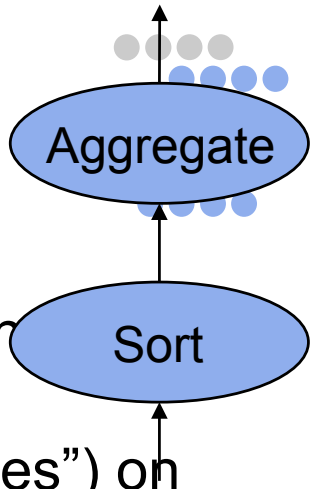
- `init()`:
  - generate the sorted runs on disk
  - Allocate `runs[]` array and fill in with disk pointers.
  - Initialize numberOfRuns
  - Allocate nextRID array and initialize to NULLs
- `next()`:
  - nextRID array tells us where we're "up to" in each run
  - find the next tuple to return based on nextRID array
  - advance the corresponding nextRID entry
  - return tuple (or EOF -- "End of Fun" -- if no tuples remain)
- `close()`:
  - deallocate the runs and nextRID arrays

# Postgres Version



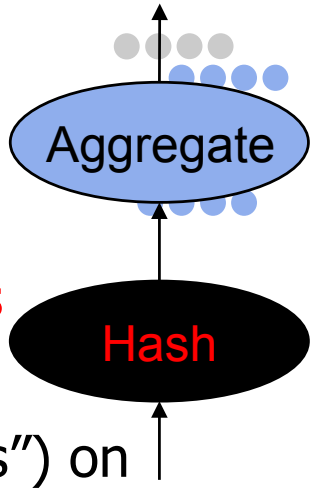
- src/backend/executor/nodeSort.c
  - ExecInitSort (init)
  - ExecSort (next)
  - ExecEndSort (close)
- The encapsulation stuff is hardwired into the Postgres C code
  - Postgres predates even C++!
  - See src/backend/execProcNode.c for the code that “dispatches the methods” explicitly!

# Sort GROUP BY




- The Sort iterator ensures all its tuples are output in sequen
- The Aggregate iterator keeps running info ("transition values") on agg functions in the SELECT list, per group
  - E.g., for COUNT, it keeps count-so-far
  - For SUM, it keeps sum-so-far
  - For AVERAGE it keeps sum-so-far and count-so-far
- As soon as the Aggregate iterator sees a tuple from a new group:
  - It produces output for the old group based on agg function
    - E.g. for AVERAGE it returns (sum-so-far/count-so-far)
  - It resets its running info.
  - It updates the running info with the new tuple's info
- Most DBMSs implement this. So does Hadoop (for "Reduce")

# Hash GROUP BY (naïve)

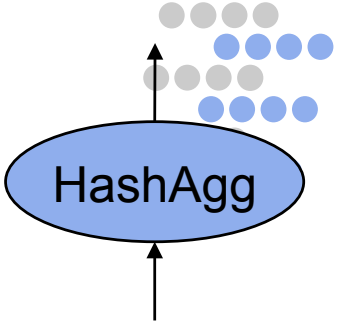



- The Hash iterator ensures all its tuples are output in batches

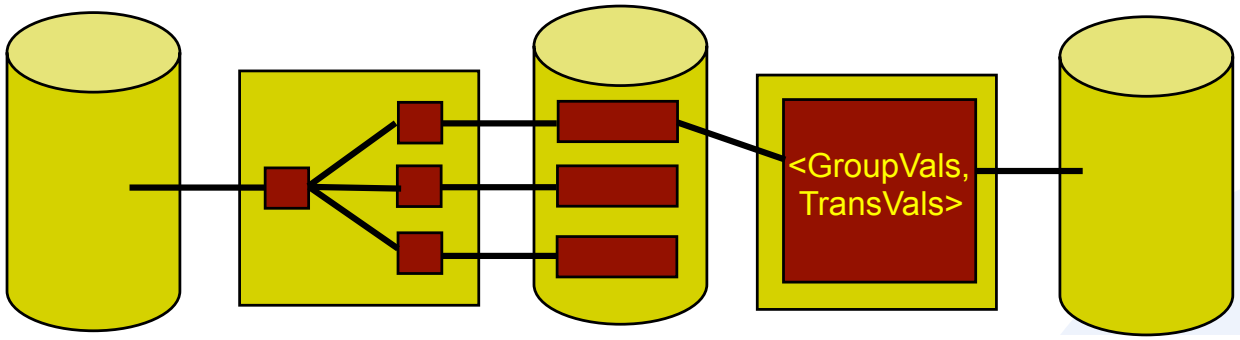
- The Aggregate iterator keeps running info ("transition values") on agg functions in the SELECT list, per group
  - E.g., for COUNT, it keeps `count-so-far`
  - For SUM, it keeps `sum-so-far`
  - For AVERAGE it keeps `sum-so-far` *and* `count-so-far`

- As soon as the Aggregate iterator sees a tuple from a new group:
  1. It produces output for the old group based on agg function
     - E.g. for AVERAGE it returns (`sum-so-far/count-so-far`)
  2. It resets its running info.
  3. It updates the running info with the new tuple's info

# We Can Do Better!

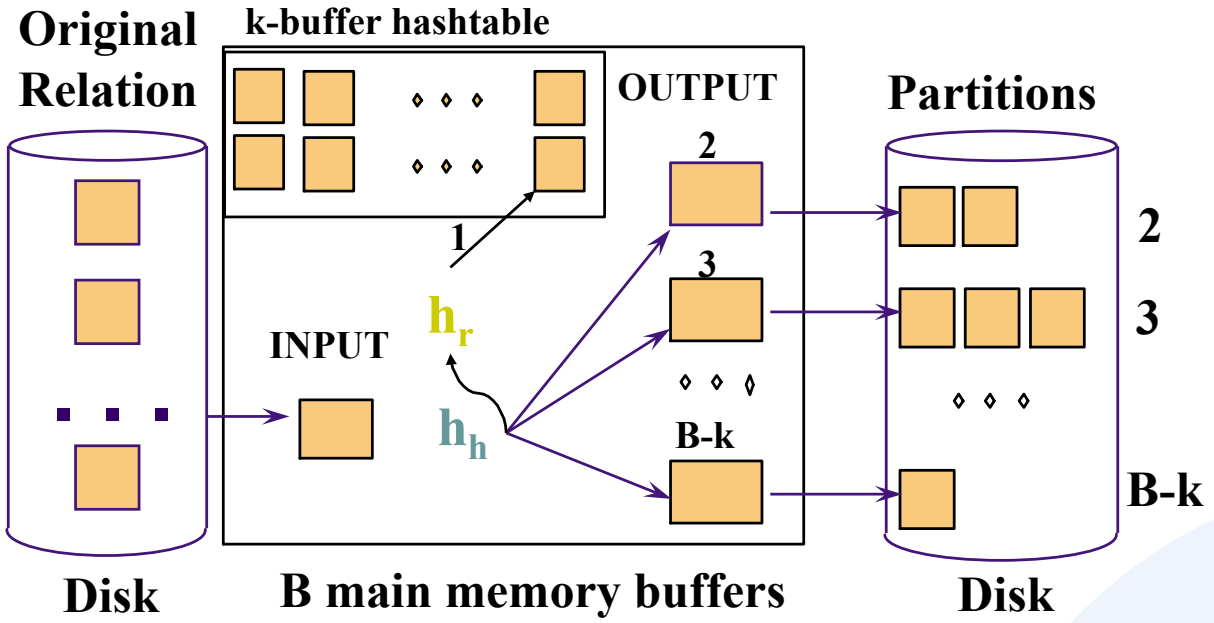



- Combine the summarization into the hashing process
  - During the ReHash phase, don't store tuples, store pairs of the form <GroupVals, TransVals>
  - When we want to insert a new tuple into the hash table
    - If we find a matching GroupVals, just update the TransVals appropriately
    - Else insert a new <GroupVals,TransVals> pair
- What's the benefit?
  - Q: How many pairs will we have to hash?
  - A: Number of distinct values of GroupVals columns
    - Not the number of tuples!!
  - Also probably "narrower" than the tuples

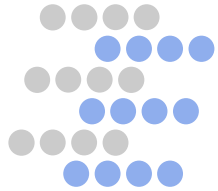


# A Hashing Tweak: Hybrid Hashing

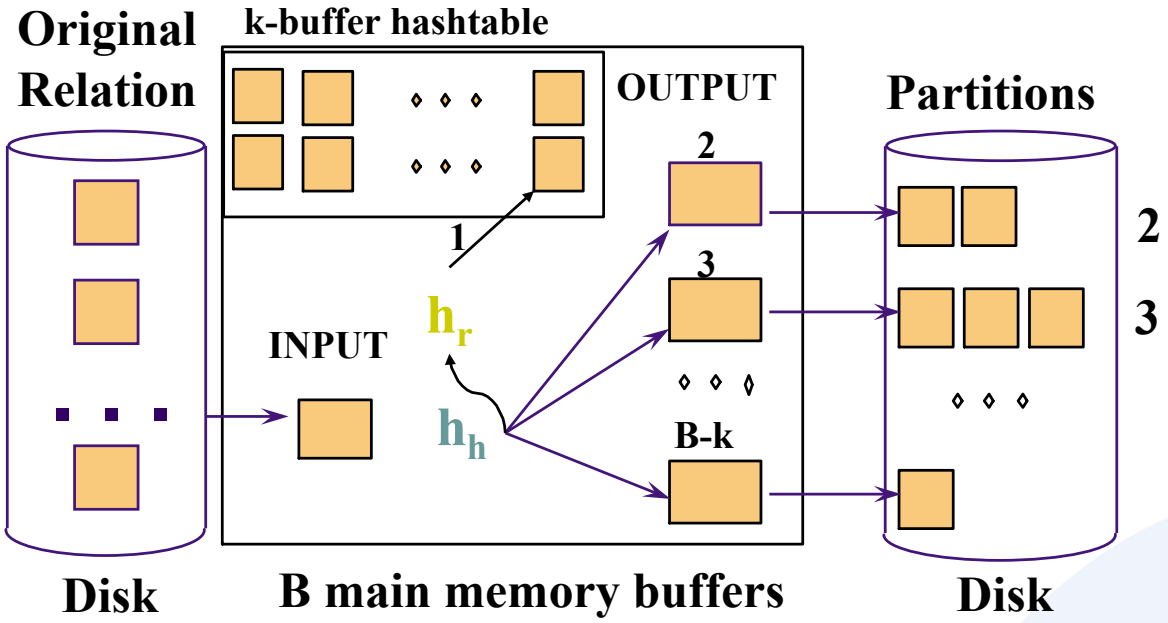
- What if the set of <GroupVals,TransVals> pairs fits in memory
  - It would be a waste to spill it to disk and read it all back!
  - Recall this could be true even if there are *tons* of tuples!
- Idea: keep a smaller 1st partition in memory during phase 1!
  - Output its stuff at the end of Phase 1.
  - Q: how do we choose the number k?



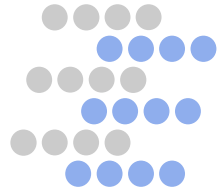
# A Hash Function for Hybrid Hashing



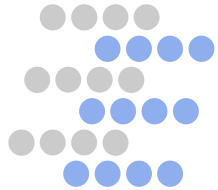
- Assume we like the hash-partition function $h_p$
- Define $h_h$ operationally as follows:
  - $h_h(x) = 1$ if in-memory hashtable is not yet full
  - $h_h(x) = 1$ if x is already in the hashtable
  - $h_h(x) = h_p(x)$ otherwise
- This ensures that:
  - Bucket 1 fits in k pages of memory
  - If the entire set of distinct hashtable entries is smaller than k, we do *no spilling!*



# Summary

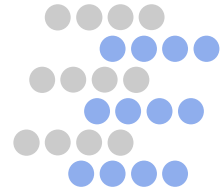


- Intro to SQL aggregation, etc.
- Iterator architecture of a query executor
  - Streams data through operators by “pulling”
    - Lazy evaluation
  - Encapsulates operator logic
    - Nice match to “logical data independence”
- Hybrid Hashing:
  - Can be a huge win if #groups is small
    - A case where hashing can kill sorting!
  - Probably should replace Sort in many cases
    - E.g. MapReduce



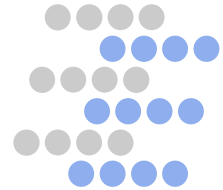
# WEB DATABASES

# The Motivation for Web Databases

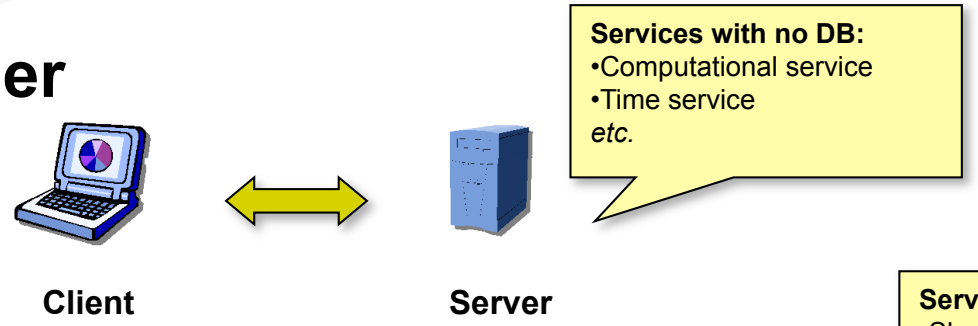


- The Web is about accessing data
- Two approaches for displaying data
  - Embed data in static Web pages
  - Dynamically generate Web pages from "raw" data
- Dynamic Web pages must be preferred when
  - Data is large
  - Data must be displayed in different ways
  - Data may change over time
- Data is generally stored in a database
  - The Web server interacts with the DB and generates pages
  - In simple cases, the file system can be sufficient
  - How could Amazon create web pages for every single item in their inventory and how could they keep all those pages up to date?
- 95% or more of all servers use some sort of database

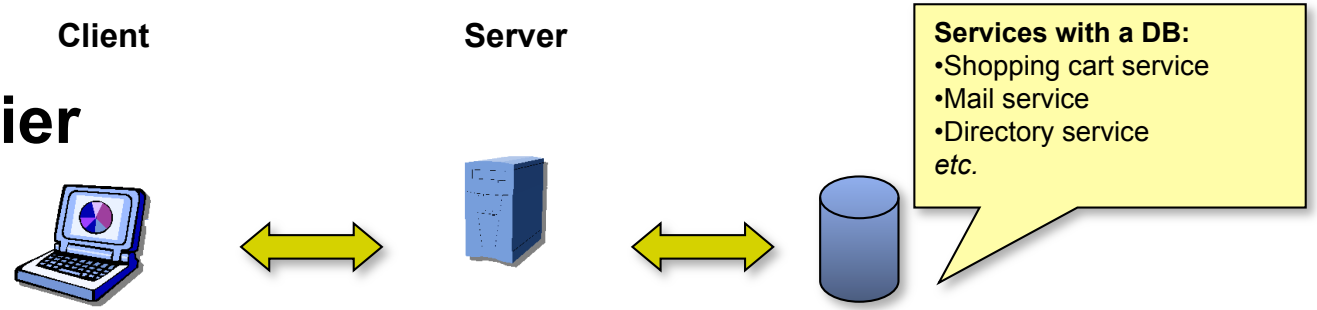
# Common Architectures



## 2-Tier

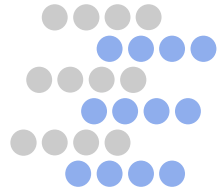



## 3-Tier




- Note: technically, files *are* databases
  - A database holds information and provides for a mechanism to access this information
- In practice, we are interested in full-featured DBs

# Interacting with Databases



- SQL: Structured Query Language
  - Pronounced "es queue el" or "sequel"
  - Used in almost all DBMSs (ANSI standard)
  - Set of statements that define and manipulate (insert, update retrieve) data
    - `SELECT ... FROM ... WHERE ...`
    - `INSERT INTO ... SET ...`
    - `UPDATE ... SET ... WHERE ...`
    - `DELETE FROM ... WHERE ...`
    - `CREATE TABLE ...`
    - etc.

# SQL Examples

```
SELECT Name FROM Employees
```

**"Employees" table**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

**Result**

| Name |
|---|
| Alice |
| Bob |
| Carole |

# SQL Examples (cont'd)

```
SELECT Name,Salary FROM Employees
```

**"Employees" table**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

**Result**

| Name | Salary |
|---|---|
| Alice | 100,000 |
| Bob | 88,000 |
| Carole | 112,000 |

# SQL Examples (cont'd)

```
SELECT Name FROM Employees
  WHERE ID = 222
```

**"Employees" table**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

**Result**

| Name |
|---|
| Bob |

# SQL Examples (cont'd)

```
SELECT * FROM Employees
  WHERE ID = 222
```

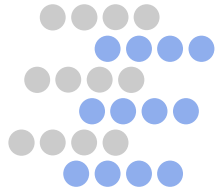
**"Employees" table**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

**Result**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Bob | 222 | 88,000 | 3 |

# SQL Examples (cont'd)



```
SELECT Name,Salary FROM Employees
  WHERE Salary >= 100,000
```

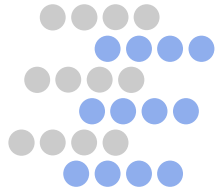
**"Employees" table**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

**Result**

| Name | Salary |
|---|---|
| Alice | 100,000 |
| Carole | 112,000 |

# SQL Examples (cont'd)



```
SELECT Name,Salary FROM Employees
  WHERE Salary >= 100,000
  AND DeptID = 3
```

**"Employees" table**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

**Result**

| Name | Salary |
|---|---|
| Alice | 100,000 |

# SQL Examples (cont'd)

```
SELECT DISTINCT DeptID
  FROM Employees
```

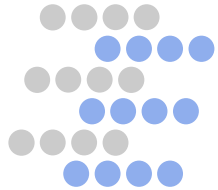
**"Employees" table**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

**Result**

| Dept ID |
|---|
| 3 |
| 2 |

# SQL Examples (cont'd)



```
SELECT Name,Salary FROM Employees
  ORDER BY Salary DESC
```

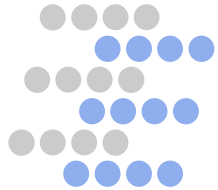
**"Employees" table**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

**Result**

| Name | Salary |
|---|---|
| Carole | 112,000 |
| Alice | 100,000 |
| Bob | 88,000 |

# SQL Examples (cont'd)



```
SELECT Employees.Name,Departments.Name
  FROM Employees,Departments
  WHERE Employees.ID = Departments.ManagerID
```

**"Employees" table**

| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

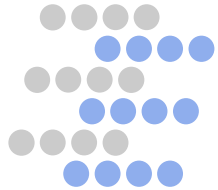
**"Departments" table**

| Dept ID | Name | Manager ID |
|---|---|---|
| 2 | Billing | 444 |
| 3 | Shipping | 111 |

**Result**

| E.Name | D.Name |
|---|---|
| Alice | Shipping |

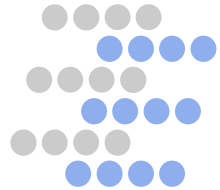
# Transactions



```
TX1: acc1.withdraw($100);      TX2: acc1.deposit($200);
     acc2.deposit($100);           acc2.withdraw($200);
```

- Transaction: group actions together so they are
  - *Atomic:* either happens or it doesn't; no partial operations
  - *Consistent:* maintains system invariants
  - *Isolated:* serializable; transactions appear to happen one after another
  - *Durable:* once it happens, stays happened
- Transaction termination
  - *Commit:* transaction is done (visible, durable)
  - *Rollback:* "forget" uncomitted transaction (e.g., if failure occurs in middle of transaction, it didn't happen at all)

# MySQL



- Multi-user, medium-scale RDBMS
- Open source, GPL
- Support for very large quantities of data
- Supports SQL as DB interaction language
- Large user base (~3,000,000)
- Transactional storage engine
- Master/slave replication for high availability
- APIs for C, C++, Java, List, Ada, PHP, Matlab, Python, etc.

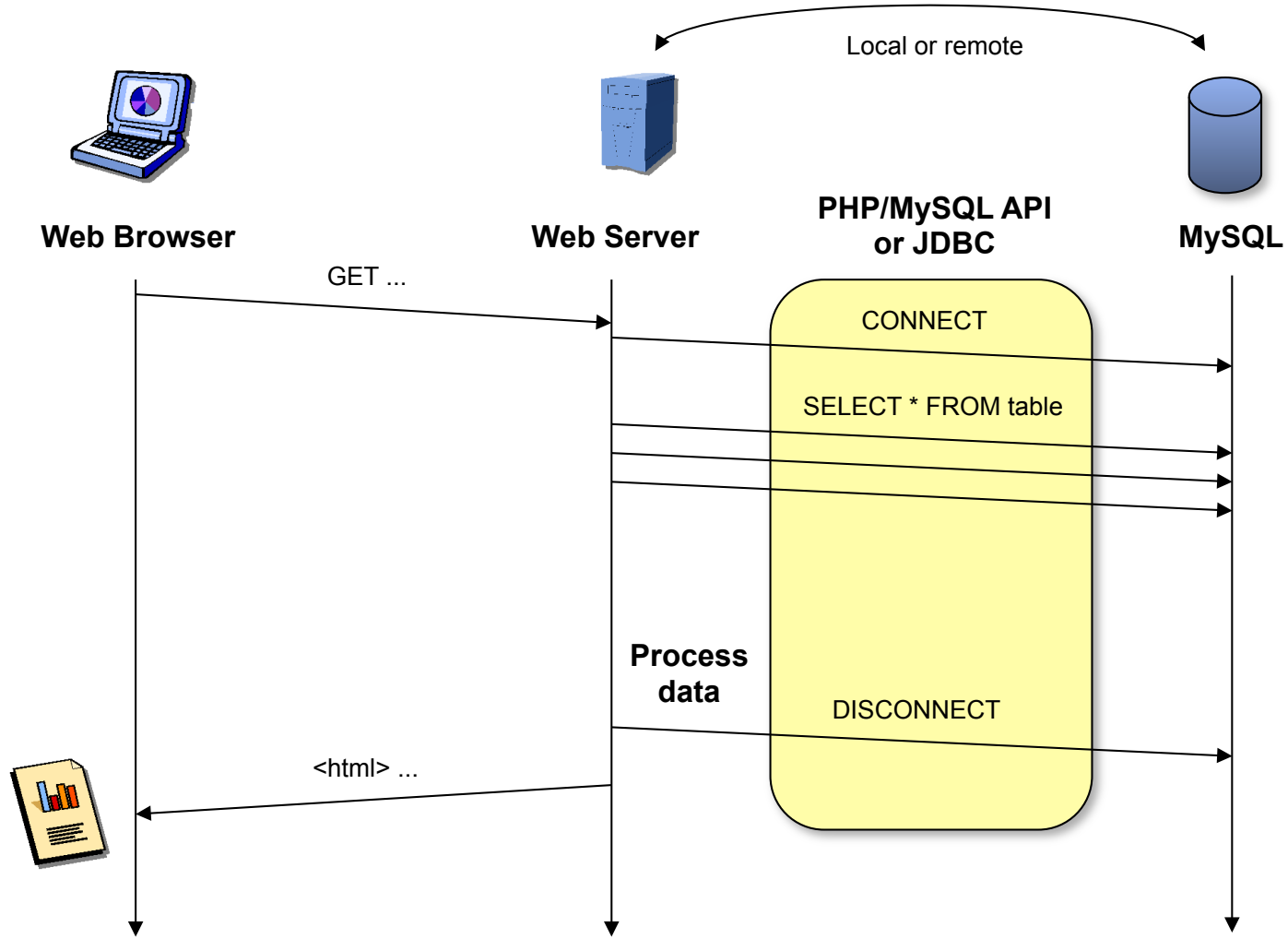
# Sample MySQL Session

```
% mysql -h host.eurecom.fr -u scott -ptiger
> USE people_db;
> SHOW TABLES;
> CREATE TABLE Employees
  ( ID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(64),
    Salary INT,
    DeptID INT );
> INSERT INTO Employees( Name, Salary, DeptID )
  VALUES ( "Alice", 100000, 3 );
> SELECT * FROM Employees;
> SELECT * FROM Employees WHERE Id = 111;
> DELETE FROM Person WHERE Id = 111;
> exit;
```

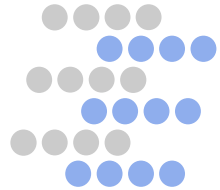
**"Employees" table**

| ID | Name | Salary | Dept ID |
|---|---|---|---|
| 111 | Alice | 100,000 | 3 |

# MySQL as Web Database

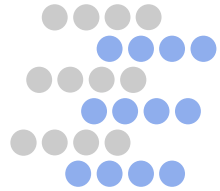


# MySQL Support in PHP



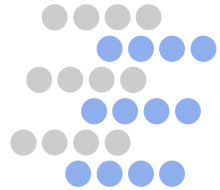
- PHP includes native support for accessing MySQL
  - About 50 `myslq_*` functions to connect, query databases, and retrieve results
- Database interaction is a five-steps process
  1. Connect to the DMBS and use a DB
  2. Run the query
  3. Retrieve a row of results
  4. Process the attribute values
     *Repeat steps 3 and 4*
  5. Close the DMBS connection

# Opening a DB Connection



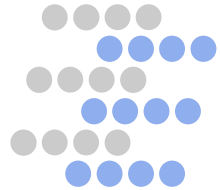
- Connect and disconnect
  - `mysql_connect(string host, string user, string pwd)`
    *Establishes a connection to the MySQL RDBMS*
  - `mysql_close()`
    *Closes a previously-opened MySQL connection*
- Managing databases
  - `mysql_select_db(string db_name)`
    *Uses the specified database on a connection*
  - `mysql_[create|drop]_db(string db_name)`
    *Creates/deletes a database*

# Performing Queries



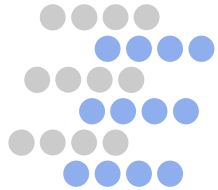
- One function to do (almost) everything
  - `resource mysql_query(string sql_command)`
    *Runs the SQL statement and return a resource*
- Can be used to create and delete tables
  - `mysql_query("CREATE TABLE Employees...")`
  - `mysql_query("DROP TABLE Employees")`
- Can be used for transactions
  - `mysql_query("BEGIN")`
  - `mysql_query("COMMIT")`
  - `mysql_query("ROLLBACK")`
- Etc.

# Using Queries Results



- Queries returned "result sets"
  - `array mysql_fetch_row(resource result_set)`
    *Fetches the result set one row at a time*
  - `array mysql_fetch_array(resource result_set)`
    *Fetches row data and their table attribute names*
  - `integer mysql_num_fields (resource result_set)`
    *Returns the number of attributes in the result set*
  - `integer mysql_num_rows (resource result_set)`
    *Returns the number of rows in the result set*
  - `mysql_free_result(resource result_set)`
    *Frees the resources associated with the result set*

# Processing Result Sets

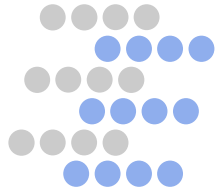


- `mysql_fetch_row` and `mysql_fetch_array` iterate through result set
  - Moves cursor forward one row in result set
  - Returns false when no more rows are available

Initial cursor →
2nd row ⇢
3rd row ⇢
false ⇢

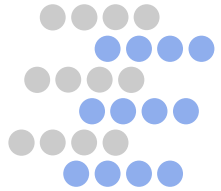
| Name | ID | Salary | Dept ID |
|---|---|---|---|
| Alice | 111 | 100,000 | 3 |
| Bob | 222 | 88,000 | 3 |
| Carole | 333 | 112,000 | 2 |

# Working Example



```
<html><head><title>PHP Test</title></head>
   <body>
   <?php
     print "<h1>Employees</h1>";
     /* Connecting, selecting database */
     $link = mysql_connect("host.eurecom.fr", "scott", "tiger")
       or die("Could not connect");
     mysql_select_db("people_db")
       or die("Could not select database");

     /* Performing SQL query */
     $query = "SELECT * FROM Employees";
     $result = mysql_query($query)
       or die("Query failed");
```

# Working Example (cont'd)



```
  $header_printed = false;
  print "<table border=1>\n";
  while($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
    if(!$header_printed) {
      print "\t<tr>\n";
      foreach ($line as $name => $value)
        print "\t\t<th>$name</th>\n";
      print "\t</tr>\n";
      $header_printed = true;
    }
    print "\t<tr>\n";
    foreach ($line as $name => $value)
      print "\t\t<td>$value</td>\n";
    print "\t</tr>\n";
  }
  print "</table>\n";
  mysql_free_result($result);  // Free result set
  mysql_close($link);  // Close connection
?>
</html>
```