Database Systems DB(H)

Dr Chris Anagnostopoulos

Senior Lecturer in Distributed & Pervasive Computing

General Course Information

- **Course Delivery**
  - Pre-recorded Lectures [Weeks: 1-10]
    - Access Passcode for ALL recordings: DBH2021!
  - Lectures Discussion: Fri 10h00-11h00 @ Zoom (live)
    - Meeting ID: 917 6999 7381; Passcode: DBH
    - Using **sli.do** for Q&A Sessions (event codes will be sent over)

  - 7 Lab Sessions (optional) [Weeks: 1 – 7]
    - Fri 14h00-15h00; 15h00-16h00; 16h00-17h00 (online/in-person sessions)
    - You will be assigned to one *of* them (TBD)
  - 2 Coursework & Revision Sessions (optional) [Weeks: 9 – 10]

  - Office hour: *devoted* slot Fri 11h00 – 11h45 @ Zoom (live)
    - Meeting ID: 945 0177 3709; Passcode: DBOFFICE
- **Assessed Group Work**: 20%
  - Group of size 4; email me your group (members & title) end of JAN
- **Exam:** 80%

Resources

- **Textbooks**
  - *Fundamentals of Database Systems*, Elmasri & Navathe, 7th Edition, Pearson, 2017.
  - Online *access* via UoG Library

  - *Database Systems: The Complete Book*, H. Garcia-Molina, J.D. Ullman and J. Widom, Pearson Education Ltd 2014.
  - Online *access* via UoG Library

- **Course Web Page on Moodle**
  - Updated regularly *as* we go
  - News/information regarding the course will be posted on Moodle
  - Lecture Notes/Recordings/handouts will be provided on Moodle.
  - *Pre/post-session activities* & *formative assessments* are completely **optional**!

Course Contents

- **Part A.1: Relational Design (2 weeks)**
  - Relational Model
  - Functional Dependency Theory
  - Normalization Theory
- **Part A.2: SQL (2 weeks)**
  - SQL basics, Advanced SQL
- **Part B.1: Physical Design & Indexing (~ 3 weeks)**
  - Physical storage on files and disks
  - Indexing, B Trees, Hashing Methods
- **Part B.2: Query Processing & Optimization (~ 3 weeks)**
  - Key *query processing* algorithms
  - Cost-based Query *optimization* and analytics

Relational Design Principles

Systems Fundamentals

Database Fundamentals & Relational Model

Database Systems (H)

Dr Chris Anagnostopoulos

ROADMAP

- *Contextual* Database
  - Database System *definition*

- The *first* Relational Model
  - Dr Edgar Codd's invention 50 years *ago*

- Database Schema *formalism*
- *Integrity* constraints
  - *Superkeys*, *candidate* keys, *foreign* keys.

- *Fundamental* operations and violations
  - *insert*, *delete*, *update*.

Contextual Database

- **Observation:** *human* activity is *data-driven*, i.e., we are *making* decisions, *proceeding* with actions and *reasoning* based on *observed* data.
- **Observation**: we are *limited* in storing *all* data in our *memory* and *recall* them;

- **Idea:** Define a robust *base* that can *store* humongous data, *update* and *delete* data, and *recall* / *search* data *efficiently*;

- A *database* holds data *relevant* to our current *contextual activity*:
  - **Web Search:** a *database* with web page links [Google Database]
  - **Data Mining:** a *database* managing *multidimensional* data for discovering *patterns*, *outliers*, *novel* trends, *prediction* and classification [UCI ML Repository]
  - **Scientific/Medical** databases used for drug discovery, health monitoring and viruses analytics [MEDLINE]
  - **Customer/Retail** databases for customers *profiles* and *preferences*, *products* [Amazon Database]

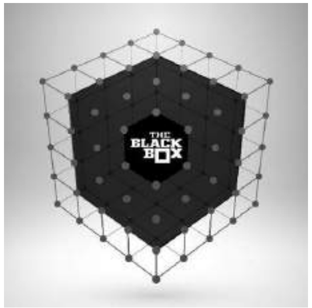
Data Management System

- The *fundamental* functionality is to provide *software* to:
  - **model data: *relational data*** modeling, *object-oriented* data modeling, *first-order logic* data modeling, *description logics* data modeling, *fuzzy logic* modeling…
  - **access data**: *query* for data, *insert*, *delete* and *update*;
  - **analyze data**: *complex* aggregation queries, *function approximation*, histograms, multi-dimensional visualization, outliers detection, …
  - **store (physically) data**: from memory to hard disks;
  - **secure data:** control access to sensitive & confidential data, cipher / encode data;

Data Management System

- **maintain data consistency** in the face of:
  - *failures*, e.g., think of machine crashes due to software bugs, power cuts, disk crashes;
    - *recovery* from failures.

- **optimize data access** to efficiently *retrieve* data
  - *index and hashing data structures*: fast access to data!
  - *optimization* algorithms.

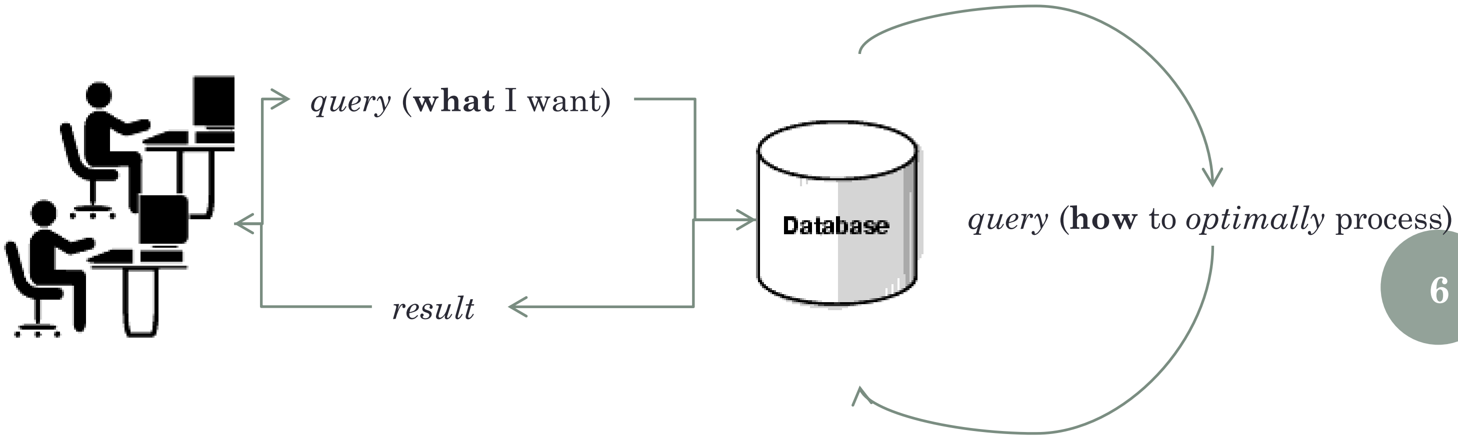


DATA MANAGEMENT SYSTEM

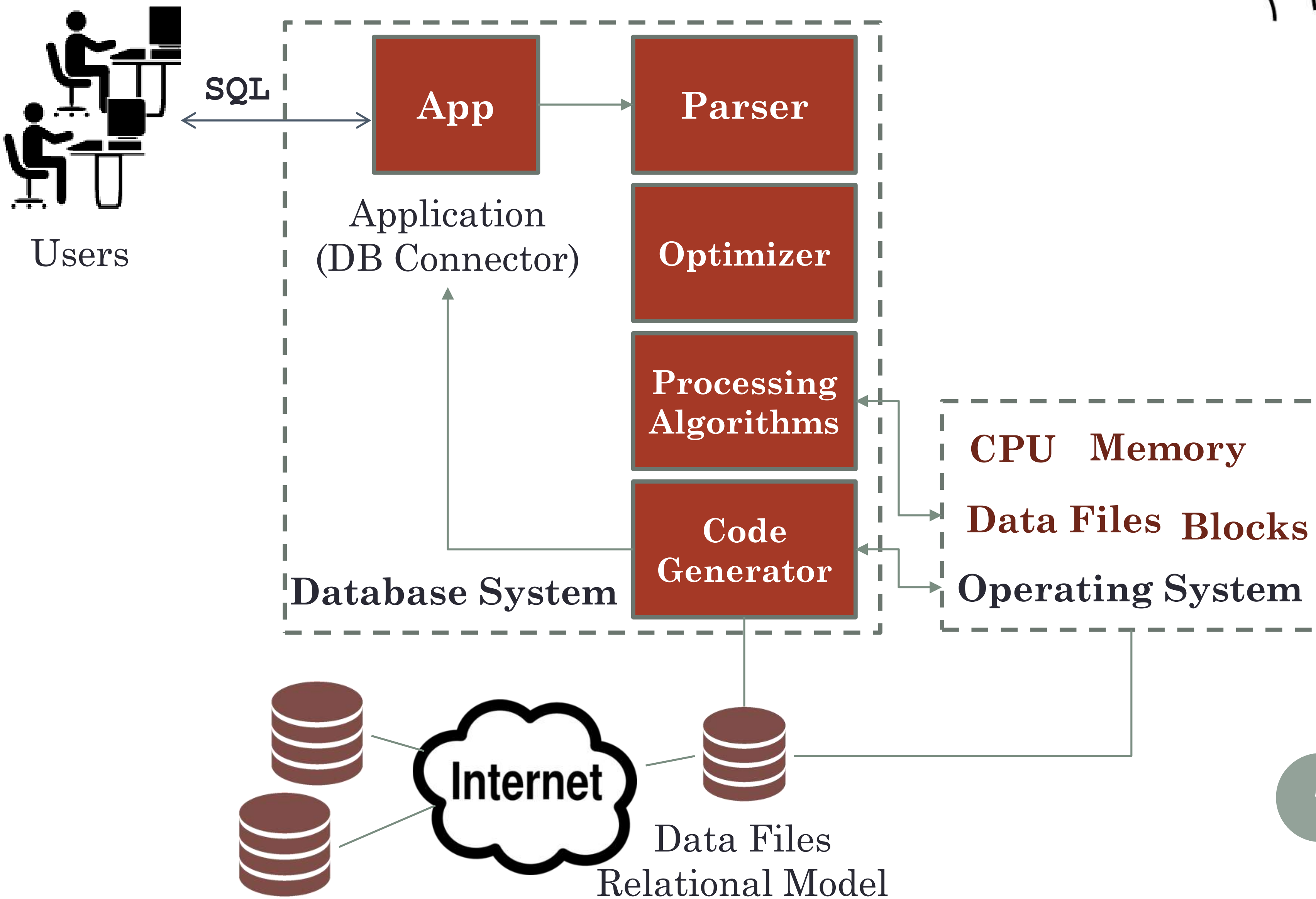
A **box** with an *interface* for users/applications offering the discussed functionality;

- Data Modelling;
- Declarative Programming Language (SQL) to manage & query data

- *Declarative*: we *tell* the database **what** to do and not *how to do* a task.



DATA MANAGEMENT SYSTEM



DATA

- Distinguish three *families* of data:
  - **Structured data**: well-defined data structure, e.g., tables;
    - E.g., **3 Kg**: 3 is *datum* and Kg is *meta-data* for this datum

Kg
3.0
4.1
3.6

  - **Unstructured data**, e.g., web pages, texts, sensor measurements;
    - Less information is provided on interpreting the data

3.0, 4.1, 3.6, 6.7, 8.8, …

  - **Semi-structured data,** e.g., XML or JSON documents
    - Self-descriptive data; they interpret themselves (*medium* entropy)

<data type=real; unit=‘Kg’>3</data>

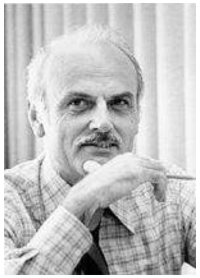
- Modern DMSs manage *all* families of data: *documents*, *graphs*, *multimedia content…*

Conceptual Data Modelling

Peter Chen (1947)

Challenge: *transform* a textual description of a real problem into a set of *concepts* conveying *exactly* the same information.

- **Approach: Entity-Relationship Modeling**
  - *Inventor:* Prof P Chen; **1976**
  - Does not guarantee *optimality* in operations and query executions.



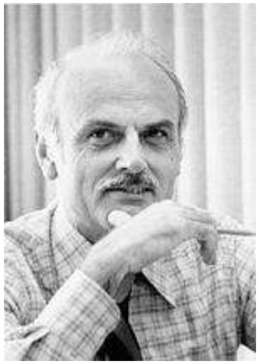
Edgar F. Codd (1923-2003)

- **Approach: Relational Modeling**
  - *Inventor:* Dr Edgar Codd; **1970**
  - *Mathematics-driven*: foundation of relational algebra, set theory, functional dependency theory.
  - Guarantees query optimization

CONCEPTUAL DATA MODEL

- A *mathematical* model for *interpreting* our data
  - **Why mathematical model**: theorems from: *set theory, functional dependency* & *normalization theory*, and *relational algebra.*
  - **Why interpretation**: need to understand the *context*
    - Which *are* the entities? e.g., bank accounts; students; employees …
    - Which *are* the attributes (characteristics) of a data entity? e.g., name; address; ID …
    - Which *are* the relationships between entities? e.g., an employee *works* in a department; a student *attends* many courses?

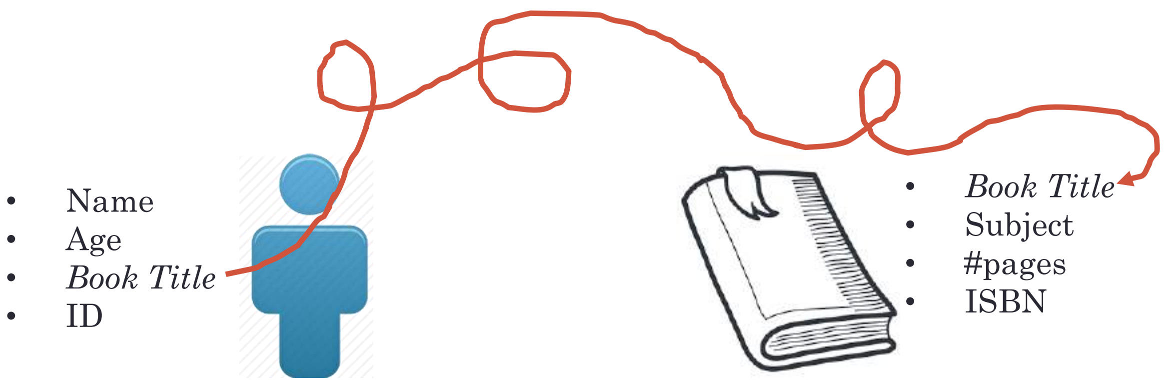
Relational Conceptual Model



Edgar F. Codd (1923-2003)

E.F. Codd; *A Relational Model for Large Shared Data Banks*, Communications of the ACM, June 1970 (ACM Turing Award).

- *Informally*, any entity might *relate with* any other entity *when* they both share *common* attributes.



Relational Model

- Any *entity* and any *relationship* are modelled as a *relation*, which maps to a 2-dimensional table:
  - an *ordered* set of *attributes* (*columns*);
  - a set of *tuples* (*rows*), which represents instances;
  - There exists a *specific* attribute that *uniquely* identifies a tuple in the relation,
    - e.g., *sequential* numbers 1, 2, 3, …, or
    - e.g., *logical* values like the matriculation number/ID.

Attribute 1	Attribute 2	…	Attribute *n*
ID1	Thomas C	…	active
ID2	Carolyn B	…	inactive

Relational Model

Example: being in the *context* of bank accounts:

attribute

Relation: BankAccount

# account	name	balance	status
1234567	'Thomas C'	£1,000	'active'
7654321	'Carolyn B'	£2,300	'inactive'

tuple

unique value per tuple

- **Query:** *attributes of interest* to be retrieved, and *constrained attributes* to filter out irrelevant tuples.
- *Return the names of those customers with active accounts and balance > £500*

```
SELECT name FROM BankAccount
WHERE balance > 500 AND status = 'active'
```

Relational Model: Formalism

- *Schema* of a Relation: $\mathbf{R(A_1, A_2, ..., A_n)}$
  - Relation with name **R** and an *ordered* set of attributes $\mathbf{A_1, A_2, ..., A_n}$
  - Each attribute $\mathbf{A}\boldsymbol{i}$ assumes values in a domain $\mathbf{D}\boldsymbol{i}$, i.e., A$i \in$ Di

e.g., **BankAccount**(account, name, balance, status)

- #account $\in$ N = {1, 2, 3, ...}; *natural numbers* (positive integers)
- name $\in$ Varchar(50); character strings of maximum length 50
- balance $\in$ R; *real numbers*
- status $\in$ {'active', 'inactive'}; finite domain / enumerated type

Relational Model: Formalism

- A tuple t of **R** is an *ordered* set of values corresponding to attributes of **R** satisfying the domain constraints:

$$t = (v_1, v_2, \ldots, v_n),\ v_i \in \mathrm{D}i$$

e.g., t = (1234567, 'Thomas C', 1000, 'active') *or*

t[account] = 1234567, t[name] = 'Thomas C', t[balance] = 1000, t[status]='active'

- An *instance* r(**R**) is a *set* of tuples

$$r(\mathbf{R}) = \{t_1, \mathrm{t}_2, \ldots, \mathrm{t}_m\}\text{: } t_i \text{ is a tuple of } \mathbf{R}$$

Relational Model: Formalism

NULL: represents an *unknown*, or *inapplicable*, or *uncertain*, or *missing* value.

relation *attribute*

STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
tuple	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21

NULL *values with* ***different*** *interpretation*

- **Relational Database Schema**: *set* of relations

$$S = \{\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_k\} \cup \{\text{NULL}\}$$

[A1] Relational Model: Company

- A company is organized into *departments*. Each department has a unique number, and a particular *employee* who manages the department.
  - We keep track of the start date that employee began managing the department.
- A department may have several locations.
- A department controls a number of *projects*, each of which has a unique name, number and a single location.
- The company stores the information about the *employee* (e.g., name, salary, birth date, …), and the unique social security number. An employee is assigned only to one department, but may work on several projects, which are not necessarily controlled by the same department.
  - We keep track of the current number of hours per week that an employee works on each project and their direct supervisor, who is another employee.
- The company keeps track of the *dependent* (e.g., child) of each employee for insurance purposes and the corresponding relationship with the employee (e.g., son, daughter).

[A1] Relational Model: Company

- A **company** is organized into ***departments***. Each department has a unique number, and a particular ***employee*** who manages the department.
  - We keep track of the start date that employee began managing the department.
- A department may have several locations.
- A department controls a number of ***projects***, each of which has a unique name, number and a single location.
- The company stores the information about the *employee* (e.g., name, salary, birth date, …), and the unique social security number. An employee is assigned only to one department, but may work on several projects, which are not necessarily controlled by the same department.
  - We keep track of the current number of hours per week that an employee works on each project and their direct supervisor, who is another employee.
- The company keeps track of the ***dependent*** (e.g., child) of each employee for insurance purposes and the corresponding relationship with the employee (e.g., son, daughter).

[A1] Relational Model: Company

Schema: Company

Entities:

- Department
- Employee
- Project
- Dependent

Relationships:

- An employee *manages* a department.
- A department *may have* several locations.
- A department *controls* a number of projects.
- An employee *is assigned* only to one department,
- An employee *may work on* several projects (for each, store hours per week), which are not necessarily controlled by the same department.
- A department *controls* several projects
- An employee is *supervised* by a supervisor, who is another employee.
- An employee *has several* dependents, each one corresponding to a specific relationship with the employee

Relational Database Schema: Company

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

EMPLOYEE

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

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Task 1:
Mail a Christmas card to John's supervisor's son.

- *Which is his name?*
- *Which relations should be joined for this query?*

Task 2:
How many hours per week does Alicia's supervisor work?

- *In which projects is she involved?*
- *Which relations should be joined for this query?*

Relational Constraints

- *Conditions* that must hold on *all* instances, for *each* relation.

- Distinguish three *fundamental* constraints:
  - **Key constraint** (*unique* tuple identification)
  - **Entity integrity constraint** (keys are *never* null!)
  - **Referential integrity constraint** (*interpretation* of relationships)

KEY CONSTRAINT; E. CODD, 1970

- **Superkey** (**SK**) of relation **R** is a *set* of attributes containing *at least* one attribute that uniquely identifies any tuple.
- For any two *distinct* tuples $t1$ and $t2 \in r(\mathrm{R})$ it holds true the *implication*:

$$\boldsymbol{t1 \neq t2 \rightarrow t1[\mathrm{SK}] \neq t2[\mathrm{SK}]}$$

EMPLOYEE (SSN, Ename, Lname, Bdate, Salary, Dno)

- {SSN, Ename, Bdate}
- {SSN} (*singleton*)
- {SSN, Ename}
- {Ename, Salary}

KEY CONSTRAINT; E. CODD, 1970

- **Candidate Key** is the ***minimal*** superkey; i.e., the set with the *smallest* number of attributes that *uniquely* identify tuples.

Let K = {A1, …, A*k*}, then:

K is **candidate key** $\leftrightarrow$ K' = K \ {A_i} is **not** a SK for any $A_i \in$ K

i.e., the *removal* of *any* attribute from K results in K' that is no longer a superkey.

- {SSN, Ename} \ {Ename} = {SSN} is SK, thus {SSN, Ename} is *not* candidate key
- {SSN, Ename, Lname} \ {Ename, Lname} = {SSN} is SK, thus {SSN, Ename, Lname} is *not* candidate key
- {SSN} is the *minimal* since {SSN}\{SSN} = { } is not a SK

- **Primary Key (PK):** If a relation has *several* candidate keys, **one** is chosen arbitrarily to be **primary key**; the rest candidate keys are called *secondary keys*.

[A2] EXAMPLE

Candidate key is the *minimum sufficient statistic* that discriminates *any* pair of tuples

CAR	LicenseNumber	EngineSerialNumber	Make	Model	Year
	Texas ABC-739	A69352	Ford	Mustang	96
	Florida TVP-347	B43696	Oldsmobile	Cutlass	99
	New York MPO-22	X83554	Oldsmobile	Delta	95
	California 432-TFY	C43742	Mercedes	190-D	93
	California RSK-629	Y82935	Toyota	Camry	98
	Texas RSK-629	U028365	Jaguar	XJS	98

- License number (plate) uniquely identifies a car;
- Engine serial number uniquely identifies a car;

Hence, there are *two* **candidate keys**:

- K1 = {LicenceNumber}
- K2 = {EngineSerialNumber}

A = {LicenceNumber, EngineSerialNumber}

What can we say for A? *superkey* or *candidate key*?

- A is *not* a candidate key since A \{LicenceNumber} = K2.
- Hence, A is a **superkey** & **not a candidate key**

Lesson Learnt: A composite set with *unique* attributes is **not** a candidate key.

Convention: the PK attributes are underlined in the relation schema.

ENTITY INTEGRITY CONSTRAINT; E. CODD, 1970

Principle: Primary Key (PK) cannot be `NULL` in *any* tuple of instance *r*(R).

t**[PK]** ≠ **`NULL`** for *any* tuple *t* in *r*(R)

- If PK has several attributes, `NULL` is *not* allowed in *any* of these attributes
- If {*student-id, course-id*} is a composite PK then:
  student-id ≠ **`NULL`** and *course-id* ≠ **`NULL`**

Note: There might be *non-key* attributes which are not allowed to be `NULL`, *e.g.*, Employee's surname specified by the database *designer*, e.g., *unique value*.

Referential Integrity Constraint ; E. Codd, 1970

Roles: *referencing* relation R1 and *referenced* relation R2.

- There exists an attribute **Foreign Key (FK)** in R1 that either has *exactly* the same value with the **Primary Key (PK)** in R2 or is `NULL`.

$$\boldsymbol{t1[FK]\ \text{references}\ t2[PK] \rightarrow t1[FK] = t2[PK]\ \textit{or}\ t1[FK] = \texttt{NULL}}$$

If $t1[FK]$ = `NULL` then the FK in R1 should **not be a part** of its own primary key (*not* violating the entity integrity constraint).

- **Notation**: *directed* arc from R1.FK to R2.PK

Redesign: Referential Integrity Constraint

- Name
- Age
- *Book Title*
- ID

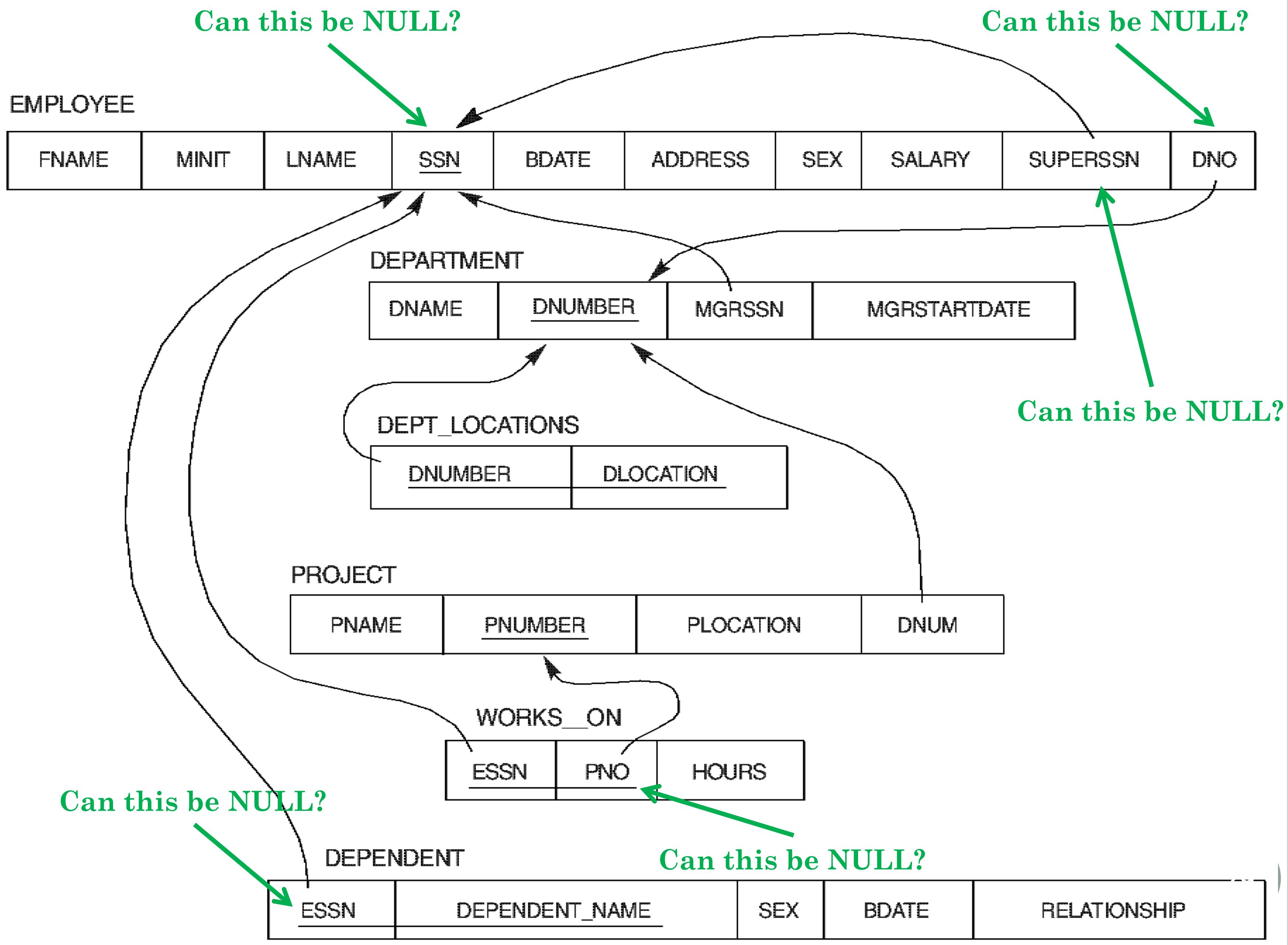
- *Book Title*
- Subject
- #pages
- ISBN

- Name
- Age
- FK **ISBN**
- PK <u>ID</u>

- Title
- Subject
- #pages
- <u>**ISBN**</u> PK

***Referencing* R1**
(Reader)

***Referenced* R2**
(Book)


Can this be NULL?
Can this be NULL?
EMPLOYEE
FNAME
MINIT
LNAME
SSN
BDATE
ADDRESS
SEX
SALARY
SUPERSSN
DNO
DEPARTMENT
DNAME
DNUMBER
MGRSSN
MGRSTARTDATE
Can this be NULL?
DEPT_LOCATIONS
DNUMBER
DLOCATION
PROJECT
PNAME
PNUMBER
PLOCATION
DNUM
WORKS_ON
ESSN
PNO
HOURS
Can this be NULL?
Can this be NULL?
DEPENDENT
ESSN
DEPENDENT_NAME
SEX
BDATE
RELATIONSHIP

So Far…

- **Superkey** is used to *uniquely* identify a tuple in a relation.
- **Lemma:** If K is a **superkey**, then *any superset* of K is a superkey.
- *Proof…left for exercise*

- **Candidate Key** is the *minimal* superkey: i.e., K is a candidate key *if* there is *no* subset of K that is *also* a superkey.

- A relation can have *several different* candidate keys; *one* of them is chosen to be the **Primary Key (PK).**
  - Convention: key attributes are <u>underlined</u> in the relation schema.

- A **Foreign Key (FK)** in a referencing relation should either be `NULL` (if it is not part of the primary key) or have a value of the primary key of the referenced relation.

Functional Dependency & Normalisation Theory

Database Systems (H)
Dr Chris Anagnostopoulos

ROADMAP

- *When* a relational schema is *good* or *bad*…
  - …*judging* the efficiency of a schema.
- **Functional Dependency Theory** for *quantifying* the degree of *goodness*.
- **Normalization Theory** for *transforming* a relational schema into a set of *good* and *efficient* relations.
- Transforming *any* relation to the **Boyce-Codd Normal Form (BCNF)** dealing with *fictitious* tuples.
  - Fundamental **Theorem** in BCNF

Relational Modeling Revisit

Challenge: *develop* a theory and *list* guidelines to *assess* a relational model in terms of (performance metrics):

- *goodness*, i.e., whether the attributes (e.g., PK, FK) form a *good* relation;
- quantify the *degree of goodness*;
- what *pitfalls* exist;
- *efficiency* in insert/delete/update operations;
- *convey as much* information *as possible* minimizing redundancy (minimize repetition of data)…

GUIDELINES FOR A GOOD DESIGN

Guideline 1: The *attributes* of a relation should *make sense*

- Attributes of *different* entities, e.g., students, employees, courses, should **not be in the same** relation.
  - **Objective:** *minimize* the similarity between relations!
- Any relationship between relations should be represented *only* through **Foreign Keys & Primary Keys**
  - e.g., an employee *works* on a specific project for 2 hours:
  - WORKS_ON(SSN **(FK),** PNO**(FK),** Hours).

Guidelines for a Good Design

Guideline 2: Avoid *redundant* tuples (repetition of the *same* information)

- *Impact* of repetition:
  - **storage cost**: replication of tuples *wastes* storage.
  - **inconsistency cost & operation anomalies:**
    - replicas *must be kept* consistent (more checks) during *insertion*, *deletion*, *update* of tuples
  - replicas *result* to consistency anomalies…

Example: Redundancy & Anomaly

EMP_PROJ(SSN, Pnumber, Hours, Ename, Pname, Plocation)

Context: Employee is *working* to a project for specific hours/week.

EMP_PROJ

SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION
123456789	1	32.5	Smith,John B.	ProductX	Bellaire
123456789	2	7.5	Smith,John B.	ProductY	Sugarland
666884444	3	40.0	Narayan,Ramesh K.	ProductZ	Houston
453453453	1	20.0	English,Joyce A.	ProductX	Bellaire
453453453	2	20.0	English,Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong,Franklin T.	ProductY	Sugarland
333445555	3	10.0	Wong,Franklin T.	ProductZ	Houston
333445555	10	10.0	Wong,Franklin T.	Computerization	Stafford
333445555	20	10.0	Wong,Franklin T.	Reorganization	Houston
999887777	30	30.0	Zelaya,Alicia J.	Newbenefits	Stafford
999887777	10	10.0	Zelaya,Alicia J.	Computerization	Stafford
987987987	10	35.0	Jabbar,Ahmad V.	Computerization	Stafford
987987987	30	5.0	Jabbar,Ahmad V.	Newbenefits	Stafford
987654321	30	20.0	Wallace,Jennifer S.	Newbenefits	Stafford
987654321	20	15.0	Wallace,Jennifer S.	Reorganization	Houston
888665555	20	null	Borg,James E.	Reorganization	Houston

EXAMPLE: UPDATE ANOMALY

inefficient

EMP_PROJ(<u>SSN, Pnumber</u>, Hours, Ename, Pname, Plocation)

- **Context:** EMP_PROJ contains 600 employees working on project Pnumber = **2** with Pname = '**ProductY**'.

Update *anomaly*:

- *If* the name of Project 2 changes from '**ProductY**' to '**ProductZ**' *then* we need to enforce *consistency*:

- We should update *all* 600 tuples referring to Project 2,
- *Otherwise*, the relation is *inconsistent*, since some employees would appear to work for a non-existent project;

Example: Delete Anomaly

inefficient

EMP_PROJ(SSN, Pnumber, Hours, Ename, Pname, Plocation)

EMP_PROJ

SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION
123456789	1	32.5	Smith,John B.	ProductX	Bellaire
123456789	2	7.5	Smith,John B.	ProductY	Sugarland
666884444	3	40.0	Narayan,Ramesh K.	ProductZ	Houston
453453453	1	20.0	English,Joyce A.	ProductX	Bellaire
453453453	2	20.0	English,Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong,Franklin T.	ProductY	Sugarland

Context: When Project 2 is deleted, *all* the employees who work on that project will be deleted as well!

- Delete *all* tuples with Pnumber = 2 to ensure consistency…

Guidelines for a Good Design

Guideline 3: Relations should have as a *few* `NULL` values as possible:

- Reasons for `NULL`
  - a value is *not applicable* or *invalid*
  - a value is *unknown* (may exist; who knows?)
  - a value is *known* to exist, but *unavailable*
- **Statistics:** Attributes that are *frequently* NULL should be placed in *separate* relations to avoid wasting storage & reducing uncertainty!

Guidelines for a Good Design

Employee

SSN	Name	…	Phone#1	Phone#2	Phone#3
					NULL
					…
					NULL

Fact: if only 10 out of 1600 employees have *three* contact numbers, then the attribute **Phone#3** contains 99.37% NULL values…(1590 nulls).

Solution: Remove #Phone3 *and* define a *new* relation:

3rdPhone(SSN, Phone#3) **Employee**(SSN, Name, …, Phone#1, Phone#2)

Guidelines for a Good Design

Guideline 4: Design relations to avoid *fictitious* tuples after *join*

EMP_PROJ(SSN, Pnumber, Hours, Ename, Pname, Plocation)

- This is a *bad* design w.r.t. delete/update anomalies.

Idea: *break* relation into *two* smaller (sub)relations that share a *common* attribute (E. Codd's intuition):

  - **R1**(SSN, Ename, Pnumber, Pname, **Plocation**)
  - **R2**(Hours, **Plocation**)

Query: Show the total working hours/week for each employee

Key insight: to *recover* all information, join R1 and R2 on the common attribute **Plocation**.

Fact: this creates tuples which *do not exist* in EMP_PROJ (*fictitious* tuples)!

[A1] Worked Example

DIY

- **R**(SSN, Pname, Plocation, Hours) *breaks into two w.r.t. Plocation*
- **Q**(SSN, Pname, **Plocation**)
- **P**(Hours, **Plocation**)

R Instances:

- r1=(111, 'PR1', Glasgow, 20)
- r2=(222, 'PR1', Glasgow, 10)
- r3=(333, 'PR2', Edinburg, 23)

Q Instances:

- q1=(111, 'PR1', Glasgow)
- q2=(222, 'PR1', Glasgow)
- q3=(333, 'PR2', Edinburgh)

P Instances:

- p1=(20, Glasgow)
- p2=(10, Glasgow)
- p3=(23, Edinburgh)

Join Algorithm:
For each tuple **q** ∈ **Q**
    Join with each tuple **p** ∈ **P**
        iff **q**.Plocation=**p**.Plocation
        Result: **r** = (**p**, **q**)
End For

Example:
q1=(111, 'PR1', Glasgow) *matches* **p1**=(20, Glasgow)
Result: (**p1**, **q1**) = **(111, 'PR1', Glasgow, 20)**

[A1] Worked Example

DIY

R Instances:

- r1=(111, 'PR1', Glasgow, 20)
- r2=(222, 'PR1', Glasgow, 10)
- r3=(333, 'PR2', Edinburg, 23)

Q Instances:

- q1=(111, 'PR1', Glasgow)
- q2=(222, 'PR1', Glasgow)
- q3=(333, 'PR2', Edinburgh)

P Instances:

- p1=(20, Glasgow)
- p2=(10, Glasgow)
- p3=(23, Edinburgh)

q1 *matches* **p1**
Result: **(111, 'PR1', Glasgow, 20)**
q1 *matches* **p2**
Result: **(111, 'PR1', Glasgow, 10)**
q2 *matches* **p1**
Result: **(222, 'PR1', Glasgow, 20)**
q2 *matches* **p2**
Result: **(222, 'PR1', Glasgow, 10)**
q3 *matches* **p3**
Result: **(333, 'PR2', Edinburgh, 23)**

Generate 2 fictitious tuples ☹ in our attempt to avoid anomalies ☺
Lesson Learnt: *Split and join* relations is challenging…*find* the best *splitting* attribute that does not generate fictitious tuples…a Theory is imperative!

Theory of Functional Dependency

Functional Dependency (FD) is a formal metric of the *degree of goodness* of relational schema:

- FD is a *constraint* derived from the *relationship* between attributes

"Given a relation, an attribute X *functionally determines* an attribute Y, *if* a value of X determines a *unique* value for Y.", Codd, 1970

In other words: give me a value of X and I'll *tell* you which is the value of Y in a specific tuple! (X *determines* Y)

Theory of Functional Dependency

- **FD: X → Y** (X *uniquely determines* Y) holds *if* whenever two tuples have the *same* value for **X**, they must have the *same* value for **Y**

Definition: for *any* two tuples $t1$ and $t2$:

If $t1[X] = t2[X]$ **then** $t1[Y] = t2[Y]$

X → Y in **R** specifies a **constraint** on *all* instances, i.e., *principle*.

[A2] Worked Example

EMP_PROJ(SSN, Pnumber, Hours, Ename, Pname, Plocation)

SSN	Pnumber	Hours	Ename	Pname	Plocation
1	5	10	Chris	PX	G12
2	5	30	Stella	PX	G12
1	7	15	Chris	PY	G45

FD1: the social security number (SSN) *determines* the employee name:

$$\textbf{SSN} \rightarrow \textbf{Ename}$$

FD2: the project number *determines* the project name and project location

$$\textbf{Pnumber} \rightarrow \{\textbf{Pname, Plocation}\}$$

FD3: SSN and project number *determine* the hours per week:

$$\{\textbf{SSN, Pnumber}\} \rightarrow \textbf{Hours}$$

FUNCTIONAL DEPENDENCY PRINCIPLES

Lemma 1: If **K** is a **Candidate Key**, then **K** *functionally determines* *all* attributes in relation **R**, i.e.,

$$\textbf{FD: K} \rightarrow \{\textbf{R}\}$$

Lemma 2: William W. Armstrong's inference rules for FDs (1974):

- (*Reflexive*) If $Y \subseteq X$ then $X \rightarrow Y$
  - $X = \{SSN, Ename\}$; $X \rightarrow \{SSN\}$, $X \rightarrow \{Ename\}$
- (*Augmentation*) If $X \rightarrow Y$ then $X \cup \{Z\} \rightarrow Y \cup \{Z\}$
- (*Transitive*) If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

[A3] Worked Example

Consider the relation: **R**(B, O, I, S, Q, D). It *holds* true that:

FD1: $S \rightarrow D$

FD2: $I \rightarrow B$

FD3: $\{I, S\} \rightarrow Q$

FD4: $B \rightarrow O$

Semantics Free!

Q: Which are the candidate keys for relation **R**?

- {I, B}
- {I, S}
- {B, O}
- {S, D}

[A3] Worked Example

In **R**(B, O, I, S, Q, D) *holds* true:

FD1: $S \rightarrow D$

FD2: $I \rightarrow B$

FD3: $\{I, S\} \rightarrow Q$

FD4: $B \rightarrow O$

1. $\{I,S\} \rightarrow Q$ //assertion
2. $I \rightarrow B \rightarrow O$ then $I \rightarrow O$ //transitivity
3. Hence, $I \rightarrow \{B, O\}$
4. $S \rightarrow D$ //assertion
5. Hence, $\{I,S\} \rightarrow \{$**R**$\}$, i.e., candidate key //reasoning

....using *transitivity*:

- **{I, B}**
- **{I, S}**
- {B, O}
- {S, D}

1. $I \rightarrow B$ //assertion
2. $I \rightarrow B \rightarrow O$ then $I \rightarrow O$ //transitivity
3. Hence, $I \rightarrow \{B, O\}$ and $\{I, B\} \rightarrow B \rightarrow O$
4. D , Q, and S are *not* dependent on {I, B}
5. Hence, {I, B} is *not* a candidate key…

FUNCTIONAL DEPENDENCY & NORMALIZATION

Idea: exploit the FDs to specify *which* attributes can become PKs and FKs!

Algorithm:

1. *Assert* which are the FDs among attributes (no *other* semantics required)
2. *Take* a pool and put into *all* the asserted FDs.
3. *Create* a *universal* **big** relation with *all* attributes
4. *Recursively decompose* the big relation based on the FDs into many **smaller** *ones*, such that, when we *re-join* them, it *guarantees* that no information is lost and *re-constructs* the original big relation *without* fictitious tuples.

This is the **normalization process.**

Challenge: *find*, via progressive decomposition, the *basic* relations, which can reconstruct the entire information *efficiently* avoiding *redundancy* and avoiding *fictitious* tuples after their composition.

Theory of Normalization

Progressive decomposition of unsatisfactory (*bad*) relations by breaking up their attributes into *smaller good* relations;

The *degree of decomposition* is referred to as **Normal Form** (NF).

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce-Codd Normal Form (BCNF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)
- Sixth Normal Form (6NF)
- …

SOME DEFINITIONS

- A **prime attribute** is an attribute that belongs to *some* candidate key of the relation
  - e.g., SSN and Pnumber are *prime* attributes.

- A **non-prime attribute** is *not* a prime attribute, i.e., it is not a member of *any* candidate key
  - e.g., Hours, Ename, Pname, Plocation are *non-prime* attributes

EMP_PROJ(SSN, Pnumber, Hours, Ename, Pname, Plocation)

First Normal Form (1NF)

- The domain D*i* of each attribute A*i* in a relation **R** *refers only* to atomic (simple/indivisible) values.

- 1NF *disallows*:
  - *nested* attributes,
  - *multivalued* attributes

Note: a relation in 1NF is expected to have *redundant & repeated* values…

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

1NF

EMP_PROJ

SSN	ENAME	PROJS	
		PNUMBER	HOURS

EMP_PROJ

SSN	ENAME	PNUMBER	HOURS
123456789	Smith,John B.	1	32.5
		2	7.5
666884444	Narayan,Ramesh K.	3	40.0
453453453	English,Joyce A.	1	20.0
		2	20.0
333445555	Wong,Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya,Alicia J.	30	30.0
		10	10.0
987987987	Jabbar,Ahmad V.	10	35.0
		30	5.0
987654321	Wallace,Jennifer S.	30	20.0
		20	15.0
888665555	Borg,James E.	20	null

1NF

SECOND NORMAL FORM (2NF)

verbosity in the primary key **X**

Definition: *full* FD X → Y means that if we remove *any* prime attribute A from the **primary key** X, then:

$$X \setminus \{A\} \nrightarrow Y$$

i.e., X \ {A} **does not** functionally determine Y anymore.

Example:

- {SSN, Pnumber} → Hours is a *full* FD, since neither SSN → Hours nor Pnumber → Hours holds true.
- {SSN, Pnumber} → Ename is **not a full** FD (*partial* dependency), since SSN → Ename holds true;
- i.e., Pnumber does *not* need to be part of the primary key; it is **verbose**...

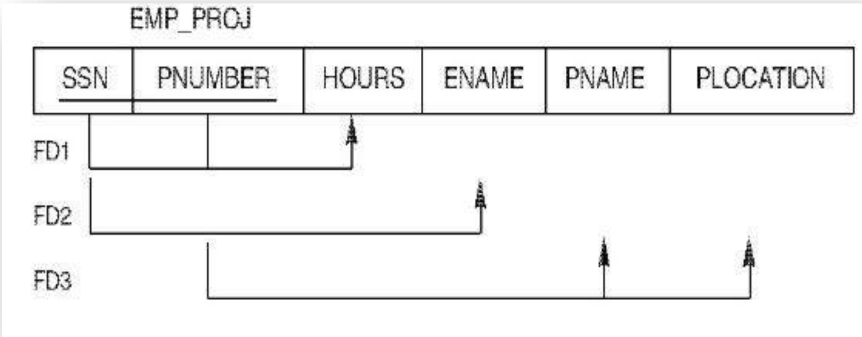
Second Normal Form (2NF)

Definition: A relation **R** is in 2NF if *every* non-prime attribute A in **R** is *fully functionally dependent* on the primary key of **R**.

Target from 1NF to 2NF: *remove* all prime attributes from the primary key, which cause *partial dependencies*.

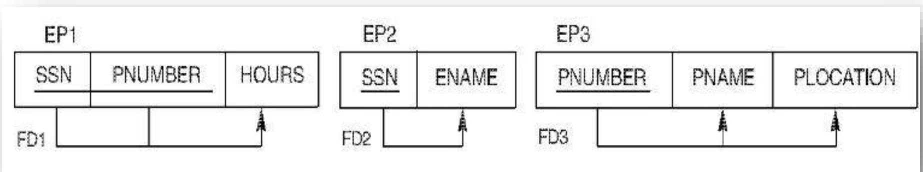
Methodology:

1. Identify *all* the partial FDs in the *original* relation (already in 1NF)
2. For *each* partial FD, create a *new* relation such that *all* non-prime attributes in there are *fully* functionally dependent on the *new* primary key:
   - i.e., the prime attribute in the original relation causing partial FDs.
3. The new relation(s) *will* be in 2NF.



Primary key is: {SSN, Pnumber}

- **FD1:** *full* FD
- **FD2:** Ename (non-prime) depends *only* on SSN and not on Pnumber; *partial* FD
- **FD3:** Pname and Plocation (non-prime) depend only on Pnumber and not on SSN; *partial* FD



2NF

Relation EP1:

- **FD1:** *full* FD

Relation EP2:

- **FD2:** Ename (non-prime) fully depends *only* on SSN

Relation EP3:

- **FD3:** Pname and Plocation (non-prime) fully depend *only* on Pnumber

1NF?
2NF?

Problem Identification

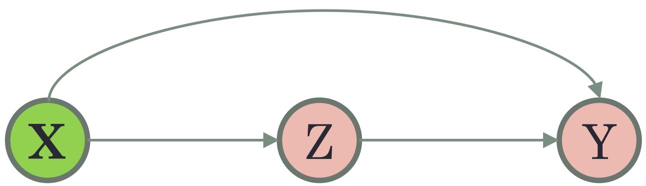
EMP_DEPT(<u>SSN</u>, Ename, Bdate, Address, Dnumber, Dname, Dmgr_Ssn)

<u>SSN</u>	Ename	Bdate	Address	Dnumber	Dname	Dmgr_ssn
1	Chris	1970	A1	3	SoCS	12
2	Stella	1988	A2	3	SoCS	12
3	Philip	2001	A3	3	SoCS	12
4	John	1966	A4	3	SoCS	12
5	Chris	1955	A5	3	SoCS	12
6	Anna	1999	A6	4	Maths	44
7	Thalia	2006	A7	4	Maths	44

Third Normal Form (3NF)

Definition: *transitive* FD means that given a **Primary Key X** and non-prime attributes Z and Y such that: **X** → **Z** and **Z** → **Y**, then the non-prime **Y** is *transitively* dependent on the primary key **X** via the non-prime **Z**.

X → Z **and** Z → Y **then X** → Y.



[<u>CourseID</u>, Lecturer, School]

FD1: CourseID → Lecturer

FD2: Lecturer → School

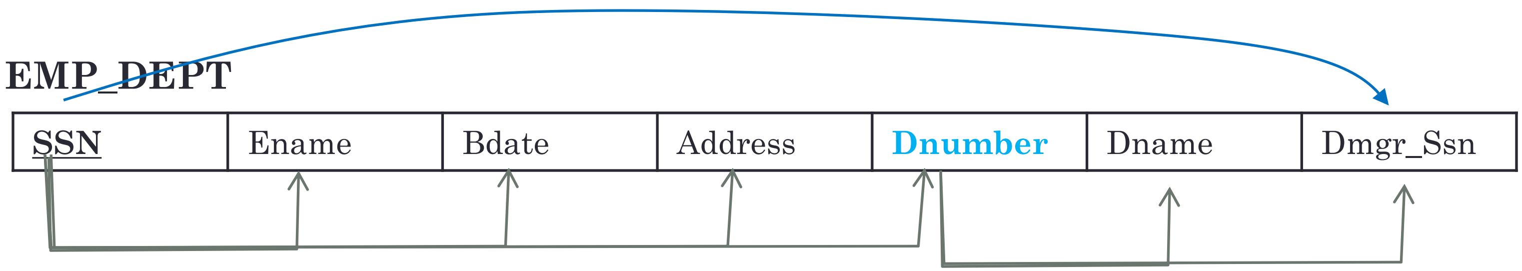
FD3: CourseID → School (*via the non-prime Lecturer*)

Definition: A relation **R** is in 3NF (being *already* in 2NF) if there is *no* non-prime attribute which is *transitively dependent* on the primary key;

- That is, *all* non-prime attributes should be *directly* dependent on the primary key.

THIRD NORMAL FORM (3NF)

EMP_DEPT(<u>SSN</u>, Ename, Bdate, Address, Dnumber, Dname, Dmgr_Ssn)



- **FD1**: SSN → **Dmgr_Ssn** is *transitive*
  - SSN → **Dnumber** and **Dnumber** → **Dmgr_Ssn**
  - Dnumber is *non-prime transitive attribute*
- **FD2: SSN** → Ename is *non-transitive*
  - there is ***no*** attribute Z, where SSN → Z and Z → Ename.
- **FD3: SSN** → Dname is *transitive* FD or not?
- **FD4: SSN** → Address is *transitive* FD or not?

THIRD NORMAL FORM (3NF)

EMP_DEPT

<u>SSN</u>	Ename	Bdate	Address	Dnumber	Dname	Dmgr_Ssn

Methodology: Split the original relation into *two* relations: the *non-prime transitive attribute* (**Dnumber**)

- is the **PK** to the *new* relation
- is the **FK** in the *original* relation referencing to the *new* relation.

Third Normal Form (3NF)

EMP_DEPT

SSN	Ename	Bdate	Address	**Dnumber**	**Dname**	**Dmgr_Ssn**

ED1

SSN	Ename	Bdate	Address	**Dnumber**

ED2

Dnumber	Dname	Dmgr_Ssn

A GOOD SCHEMA NOW…

1NF
2NF
3NF

EMPLOYEE

SSN	Ename	Bdate	Address	Dnumber
1	Chris	1970	A1	3
2	Stella	1988	A2	3
3	Philip	2001	A3	3
4	John	1966	A4	3
5	Chris	1955	A5	3
6	Anna	1999	A6	4
7	Thalia	2006	A7	4

DEPARTMENT

Dnumber	Dname	Dmgr_ssn
3	SoCS	12
4	Maths	44

Generalized Third Normal Form (3NF)

X → **Z** and **Z** → **Y**, with **X** as the **primary key**, we consider this as a **problem** *if only if* Z is **non-prime** attribute.

- Violation of 3NF: when a *non-prime attribute* (Z) determines another *non-prime attribute* (Y) and Y is *transitively* dependent on the prime key (X).

Case: If **Z** is a candidate key, there is **no problem in 3NF!**

EMPLOYEE(<u>SSN</u>, PassportNo**,** Salary)

- **FD1:** SSN → PassportNo
- **FD2:** PassportNo → Salary,
- **FD3:** SSN → Salary is *not* a transitive FD since PassportNo is a prime attribute (**candidate key)**.

Generalized 3NF: Every *non-prime* attribute **A** in relation **R**:

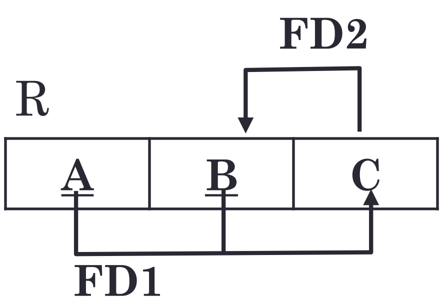
- is *fully* functionally dependent on *every* candidate key in **R**.
- is *non*-transitively dependent on *every* candidate key in **R**.

Ok, *now*, what is happening with the *prime* attributes? BCNF…

Boyce-Codd Normal Form (BCNF)

Idea: *remove all inherent dependencies:* any attribute should be functionally dependent *only* on the Primary Key.

A relation is in BCNF *iff whenever* there exists a **FD: X → A** *then* **X** is a **PK**, i.e., the *left-hand side* should be a PK.



Primary key: {A,B}
Relation is in 3NF: there is *no* transitivity of a *non-prime* attribute to the key, thus, in 3NF.

FD1: {A,B} → C
FD2: C → B; the non-prime attribute C determines B, *however*, C is not a PK.

TEACH

STUDENT	COURSE	INSTRUCTOR
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe

Primary key: {Student, Course}

FD1: {Student, Course} → Instructor
FD2: Instructor → Course /*each instructor teaches *only* one course*/

Is it in 3NF?
- *Yes*, there is no transitive dependency of a *non-prime* attribute on the PK

Is it in BCNF?
- *No*, in FD2 the left-hand side of the dependency (Instructor) is not a PK.

Challenge: How to decompose the relation?

Decomposition in BCNF

- Possibilities for decomposing relation TEACH
  - {student, instructor} and {student, course}
  - {course, instructor } and {course, student}
  - {instructor, course } and {instructor, student}
  - …

  - **Objective**:
    - Can we *reconstruct* TEACH after joining?
    - Which is the *best* decomposition to avoid *fictitious* tuples?
    - Which is the *splitting attribute* in 3NF relations?

Answer?

Decomposition in BCNF

Examine & Join: {course, instructor } and {course, student}

TEACH

STUDENT	COURSE	INSTRUCTOR
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar

R2

STUDENT	COURSE
Narayan	Database
Smith	Database
Smith	Operating Systems

R1

COURSE	INSTRUCTOR
Database	Mark
Database	Navathe
Operating Systems	Ammar

student	course	instructor
Narayan	Database	Mark
Smith	Database	Mark
Narayan	Database	Navathe
Smith	Database	Navathe
Smith	Operating System	Ammar

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

n=256; k=2, then check **32640** split relations!

Combinations n choose k; $n! = 1 \cdot 2 \cdot 3 \cdots n$

BCNF Decomposition Theorem

Theorem 1: Let relation **R** not in BCNF and let **X** → **A** be the FD which causes a violation in BCNF.

Then, the relation **R** should be decomposed into *two* relations:

- **R1** with attributes: **R**\{A} (*all attributes in R apart from A*)
- **R2** with attributes: {X} ∪ {A} *(put together X and A)*

If either **R1** or **R2** is not in BCNF, *repeat* the process.

Proof: see [*].

Experiment: http://www.ict.griffith.edu.au/~jw/normalization/ind.php

[*] C. Zaniolo, ‘A New Normal Form for the Design of Relational Database Schemata’, *ACM Transactions on Database Systems* 7(3):493, Sept. 1982

Theorem Application

- **FD1**: {Student, Course} → Instructor
- **FD2**: Instructor → Course (*violation*)

X → **A** refers to **Instructor** → **Course**, which violates the BCNF.
Hence, **X** *is* Instructor and **A** *is* Course.

Given **R** is TEACH(Student, Instructor, Course).
Split R into *two* relations:

- **R1** with attributes: **R**\{A} = {Student, Course, Instructor} \ {Course} = {Student, Instructor}
- **R1(<u>Student, Instructor</u>)**

- **R2** with attributes: {X} U {A} = {Instructor} U {Course} = {Instructor, Course}
- **R2(<u>Instructor,</u> Course)**

Note: If you join **R1** and **R2** w.r.t. Instructor common attribute then we obtain the *original* TEACH relation with **no** fictitious tuples!

Optimal Decomposition in BCNF

Check: {student, instructor} and {instructor, course}

TEACH

STUDENT	COURSE	INSTRUCTOR
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar

R1

STUDENT	INSTRUCTOR
Narayan	Mark
Smith	Navathe
Smith	Ammar

R2

COURSE	INSTRUCTOR
Database	Mark
Database	Navathe
Operating Systems	Ammar

student	course	instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating System	Ammar

SQL

Database Systems (H)
Dr Chris Anagnostopoulos

ROADMAP

- **Structured Query Language (SQL);**
  - Create a database *schema* & *relations* in SQL;
  - Assign key/integrity/referential *constraints* in SQL;

- **SELECT clause for *selection* queries;**
  - *Multi-sets* and *Sets* in SQL
  - Dealing with NULL values

- **Nested Correlated & Uncorrelated Queries**

Philosophy of the Declarative Language

- Structured Query Language *by* R. Boyce (**1974**).

- SQL is a **declarative** language, i.e.,
  - declare *what to do* rather than *how to do it*
  - different from procedural languages, e.g., Java, Python, C.

- First *official standard*: **SQL-92**
- Latest release: **SQL:2016**…
- **Advice:** *follow* standard SQL to be compliant with *most* of the Data Management Systems ☺

SQL: Database Schema

- Statement: `CREATE SCHEMA`

`CREATE SCHEMA` Company;

Each statement in SQL *ends* with a semicolon ‘;’

SQL: CREATE TABLE

- A *new* relation (*table* is SQL):
  - Specify the *name* of the relation
  - Specify *attributes*, their *types* (domain), *constraints*

`CREATE SCHEMA` **Company**;

`CREATE TABLE` **EMPLOYEE …**;

SQL: Attributes & Domains

- **Numeric data types**
  - Integer numbers: **INT**
  - Floating-point (real) numbers: **REAL** or **DECIMAL(*n*, *m*)**
  - DECIMAL(3,2) has 3 digits; 2 digits after the decimal '.' e.g., 9.99

- **Character/String data types**
  - Fixed length: **CHAR(*n*)**
    - i.e., exactly *n* characters
    - e.g., CHAR(5) has exactly 5 characters like 'Chris'

  - *Variable* length: **VARCHAR(*n*)**
    - i.e., from 0 to *n* characters
    - e.g., VARCHAR(5) has up to 5 characters like 'C', or, 'Ch', or 'Chris'

SQL: Attributes & Domains

- **Bit-string data types (*sequence* of bits: e.g., 0101100)**
  - *Fixed* length: **BIT(*n*)**
  - *Varying* length: **BIT VARYING(*n*)**

- **Boolean data type**
  - Values of `TRUE` or `FALSE` or `NULL`
  - `SQL` is a 3-valued *logic*…(*yes*, *no*, and *maybe*)

- **`DATE` data type**
  - *Ten* positions for YEAR, MONTH, and DAY in the form `YYYY-MM-DD`

- **More,** like **TIMESTAMP, DATE INTERVALS, …**

  Visit: https://www.postgresql.org/docs/9.5/static/datatype.html

SQL: CREATE TABLE

```
CREATE TABLE EMPLOYEE
        ( Fname          VARCHAR(15)        NOT NULL,
          Minit          CHAR,
          Lname          VARCHAR(15)        NOT NULL,
          Ssn            CHAR(9)            NOT NULL,
          Bdate          DATE,
          Address        VARCHAR(30),
          Sex            CHAR,
          Salary         DECIMAL(10,2),
          Super_ssn      CHAR(9),
          Dno            INT                NOT NULL,
        PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
        ( Dname          VARCHAR(15)        NOT NULL,
          Dnumber        INT                NOT NULL,
          Mgr_ssn        CHAR(9)            NOT NULL,
          Mgr_start_date DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber        INT                NOT NULL,
          Dlocation      VARCHAR(15)        NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
```

attributes

domain (type)

constraints

SQL: Value Constraints

- *Default* value of an attribute
  - **DEFAULT** {value}
  - **NULL** is not permitted for a attribute (**NOT NULL**)
  - e.g., DNO **INT NOT NULL DEFAULT 1;**

- **CHECK** clause (*range domain* constraint)
  - e.g., Dnumber **INT NOT NULL CHECK**(Dnumber > 0 **AND** Dnumber < 21);

SQL: Key Constraints

- **Key constraint**: a **primary key value** is unique (no duplicates);
- **Entity Integrity constraint**: a primary key cannot be `NULL`;

- Primary Key Clause:
  - Dnumber **INT NOT NULL, PRIMARY KEY** (Dnumber);

- **UNIQUE** clause, specifies *candidate* keys
  - Dname **VARCHAR(15) NOT NULL, UNIQUE** (Dname);

```
CREATE TABLE DEPARTMENT
        ( Dname              VARCHAR(15)        NOT NULL,
          Dnumber            INT                NOT NULL,
          Mgr_ssn            CHAR(9)            NOT NULL,
          Mgr_start_date     DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
```

SQL: Referential Constraints

- FOREIGN KEY clause in EMPLOYEE
  FOREIGN KEY (Super_SSN) REFERENCES Employee(SSN)

- FOREIGN KEY clause in DEPARTMENT
  FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN)

- *Triggered* actions for **Mgr_SSN, Super_SSN** when **SSN** is updated or deleted:
  - Action: **ON DELETE SET NULL/ DEFAULT/ CASCADE**
  - Action: **ON UPDATE SET NULL/ DEFAULT /CASCADE**

- CASCADE option *propagates* DELETE / UPDATE to *all* referential tuples!
  - *e.g.*, when **SSN** is updated, then all foreign keys *refer* to it should be updated: **ON UPDATE CASCADE**
  - *e.g.*, when **SSN** is deleted, then all foreign keys *refer* to this tuple might be deleted: **ON DELETE CASCADE**

IN-CLASS QUIZ

Q1: What is happening when we *delete* a department?

Q2: What is happening when we *delete* an employee?

```
CREATE TABLE EMPLOYEE
    ( … ,
      Dno            INT            NOT NULL       DEFAULT 1,
    CONSTRAINT EMPPK
      PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
      FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                   ON DELETE SET NULL          ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
      FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                   ON DELETE SET DEFAULT       ON UPDATE CASCADE);
CREATE TABLE DEPARTMENT
    ( … ,
      Mgr_ssn CHAR(9)          NOT NULL       DEFAULT '888665555',
      … ,
    CONSTRAINT DEPTPK
      PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
      UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
      FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                   ON DELETE SET DEFAULT       ON UPDATE CASCADE);
CREATE TABLE DEPT_LOCATIONS
    ( … ,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
                 ON DELETE CASCADE             ON UPDATE CASCADE);
```

SELECT-FROM-WHERE

```
SELECT    <attribute list>
FROM      <table list>
WHERE     <condition>;
```

- Declare *what* to retrieve, i.e., which are the **attributes of interest**
- Declare *from* where to retrieve, i.e., which is the **table/relation**
- Declare with what *condition* to retrieve, i.e., logical statements involving **OR**, **AND**, and/or **NOT**

But, not saying how to *implement* this, e.g.,

- *how* to load the data from disk to memory,
- *how* to *search* and *check* if a tuple satisfies the condition, etc.

SELECT-FROM-WHERE

Query 0: Which are the addresses of employees working in the department 4 *or* their salary is less that 31000.

SELECT Address
FROM EMPLOYEE
WHERE DNO = 4 **OR** Salary < 31000

EMPLOYEE

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

SELECT-FROM-WHERE: JOIN & SELECT

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

SELECT Fname, Lname, Address
FROM EMPLOYEE, DEPARTMENT
WHERE Dname = 'Research' **AND** Dno = Dnumber;

EMPLOYEE

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

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

Fname	Lname	Address
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

```
Q2:  SELECT  Pnumber, Dnum, Lname, Address, Bdate
     FROM    PROJECT, DEPARTMENT, EMPLOYEE
     WHERE   Dnum=Dnumber AND Mgr_ssn=Ssn AND
             Plocation='Stafford';
```

EMPLOYEE

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

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

Pnumber	Dnum	Lname	Address	Bdate
10	4	Wallace	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, Bellaire, TX	1941-06-20

TABLE AS A VARIABLE

//in Java.

```
int e = 5;
int s = 7;
```

//in SQL

```
EMPLOYEE AS e  --AS is the definition operator
EMPLOYEE AS s
```

- A relation might play different *roles* within a query, e.g., employee *might* be a *supervisee* and employee might be a *supervisor* (recursive references)

SELECT …
FROM EMPLOYEE **AS E**, EMPLOYEE **AS S**
WHERE…

Query 3. For each *employee*, retrieve the employee's first and last name and the first and last name of their supervisor.

```
SELECT  E.Fname, E.Lname, S.Fname, S.Lname
FROM    EMPLOYEE AS E, EMPLOYEE AS S
WHERE   E.Super_ssn=S.Ssn;
```

EMPLOYEE E

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

{John Smith, Franklin Wong}

EMPLOYEE S

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

If WHERE is Missing…

R. Cartesius;1648

- Missing WHERE: *no condition* on tuple selection
- If FROM involves *two* or *more* relations, **avoid**; *unreasonable* tuples.
- **Why?** CROSS (*Cartesian*) PRODUCT: *all* possible tuple combinations!

```
SELECT    Ssn
FROM      EMPLOYEE;

SELECT    Ssn, Dname
FROM      EMPLOYEE, DEPARTMENT;
```

Each *tuple* from **EMLOYEE** is *concatenated* with *each* tuple from **DEPARTMENT**…disaster, computationally heavy, and meaningless!

Missing WHERE is Catastrophe

SELECT Ssn, Dname

FROM EMPLOYEE, DEPARTMENT

EMPLOYEE

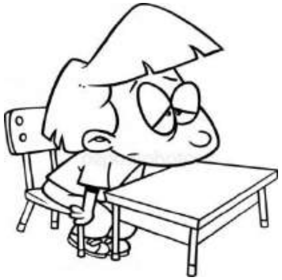
Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer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

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

John…Research
John…Administration
John…HQ
Franklin…Research
Fraklin…Administration
…

USE OF THE ASTERISK



If *bored* listing *all* the attributes, then *use* asterisk (*), i.e., *all* attributes are of interest ☺

```
SELECT    *
FROM      EMPLOYEE
WHERE     Dno=5;
```

Select all the information about **those** employees working at the department 5

```
SELECT    *
FROM      EMPLOYEE, DEPARTMENT
WHERE     Dname='Research' AND Dno=Dnumber;
```

Select all the information (employee and department) from **those** employees working at the department 'Research'

```
SELECT    *
FROM      EMPLOYEE, DEPARTMENT;
```

Select all the information about employees and departments **with no meaning** ☺

TABLES AS MULTI-SETS IN SQL

- **Set**: has only unique elements, e.g., S = {a, b, c}
- **Multiset**: might have duplicates, e.g., M = {a, a, a, b, c, c}
- Operators: `UNION, EXCEPT, INTERSECT`

Query 5: Retrieve the salary of *each* employee, and retrieve *all* the distinct salaries

```
SELECT    Salary
FROM      EMPLOYEE;

SELECT    DISTINCT Salary
FROM      EMPLOYEE;
```

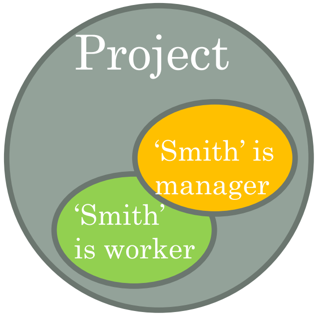
Salary
10000
10000
25000
30000
25000
30000
30000

Salary
10000
25000
30000

In-Class Activity [A2]

List *all* project numbers for projects that involve employees, whose last name is ‘Smith’, *either* as **workers** *or* as **managers** of departments controlling these projects.

Idea: *split* this into *two* sub-queries and then use the *set* UNION operator over the partial results.



In-Class Activity [A2]

Step 1: Retrieve the projects where an employee with surname 'Smith' is *working* on;

Associate **EMLOYEE** with **PROJECT** via **WORKS_ON**

```
( SELECT    DISTINCT Pnumber
  FROM      PROJECT, WORKS_ON, EMPLOYEE
  WHERE     Pnumber=Pno AND Essn=Ssn
            AND Lname='Smith' );
```

In-Class Activity [A2]

Step 2: Retrieve the projects where an employee with surname 'Smith' is a *manager* of the department which controls these project(s);

Associate **EMLOYEE** with **DEPARTMENT** to get the manager, and *then* **DEPARTMENT** with **PROJECT** to get the controlled projects by this department.

```
( SELECT    DISTINCT Pnumber
  FROM      PROJECT, DEPARTMENT, EMPLOYEE
  WHERE     Dnum=Dnumber AND Mgr_ssn=Ssn
            AND Lname='Smith' )
```

IN-CLASS ACTIVITY [A2]

Step 3: `UNION` over the two *sets* of project numbers:

```
(SELECT   DISTINCT Pnumber
 FROM     PROJECT, DEPARTMENT, EMPLOYEE
 WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn
          AND Lname='Smith' )
 UNION
( SELECT  DISTINCT Pnumber
 FROM     PROJECT, WORKS_ON, EMPLOYEE
 WHERE    Pnumber=Pno AND Essn=Ssn
          AND Lname='Smith' );
```

THREE-VALUED LOGIC

SQL is a three-valued logic: `TRUE (1)`, `FALSE (0)` and `UNKNOWN(0.5)`

Recall: Each NULL value is *different* from any other NULL value!

Principle: *Any* value compared with `NULL` evaluates to `UNKNOWN`

Example:

WHERE Address = `NULL` …evaluates to `UNKNOWN;`

WHERE Address <> `NULL` …evaluates to `UNKNOWN;`

WHERE `NULL` = `NULL` …evaluates to `UNKNOWN`

Always adopt: **IS NULL** or **IS NOT NULL**

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

min

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

max

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

1-x

Comparison Involving NULL

Query 6: Retrieve the first and last names of *all* employees who do not have supervisors.

```
SELECT  Fname, Lname
FROM    EMPLOYEE
WHERE   Super_ssn IS NULL
```

```
SELECT  Fname, Lname
FROM    EMPLOYEE
WHERE   Super_ssn = NULL
```

it produces *no* tuples!
Hence, *wrong* reasoning!
Why?

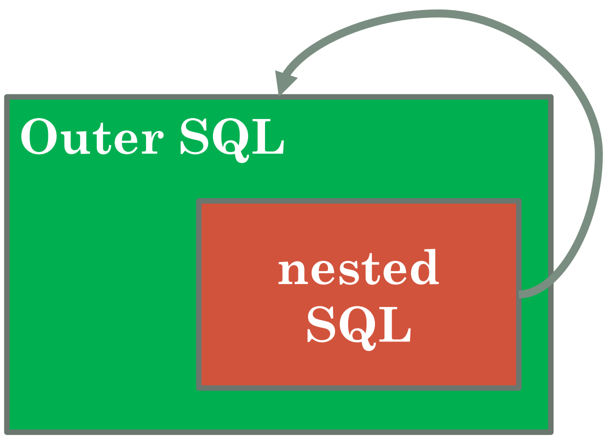
NESTED (INNER) QUERY

Nested query is a query *within* another (*outer*) query;

- SELECT-FROM-WHERE block *within* another outer WHERE clause.
- Nested query's *output* is *input* to outer's WHERE via: **IN**, **ALL**, **EXISTS**

- **Nested Uncorrelated Query**: *first* execute the nested query, and *then* execute the outer query using inner's output.

- **Correlated Query**: for *each* tuple of the outer query, we execute the nested query.



WHERE <*comparison*>

Nested Uncorrelated Query: Operator IN

IN: checks whether a value belongs to the inner's output *set* (or *multiset*), i.e., $v \in S$

Query 7: Show the SSN of those employees who work in the projects with number: either 1, or 2, or 3.

```
SELECT  Essn
FROM    WORKS_ON
WHERE   PNO IN (1, 2, 3);
```

- if PNO = 1 then PNO **IN** (1, 2, 3) evaluates to `TRUE`
- if PNO = 4 then PNO **IN** (1, 2, 3) evaluates to `FALSE`

Nested Uncorrelated Query: Operator IN

Query 8: Show the names of those employees who work in the department 'Research'.

```
SELECT  FNAME
FROM    EMPLOYEE
WHERE   DNO IN ( SELECT  DNUMBER
                 FROM    DEPARTMENT
                 WHERE   DNAME = 'Research');
```

Evaluates to **5**

```
SELECT  FNAME
FROM    EMPLOYEE
WHERE   DNO IN ( 5 );
```

NESTED UNCORRELATED QUERY: OPERATOR ALL

ALL: compares a value with *all* the values from the inner's output *set* using >, >=, <, <=, =, <>

Query 9: Show the last and first names of those employees whose salary is *greater* than the salaries of *all* employees in Department 5.

```
SELECT   Lname, Fname
FROM     EMPLOYEE
WHERE    Salary > ALL   ( SELECT   Salary
                          FROM     EMPLOYEE
                          WHERE    Dno=5 );
```

First, find *all* salaries from employees in Department 5;

Nested Correlated Query

For **each** *tuple* of the *outer* query, we execute the inner query!

Relation as a variable: **global scope** (*outer*) and **local scope** (*inner*).

Query 10: Retrieve the name of *each* employee who has a dependent with the same first name as that employee.

For each *outer* employee E, retrieve the dependents D and check!

```
SELECT E.FNAME, E.LNAME
FROM   EMPLOYEE AS E
WHERE  E.SSN IN (  SELECT D.ESSN
                   FROM   DEPENDENT AS D
                   WHERE  E.FNAME = D.DEPENENT_NAME)
```

NESTED CORRELATED QUERY

Lemma 1: Correlated queries using **IN** are collapsed into one *single* block.

Query 11: Retrieve the name of *each* employee who has a dependent with the same first name as that employee.

```
SELECT E.Fname, E.Lname
FROM   EMPLOYEE AS E, DEPENDENT AS D
WHERE  E.Ssn=D.Essn
   AND  E.Fname=D.Dependent_name;
```

Nested Correlated Query: Exists

EXISTS: checks *whether* the inner's output is an *empty* set or *not*, and returns FALSE or TRUE, respectively, e.g., $S = \{\}$ or $S \neq \{\}$

Opposite: NOT EXISTS

```
SELECT E.Fname, E.Lname
FROM   EMPLOYEE AS E
WHERE  EXISTS
(SELECT * FROM DEPARTMENT AS D WHERE  E.DNO = D.DNUMBER)
```

- Checks if a *given* employee is working at *some* department.
- Reason about `E.DNO` being `NULL`.

Nested Correlated Query: Exists

```
SELECT E.Fname, E.Lname
FROM   EMPLOYEE AS E
WHERE  EXISTS
      (SELECT *
       FROM   DEPARTMENT AS D
       WHERE  E.DNO = D.DNUMBER AND D.DNAME = 'Research')
```

- Describe this…

In-Class Quiz

```
SELECT E.Fname, E.Lname
FROM   EMPLOYEE AS E
WHERE EXISTS
 (SELECT * FROM DEPENDENT AS P WHERE E.SSN = P.Essn)
  AND EXISTS
 (SELECT  * FROM DEPARTMENT AS D WHERE E.SSN = D.Mgr_SSN)
```

Checks if a *given* employee:

- has at least a dependent **and**
- manages a department, i.e., there *exists* a department, which is managed by that employee.

WORKED EXAMPLE

STUDENT (Name, StudentID, Class)
COURSE (Name, CourseID, Credits, School)
GRADES (StudentID, CourseID, Grade)

/*Grade: {‘A’, ‘B’, ‘C’, ‘D’, ‘E’}*/

Task: Retrieve the names of *all* students who have a Grade of ‘A’ in *all* of their courses (‘distinction’ students)

Worked Example

STUDENT (Name, StudentID, Class)
COURSE (Name, CourseID, Credits, School)
GRADES (StudentID, CourseID, Grade)

/*Grade: {'A', 'B', 'C', 'D', 'E'}*/

```
SELECT S.Name
FROM   STUDENT S
WHERE  NOT EXISTS
      (SELECT * FROM GRADES G
       WHERE G.StudentID = S.StudentID
             AND G.Grade <> 'A'
       )
```

There does *not* exist a grade which is *not* 'A', i.e., *all* grades are 'A'

Advanced SQL & Analytics

Database Systems (H)
Dr Chris Anagnostopoulos

ROADMAP

- **Join Query**
  - Dealing with `NULL` FKs

- **Analytics Query**
  - Complex Un/Correlated Query *using* Aggregation Functions over *Groups* of Tuples;
  - **Objective:** Extract *knowledge* from tuples and *not* just retrieving tuples…

- **Modification Query**

INNER JOIN

- **INNER JOIN** *matches* tuples using FK and PK (**THETA-JOIN**).

- The matching operator is, usually, the equality ‘ = ’, thus, it is referred to as **EQUIJOIN**: **R1.PK = R2.FK**

Query 0: Show the employees who are working in the department ‘Research’.

```
SELECT    Fname, Lname, Address
FROM      (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
WHERE     Dname=‘Research’;
```

Inner Join

SELECT Fname, Lname, Address

FROM EMPLOYEE, DEPARTMENT

WHERE Dname = ‘Research’ %selection condition

AND DNO = DNUMBER; %equi-join condition

Note: Join and selection conditions are both in the WHERE clause

INNER AND OUTER JOINS

- **INNER JOIN** (versus **OUTER JOIN**)
  - A tuple is retrieved *if and only if* there exists a matching tuple;
  - i.e., FK is **not** NULL.

- **LEFT OUTER JOIN (LR** LEFT OUTER JOIN **RR)**
  - Every tuple in the *left* relation LR *must* appear in result
  - If no matching tuple exists, just add NULL values for attributes of right relation RR

- **RIGHT OUTER JOIN (LR** RIGHT OUTER JOIN **RR)**
  - Every tuple in the *right* relation RR *must* appear in result
  - If no matching tuple exists, just add NULL values for attributes of left relation LR

LEFT OUTER JOIN

Query 1: Show the last name of an employee and the last name of their supervisor, *if there exists*!

SELECT E.Lname, S.Lname
FROM (EMPLOYEE **AS** E **LEFT OUTER JOIN** EMPLOYEE **AS** S
ON E.Super_SSN = S.SSN)

E.Lname	S.Lname
Smith	Wong
Borg	NULL
Franklin	Jennifer

SELECT E.Lname, S.Lname
FROM EMPLOYEE **AS** E, EMPLOYEE **AS** S
WHERE E.Super_SSN = S.SSN

In-Class Quiz

SELECT E.Fname, E.Minit, E.Lname, D.Dname
FROM EMPLOYEE **AS** E **LEFT OUTER JOIN** DEPARTMENT **AS** D
ON E.SSN = D.MGR_SSN

What are we expecting? What knowledge do we extract?

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

Aggregate Function

Aggregation function: statistical summary/value over group of tuples.

- *Built-in* aggregation functions over attribute **X**:
  - COUNT(*): How many employees are working in Dept 5?
  - SUM (**X**): Sum up all salaries of employees in Dept 5.
  - MAX(**X**) / MIN (**X**): Who is the youngest employee in Dept. 5?
  - AVG(**X**): Average salary of employees in Dept. 5
  - CORR(**X**, **Y**): Correlation between Age and Salary of employees in Dept. 5
  - …

Note: NULL values are *discarded* apart from **COUNT(*).**

Note: Define our *own* function,
  - e.g., *calculate* the GPA as a weighted sum of grades;
  - e.g., *calculate* the Euclidean distance between two geo-locations in Location-base Services applications…

Aggregate Function

Query 2: Show the maximum, minimum and average salary of those employees who work in Dept. 5.

SELECT	**MAX** (Salary) **AS** Highest_Sal,
	MIN (Salary) **AS** Lowest_Sal,
	AVG (Salary) **AS** Average_Sal
FROM	EMPLOYEE
WHERE	DNO = 5;

Analytics: Grouping Tuples

KNOWLEDGE

- *Partition* a relation into *groups* based on grouping attribute, *i.e.*, clustering tuples having the *same* value in the grouping attribute.

```
GROUP BY {grouping attribute}
```

Which is the grouping attribute and the expected number of groups?

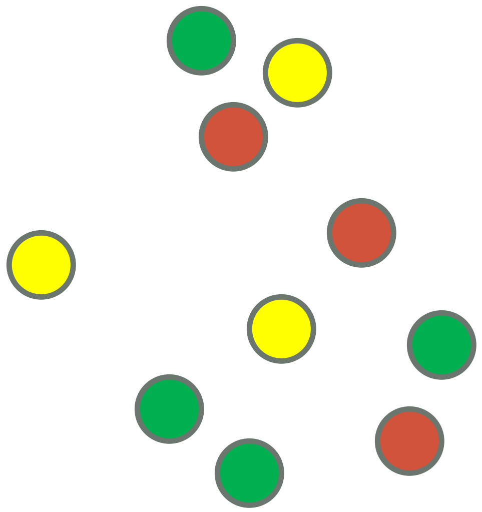
- 1: *Group of employees with the same last-name*
- 2: *Group of employees working in the same department*
- 3: *Group of dependents of the same employee*
- 4: *Group of employees with the same salary…*

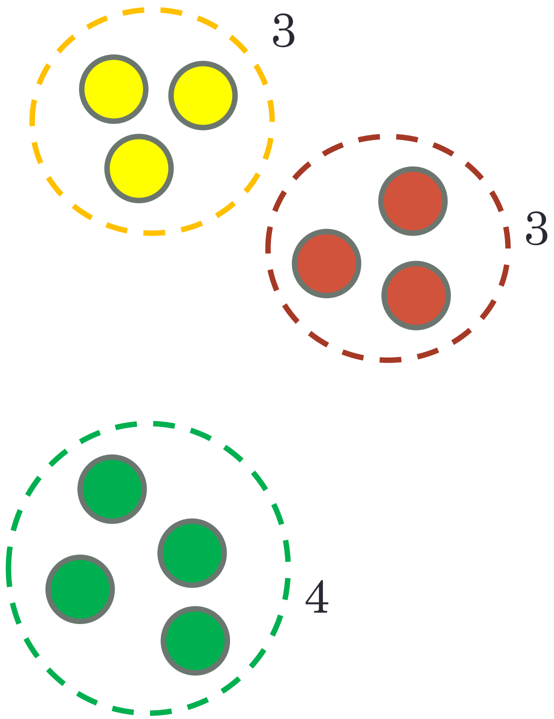
- *Then*, we apply aggregation *functions* to **each group**.

Analytics: Grouping Tuples

VISUALIZATION

aggregation function *f*, e.g., COUNT(*)





#tuples (whole)

#tuples per group

In-Class Example: GROUP BY

Query 3: Show the number of employees per department & average salary per department.

```
SELECT      DNO, COUNT (*), AVG (Salary)
FROM        EMPLOYEE
GROUP BY    DNO;
```

Step 1: Partition EMPLOYEE *into* separate groups w.r.t. department.

Step 2: For *each* group, calculate its *cardinality* and average salary.

Note 1: The grouping attribute *must appear* in SELECT

Note 2: If the **grouping** attribute has NULL values, then a *separate* group is created for the NULL value.

Actual Analytics Query is: *Which is the most populated department?*

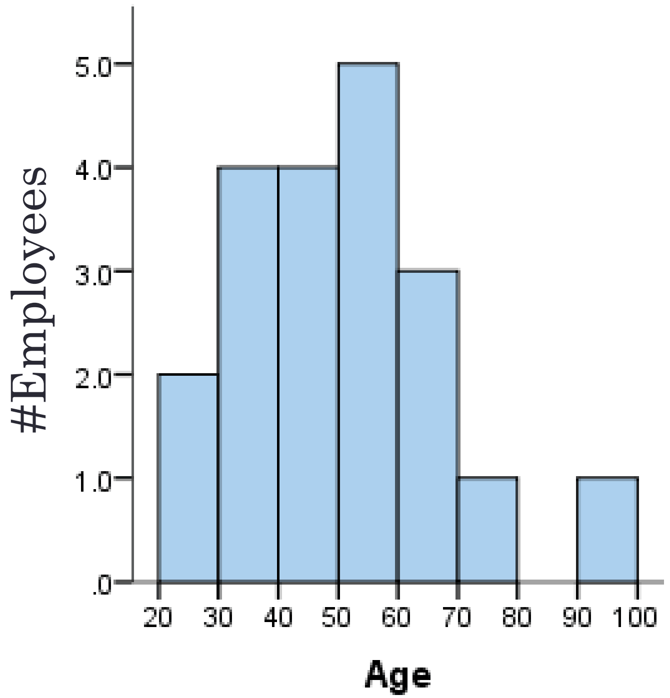
Fname	Minit	Lname	Ssn	. . .	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	. . .	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

In-Class Example: Histogram

Query 4: Show the number of employees *per* age.

```
SELECT      E.AGE, COUNT (*)
FROM        EMPLOYEE AS E
GROUP BY    E.AGE;
```



Step 1: Partition EMPLOYEE w.r.t. age.

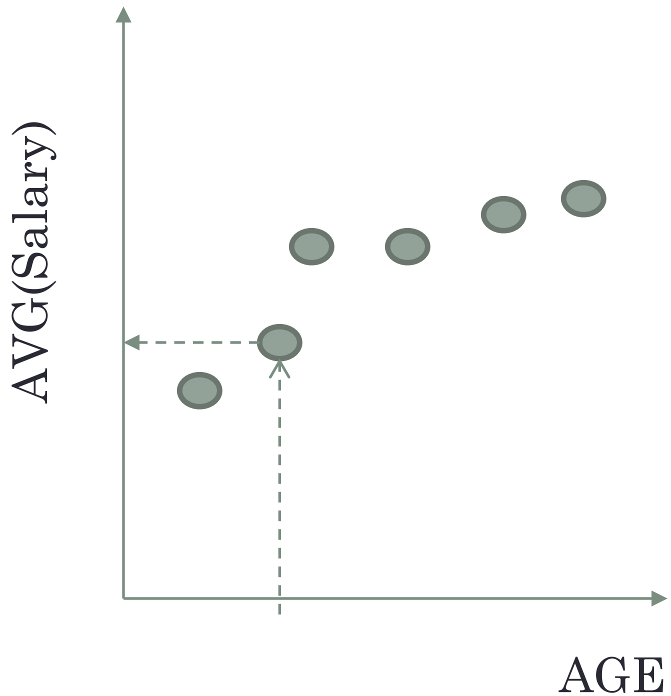
Step 2: For *each* group, calculate its *cardinality*.

Note: Use this analytics to approximate the *histogram* of AGE, i.e., how the AGE is distributed over the tuples…

In-Class Example: Regression Analytics

Query 5: How is the average salary of employees distributed along the age?

```
SELECT      E.AGE, AVG(E.Salary)
FROM        EMPLOYEE AS E
GROUP BY    E.AGE;
```



Note: Use this analytics to *approximate* the *dependency* of Salary *on* AGE.

Note: *Predictive Analytics*: given a *age*, predict the expected *salary*

In-Class Example

Query 6: How many employees are working in *each* project? Include the project name at the results.

```
SELECT      P.PNAME, COUNT (*)   //employees per project
FROM        PROJECT AS P, WORKS_ON AS W
WHERE       P.PNUMBER = W.PNO
GROUP BY    P.PNAME;
```

Step 1: Associate a PROJECT with the WORKS_ON

Step 2: Group *associated* tuples together w.r.t. **PNAME**

Step 3: For *each* group, *count* the number of tuples (employees)

Pname	Pnumber	. . .	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	. . .	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

2 members

3 members

2 members

3 members

3 members

3 members

PNAME	COUNT(*)
Product X	2
Product Y	3
Product Z	2

[A1] Worked Example

Task 1: Which is the average salary of employees *per* department? Include the department name at the results.

EMPLOYEE

SSN	Salary	DNO
1	£30K	1
2	£60K	1
3	£20K	2
4	$25K	3

DEPARTMENT

DNUMBER	DNAME
1	D1
2	D2
3	D3

DNAME	AVG(Salary)
D1	£45K
D2	£20K
D3	£25K

[A1] Worked Example

Task 1: Which is the average salary of employees *per* department? Include the department name at the results.

```
SELECT      D.DNAME, AVG (E.SALARY) //avg salary/dept
FROM        DEPARTMENT AS D, EMPLOYEE AS E
WHERE       D.DNUMBER = E.DNO
GROUP BY    D.DNAME;
```

Step 1: Associate EMPLOYEE with DEPARTMENT

Step 2: Group *associated* tuples together w.r.t. **DNAME**

Step 3: For *each* group, *take* the average of salaries

GROUP BY & HAVING

HAVING: *condition* to select/reject a group *after* grouping!

Query 7: Show the number of employees per project *only* from those projects with *more than* 2 employees. Include the project name in the results.

PROJECT

PName	PNO
GLA	P1
EDI	P2

WORKS_ON

ESSN	PNO
1	P1
2	P1
3	P2
4	P1

PName	COUNT(*)
GLA	3

```
SELECT      P.PNAME, COUNT (*)          3
FROM        PROJECT AS P, WORKS_ON AS W
WHERE       P.PNUMBER = W.PNO           1
GROUP BY    P.PNAME                     2
HAVING      COUNT(*) > 2                4
```

order

In-Class Example

Task: Who are the employees (SSN, last name) with *more* than two dependents.

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

In-Class Example

Task: Who are the employees (SSN, last name) with *more* than two dependents.

Idea: Group dependents of the *same* employee; count.

```
SELECT      E.SSN, E.LNAME, COUNT (*)
FROM        DEPENDENT AS D, EMPLOYEE AS E
WHERE       E.SSN = D.ESSN
GROUP BY    E.SSN
HAVING      COUNT(*) > 2
```

[A2] Worked Example

Task: Who are the managers (last names) of those departments with *more* than 100 employees?

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

[A2] Worked Example

Task: Which are the managers (last names) of those departments with *more* than 100 employees?

```
SELECT M.LNAME
FROM    EMPLOYEE M, DEPARTMENT P
WHERE  M.SSN = P.MGR_SSN
AND     P.DNUMBER IN (
          SELECT       E.DNO
          FROM         EMPLOYEE AS E
          GROUP BY     E.DNO
          HAVING       COUNT(*) > 100);
```

Tricky Analytics Query

Query 8: For *each* department with *more* than 5 employees, *tell me* how many of them are making more than £40K.

```
SELECT      DNO, COUNT (*)
FROM        EMPLOYEE
WHERE       Salary > 40000
GROUP BY    DNO
HAVING      COUNT (*) > 5;
```

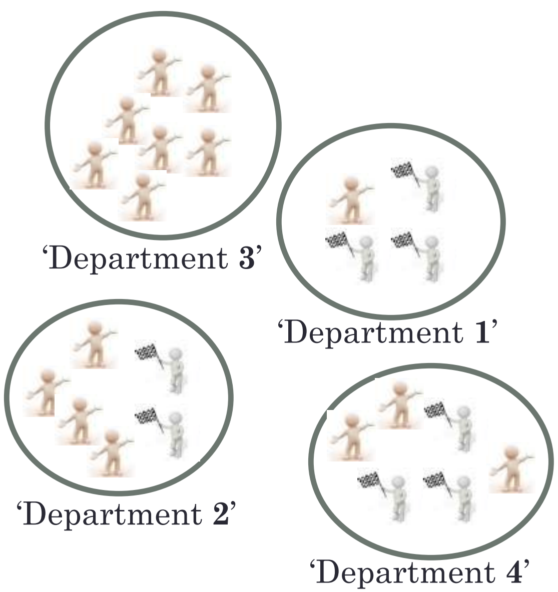
Step 1: Identify the departments with more than 5 employees.

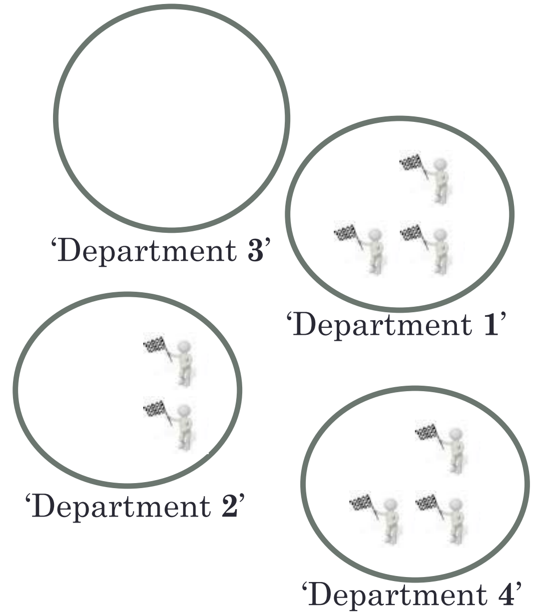
Step 2: Check if employees of those departments earn more than £40K; count

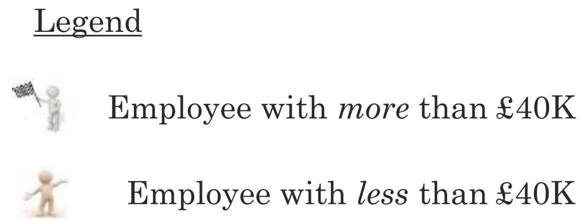
But: WHERE *filters* out employees with Salary <= £40K *before* grouping…thus, the group sizes (employees per department) are not correct…

Tricky Analytics Query

```
SELECT      DNO, COUNT (*)
FROM        EMPLOYEE
WHERE       Salary > 40000
GROUP BY    DNO
HAVING      COUNT (*) > 5;
```



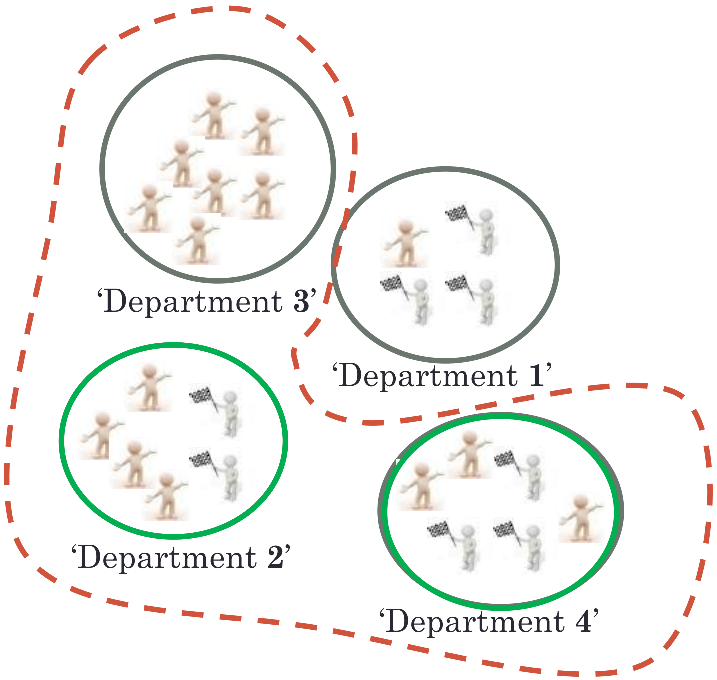




TRICKY ANALYTICS QUERY

Correct: we want to count the *total* number of employees whose salaries exceed £40K in *those* departments with *more than five* employees.

DNO	COUNT(*)
2	**2**
4	**3**



Legend

 Employee with *more* than £40K

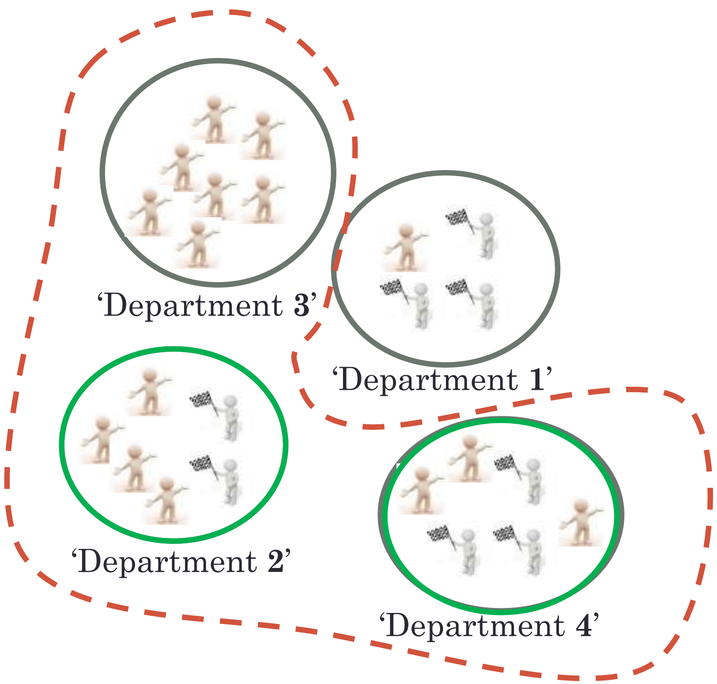
 Employee with *less* than £40K

```sql
SELECT DNO, COUNT(*)
FROM   EMPLOYEE
WHERE Salary > 40000 AND DNO IN
        (SELECT     A.DNO
         FROM       EMPLOYEE A
         GROUP BY  A.DNO
         HAVING COUNT(*) > 5)
GROUP BY DNO
```

Second, for *each* department, check if members earn more than £40K.

First, find the *departments* with more than 5 employees

Then, group and count the £40K-employees per department



[A3] Worked Example

EMPLOYEE(SSN, …, DNO)

Task: Show the department(s) with the maximum number of employees.

Note: It might be the case that *more* than one department has the maximum number of employees.

[A3] Worked Example

EMPLOYEE(<u>SSN</u>, …, DNO)

Task: Show the department(s) with the maximum number of employees. It might be the case that *more* than one department has the maximum number of employees.

```
SELECT      DNO, COUNT(*)
FROM        EMPLOYEE
GROUP BY    DNO
HAVING      COUNT(*) = (SELECT MAX(A.members)
                        FROM
                        (SELECT D.DNO, COUNT(*) AS members
                         FROM    EMPLOYEE D
                         GROUP BY D.DNO) AS A
                        );
```

SQL: INSERT

Key, integrity and referential constraints are *automatically* enforced!

```
INSERT INTO EMPLOYEE (SSN, FNAME, LNAME, …)
VALUES      ('1234567', 'Chris', 'McReader', …)
```

SQL: DELETE

Get a *relation* and **include** a WHERE to specify the tuple(s) to be deleted:

```
DELETE FROM    EMPLOYEE
WHERE          Lname='Brown';

DELETE FROM    EMPLOYEE
WHERE          Ssn='123456789';

DELETE FROM    EMPLOYEE
WHERE          Dno=5;

DELETE FROM    EMPLOYEE;
```

- **Note:** A missing WHERE specifies that *all* tuples in the relation are to be deleted; the table then becomes an **empty table**.

- **Referential Integrity:** tuples are deleted from only *one* table at a time *unless* ON DELETE CASCADE is specified on a constraint!

SQL: UPDATE

Modify attribute values of tuples, which satisfy WHERE *before modification*!

Query 9: Change the *location* and *controlling department number* of project number 10 to 'Bellaire' and 5, respectively.

```
UPDATE    PROJECT
SET       Plocation = 'Bellaire', Dnum = 5
WHERE     Pnumber = 10
```

Referential Integrity: tuples are updated from only *one* table at a time *unless* ON UPDATE CASCADE is specified on a constraint!

In-Class Example

Task: Give *all* employees in the department 5 a 10% raise in salary.

```
UPDATE    EMPLOYEE
SET       SALARY = SALARY *1.1
WHERE     DNO = 5
```

Modified SALARY depends on the original SALARY in each tuple:

- **SALARY** on the *right of* = refers to the **old** salary *before modification*
- **SALARY** on the *left of* = refers to the **new** salary *after modification*

DROP

- DROP is used to *drop* named schema elements, such as *tables*, *domains*, or *constraint*

Example: `DROP SCHEMA COMPANY CASCADE;`

It *removes* the schema and *all* its elements including tables, constraints, etc.

Example: `DROP TABLE EMPLOYEE;`

It *drops* the existing table `EMPLOYEE` and *all* of its tuples.

Example: `DROP TABLE EMPLOYEE CASCADE CONSTRAINTS;`

It *drops* the existing table `EMPLOYEE`, all of its tuples and *drop* the FOREIGN KEY constraints of the tables referring to `EMPLOYEE` (*but* not those tables).

ALTER

- ALTER includes:
  - *Adding* or *dropping* a column (attribute)
  - *Changing* a column definition
  - *Adding* or *dropping* table constraints

Example: ALTER TABLE COMPANY.EMPLOYEE **ADD COLUMN Job VARCHAR(12);**

Example: ALTER TABLE COMPANY.EMPLOYEE **DROP CONSTRAINT EMPSUPERFK;**

Example: ALTER TABLE COMPANY.DEPARTMENT **ADD CONSTRAINT NEW_UNIQUE UNIQUE (Dname);**

Example: ALTER TABLE COMPANY.EMPLOYEE **DROP COLUMN Address;**

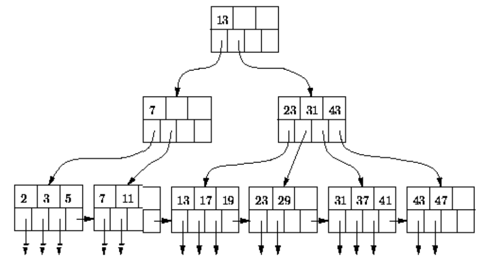
Example: ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN Mgr_ssn DROP DEFAULT;**

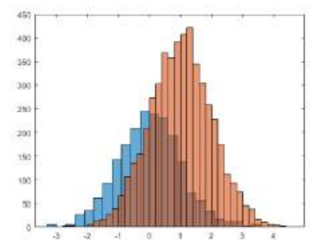
Example: ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN Mgr_ssn SET DEFAULT '333445555';**

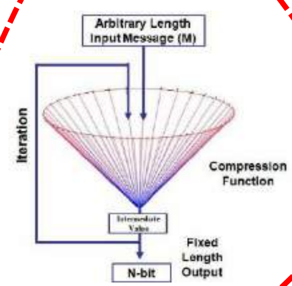
Physical Design & Hashing

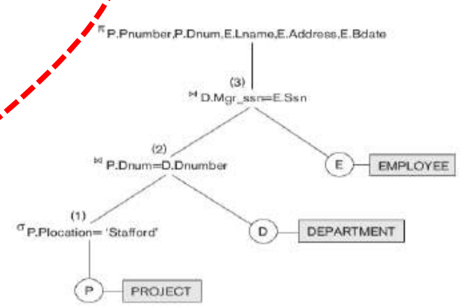
Database Systems (H)
Dr Chris Anagnostopoulos

PANDORA'S DATA MANAGEMENT SYSTEM BOX…









ROADMAP

- Physical storage and file organization.
- *record*, *block* and *blocking factor*.

- File Structures
  - Heap file
  - Sequential file
  - Hash file

- *Algorithms* for accessing data from files:
  - *Search, insert, delete, update*
  - Estimate the **expected cost**.

PHYSICAL STORAGE HIERARCHY

- 3-level storage hierarchy
  - *Primary storage*
    - e.g., RAM: main memory, cache;
  - *Secondary storage*
    - e.g., hard-drive disks (HDD), solid-state disks (SSD);
  - *Tertiary storage*
    - e.g., optical drives.

- As we *go down* the storage hierarchy
  - **Storage capacity**: increases
  - **Access speed**: decreases
  - **Money-costs**: decreases

PHYSICAL STORAGE FOR A DATABASE

Fundamental: Database is *too large* to fit in the main memory;

By default: Data access *involves* secondary storage (HDD) due to low cost…

Consequences:

- **[C1]** Since HDD is not CPU-accessible, then
  - **Step 1:** data must be first loaded *into* main memory *from* disk
  - **Step 2:** data are then processed *in* the main memory

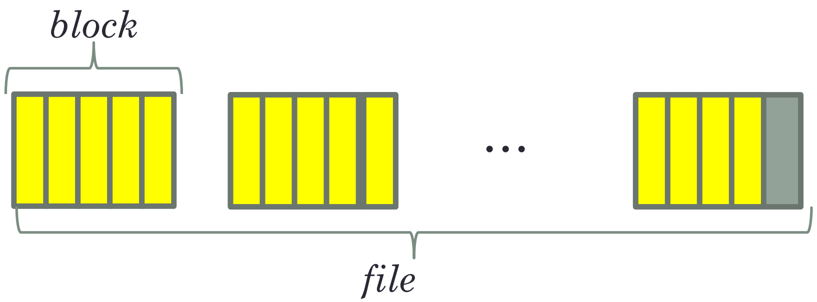
- **[C2]** The speed of data access becomes *low*
  - Data access from HDD takes **30ms**; while *only* **30ns** in RAM, thus, HDD is the main bottleneck (data transfer)

General Challenge: Organize data on HDD to *minimize* this latency.

ORGANIZATION-BASED OPTIMIZATION

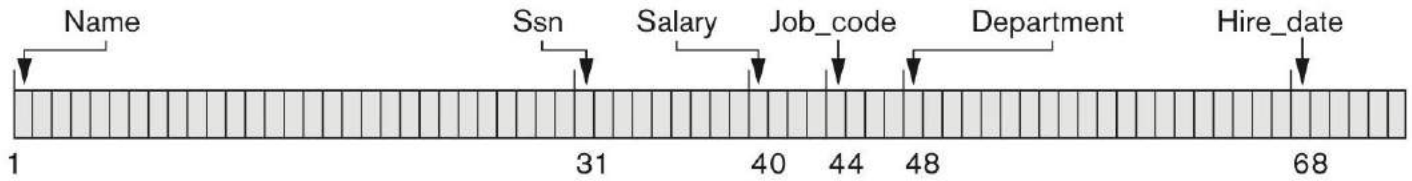
Challenge 1: Organize tuples on the disk to minimizing I/O access cost.

- Representation
  - **Tuple** is represented as a **Record**.
  - **Records** are grouped together forming a **Block.**
  - **File** is a *group* of blocks.



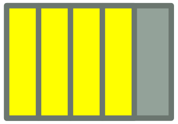
- Records can be of:
  - **Fixed length**, i.e., attributes are of fixed size in *bytes*.
  - **Variable length**, i.e., the size of each attributes varies.

Record as a Tuple



Total size of tuple: **R** = 30 + 10 + 4 + 4 + 20 + 6 = 74 bytes

Blocking Factor



Block is of *fixed*-length, normally 512 bytes to 4096 bytes.

Definition 1: **floor(*x*)** is the *largest* integer *less* than or equal to ***x*.**
e.g., floor(3.7) = 3, floor(1.1) = 1. floor(0.6) = 0.

Consider a record of **R** bytes and a block of **B** bytes.

Definition 2: The number of records stored in a block, i.e., *records per block*, is called: **blocking factor (*bfr*)**

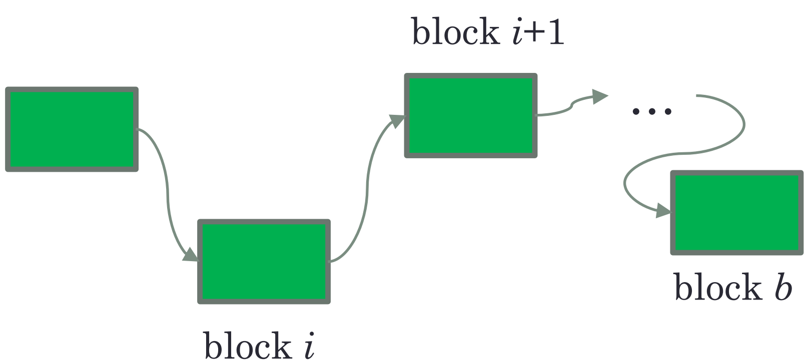
$$\boldsymbol{bfr = \mathrm{floor}(B/R)}$$

e.g., bfr = 100 records per block, that is a block can store *up to* 100 records.
Note: At least *one* record per block, i.e., $\mathbf{B \geq R}$

Blocks to Files on Disk

Challenge 2: How do we *allocate* blocks of files on the disk?

Linked allocation: Each block i has a pointer to the *physical address* of the *logically* next block i+1 anywhere on the disk, i.e., a *linked list* of blocks;



File of b blocks

IN-CLASS EXAMPLE

Context: The relation EMPLOYEE has r = 1103 tuples, each one corresponding to a *fixed-length record*. Each record has the fields:

- NAME (30 bytes),
- SSN (10 bytes),
- ADDRESS (60 bytes).

Task: Given a block size **B** = 512 bytes provided to us by the OS:

- Which is the *blocking factor* (*bfr*) of the file that accommodates this relation?
- How *many blocks* (*b*) are in the file?

In-Class Example

- Each record/tuple has size: 30+10+60 = **R** = 100 bytes

- Blocking factor is: *bfr* = floor(**B/R**) = floor(512/100) = floor(5.12) = **5 records per block**

- Number of blocks *b* = ceil(*r* / *bfr*) = ceil(1103/5) = ceil(220.6) = **221 blocks.**

Note: ceil(*x*) is the *least* integer that is *greater* or equal than *x*, e.g., ceil(2.6) = 3.

Parameter	Notation
bfr	Blocking factor (tuples per block)
R	Tuple size (bytes)
b	Number of file blocks
B	Block size (bytes)

File Structures

Challenge 3: How to *distribute* records within blocks to *minimize* I/O Cost?

- **Heap File** (*unordered* file)
  - **Principle**: a new record is added to the *end* of the file, i.e., at the end of the *last* block (*append*).
- **Ordered File** (*sequential* file)
  - **Principle:** records are kept *physically sorted* w.r.t ordering field.
- **Hash File**
  - **Principle:** a *hashing* function $y = h(x)$ is applied to each record field x (hash field)
  - The output y is the *physical block* address; *mapping a record to a block*!

Challenge 4: Which *will* be the ordering field or the hash field of a relation to *minimize* the I/O cost?

Expected I/O Access Cost

Fix a file type *heap*, *ordered*, or *hash*. We define **I/O access cost as the cost for:**

- **Retrieving** a *whole* block *from* disk *to* memory to search for a record w.r.t. searching field (search cost);
- **Inserting/deleting/updating** a record by transferring the *whole* block *from* memory *to* disk (update cost);

Cost Function: expected number of block accesses (read/write) to search/insert/delete/update a record.

Note: *block* is the minimum **communication unit;** we transfer *only* blocks and *not* records from disk to memory and vice versa!

Expectation of a Random Variable

- Let X be a *discrete* random variable (*attribute*),
  - e.g., $X \in \{1, 3, 6, 10\}$

- Let $P(X = x)$ be the probability that X has the value x
  - $P(X = 1) = 0.2$, $P(X=3) = 0.1$, $P(X = 6) = 0.5$, and $P(X = 10) = 0.2$
  - $\sum_x P(X = x) = 1$

- Expectation of X, E[X], is the weighted sum of the values:

$$E[X] = \sum_x P(X = x) \cdot x$$

- e.g., $E[X] = P(X=1)\cdot 1 + P(X=3)\cdot 3 + P(X=6)\cdot 6 + P(X=10)\cdot 10 = 5.5$

Expectation of a Cost Function

- Let assign for *each* value *x*, a specific real-valued function C(X) indicating e.g., the *cost* for accessing X in number of block accesses.

- C(X) is a random variable *being* a function of X.

- e.g., C(X = 1) *or* C(1) = 3 block accesses, C(3) = 1 block access, C(6) = 2 block accesses, and C(10) = 4 block accesses.

The expected cost E[C(X)] in number of block accesses is:

$$E[C(X)] = \sum_x P(X = x) \cdot C(x) = 2.5 \text{ block accesses}$$

HEAP FILE

Inserting a new record is *efficient*:

- Load the *last* block from disk to memory (address is in file header)
- Insert the new record at the end of the block *and* write it back to disk.

Complexity: 2 block accesses, i.e., **constant O(1) block access**.

Retrieving a record is *inefficient*:

```
SELECT * FROM EMPLOYEE
WHERE Salary = 23000
```

- *Linear* search through all the b file blocks.
- Load a block at a time from disk to memory; search for the record; *repeat*

Complexity:

- On average: ~b/2 blocks (best case: 1 block; worst case: b blocks)
- We access b blocks if the record is not in the file.
- **O(b) block accesses,** does not scale well with b.

Heap File

```
DELETE FROM EMPLOYEE
WHERE SSN = '1234567'
```

Deleting a record is *inefficient*:

- *Find* and *load* the block containing the record; (retrieval process).
- Remove the record from the block and *write* the block back to disk.
- This leaves *unused* spaces within blocks!

Complexity: **O(*b*) + O(1) block accesses**.

Use *deletion marker* per record

- a bit from 0 to 1: bit = 1 indicates that a record is deleted.
- *periodically*, re-organize the file by gathering the non-deleted records (bit=0) and freeing up blocks with deleted records.

Sequential File

All the records are *physically sorted* by an ordering field and are kept sorted at *all* times.

Suitable for SQL queries that:

- Require *sequential* scanning:

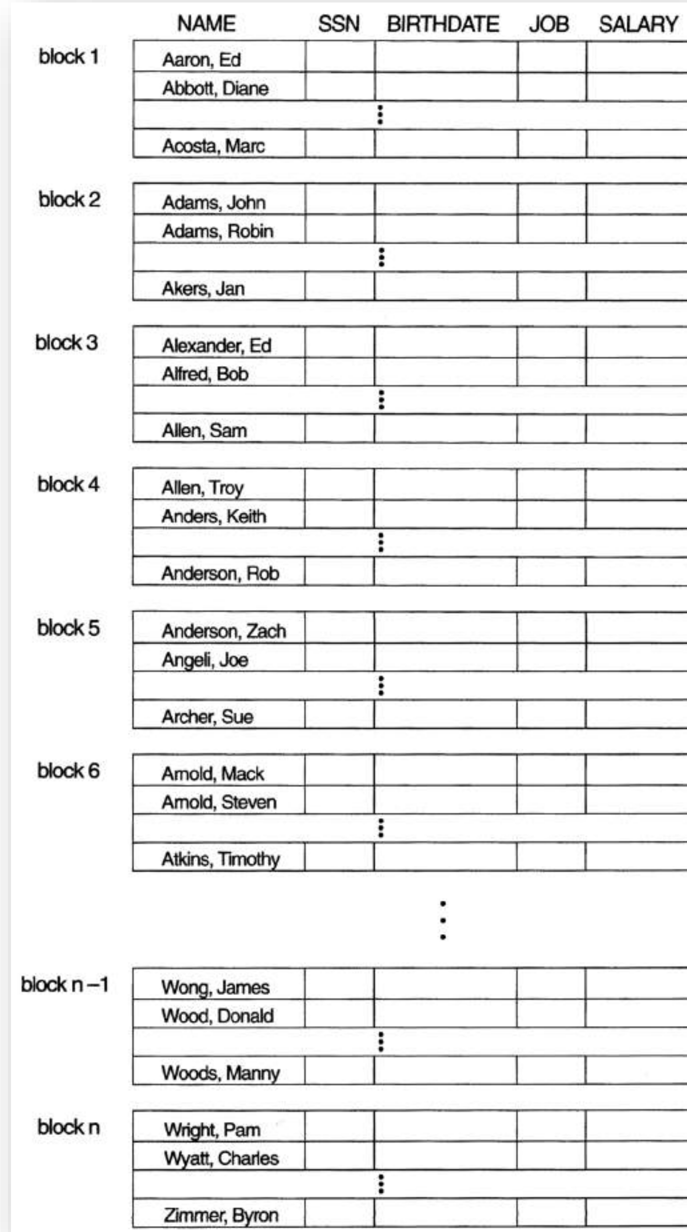
  ```
  SELECT Name
  FROM EMPLOYEE ORDER BY Name
  ```

- *Involve* ordering field in search:

  ```
  SELECT * FROM EMPLOYEE
  WHERE Name LIKE 'Allen, Troy'
  ```

- Range *queries over* the ordering field:

  ```
  SELECT * FROM EMPLOYEE
  WHERE Name > 'Aaron'
  AND   Name < 'Archer'
  ```



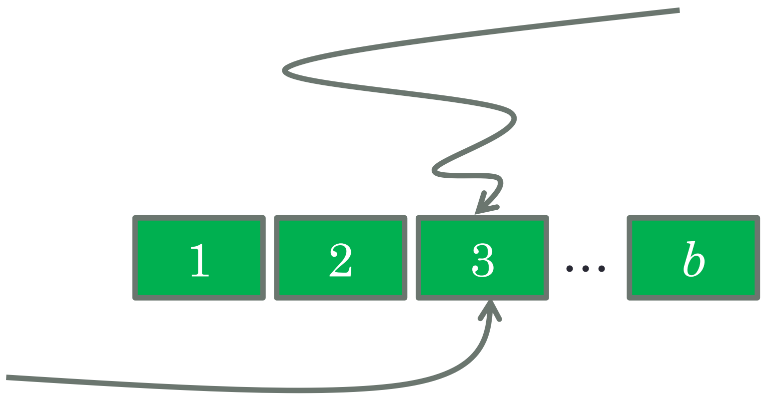
SEQUENTIAL FILE

Retrieve a record using the *ordering* field; *efficient*

- The block is found using *binary search* on the ordering field.

Complexity: O(log$_2$*b*), i.e., *sub-linear* with *b*

```
SELECT * FROM EMPLOYEE
WHERE Name = 'Chris'
```



Retrieve a record using a *non-ordering* field; *inefficient*

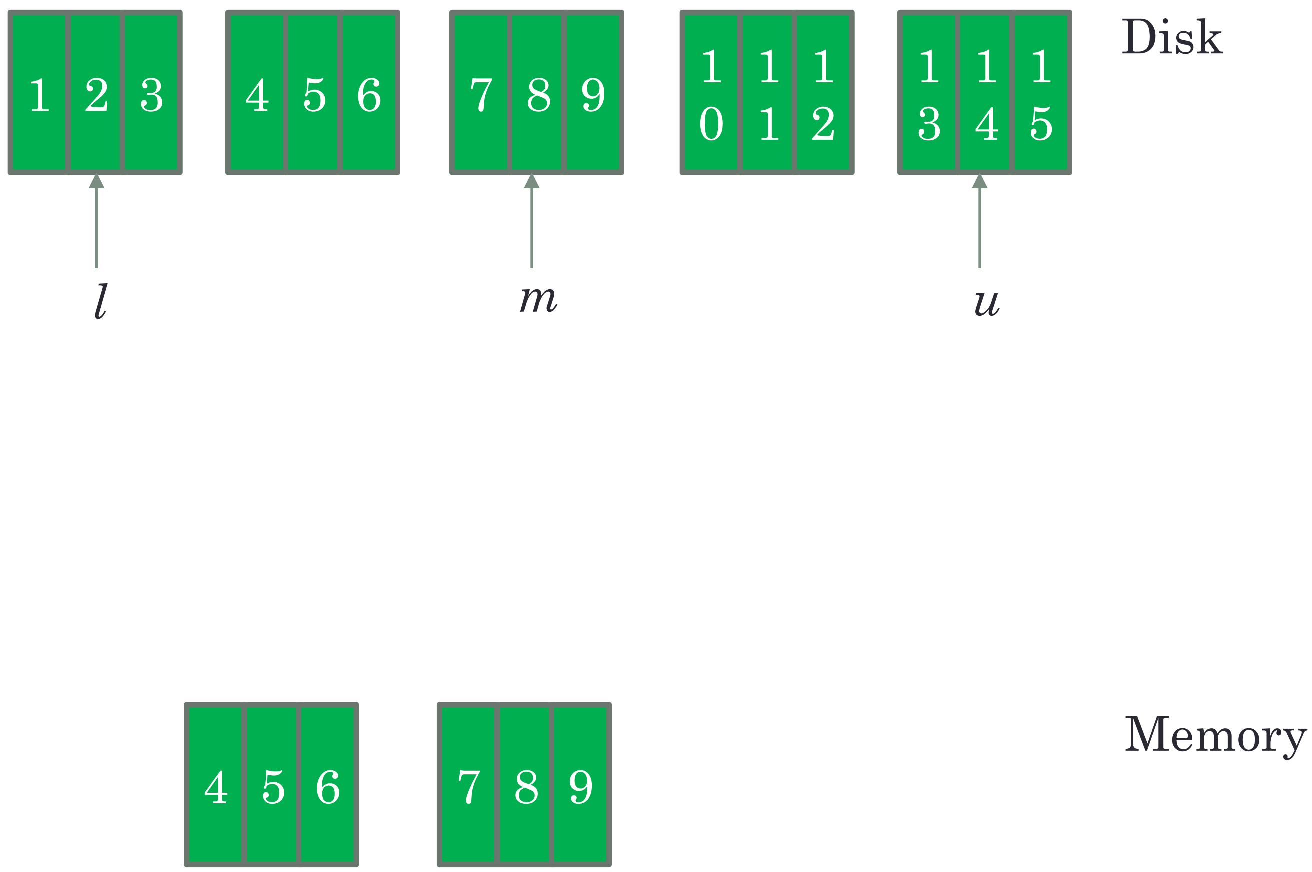
- We do not *exploit* the ordering…like a heap file.

Complexity: O(*b*) i.e., *linear* with b.

```
SELECT * FROM EMPLOYEE
WHERE Salary = 23000
```

Binary Search (Block Mode)

$k = 5$



Binary Search Algorithm

Binary Search (k) /*search for a record with **key** value k*/

$l \leftarrow 1$; $u \leftarrow b$; /* b is the number of blocks */

while ($u \geq l$) **do**

begin

$i \leftarrow (l + u)$ div 2; /*go to the *middle* block*/

read block i from disk to the memory; /*1 block access*/

if $k <$ (**ordering** key value of the *first* record in block i)

then $u \leftarrow i - 1$; /**narrow* the search in *previous* blocks of block i*/

else if $k >$ (ordering key field value of the *last* record in block i)

then $l \leftarrow i + 1$; /**narrow* the search in *next* blocks of block i*/

else if record with ordering key value = k is in memory

then *found*

else *not-found*;

end;

Discussion

Hypothesis: Range queries are *efficient*.

Experiment: retrieve records such that: £30K ≤ *Salary* ≤ £50K;

Note: the file is sorted w.r.t. *Salary*

```
SELECT *
FROM   EMPLOYEE
WHERE  Salary BETWEEN 30 AND 50
```

Methodology:

- Find the block *i* which contains the record with salary = £30K using *binary* search: $\mathbf{O(\log_2 b)}$
- Then, the *contiguous* blocks $i+k$ are *fetched* until the range is *exhausted*, $k = 0, \ldots, < b$, thus $\mathbf{O(b)}$

Complexity: $\mathbf{O(\log_2 b) + O(b)}$ **block accesses**

```
INSERT INTO R
VALUES(32222,'Verdi','Music',48000)
```

Sequential File

- **Insertion** is *expensive.*
  - First, *locate* the block where the record should be inserted: *binary search.*
  - On average, *half* of the records must be moved to *make* room for the new record.

    …very expensive for large files!

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	

Overflow block

- **Alternative:** *chain* pointers
  - **Principle:** Each record points to the logically *next* ordered record.
  - **If** there is *free* space in the *right* block, insert the new record there.
  - **Else**, insert the new record in an *overflow block* and use chain pointers;
  - **Pointers must be updated;** it is a *sorted-linked* list.

Sequential File

Deletion is *expensive*.

- First, *locate* the block where the record is to be deleted; *binary search*.
- Update the deletion marker from 0 to 1 and **update** the pointer *not* to point to the deleted record.
- *Periodically* re-sort the file to restore the physically sequential order (i.e., external sorting…*expensive*)

```
DELETE FROM EMPLOYEE
WHERE SSN = 123
```

Update on the *ordering field* is *costly*.

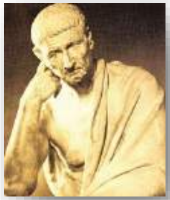
- The record is *deleted* from its *old* position & *inserted* into its *new* position.

```
UPDATE EMPLOYEE
SET SSN = 123
WHERE SSN = 456
```

Update on a non-*ordering field* is *efficient*!

- **Complexity:** $O(\log_2 b) + O(1)$ block accesses

```
UPDATE EMPLOYEE
SET Salary = 20000
WHERE SSN = 123
```



Sequential File Performance

Sequential File	I/O Cost Complexity
Search by ordering field	$O(\log_2(b))$ binary search
Search by non-ordering field	$O(b)$ linear search

Rhetoric Question: *Why* the binary search over b blocks requires $\sim\log_2 b$ accesses?

Answer: Logarithm of x indicates the *number of divisions* we need to divide x by 2 to reach 1, *e.g.*, $log_2(140) = 7.12$ steps to divide 140 by 2 to reach 1.

We split the search space into two sub-spaces at every step, and repeat that…until finding the block!

What if: we could split into 3 or m > 3 subspaces every time?
Conjecture: $\sim\log_m b$ block accesses?

Hash File

Let's focus *only* on selections using the *equality* predicate over a searching field *k*:

```
SELECT * FROM EMPLOYEE WHERE SSN = 1234567
SELECT * FROM EMPLOYEE WHERE SALARY = 23000
```

Idea of Hashing

- *Partition* the records into M *buckets*: bucket 0, bucket 1, ..., bucket M-1.
- Each bucket can have more than *one* block.
- Choose a **hash function $y = h(k)$** with output $y \in \{0, 1, \ldots, M-1\}$ for a given *k*.

- **Requirement**: *h uniformly* distributes records into the buckets {0, …, M-1}, i.e., for each value *k,* each bucket is chosen with *equal* probability 1/M:

$$\boldsymbol{y = h(k) = k \bmod M},$$

modulo (**mod**) operator is the remainder of the division: *k* divided by M.

Arithmetic: 3 mod **8** = 3; 12 mod **8** = 4; …

Hash File Construction: External Hashing

Mapping a record to a bucket $y = h(k)$ is called *external hashing over hash-field* k.

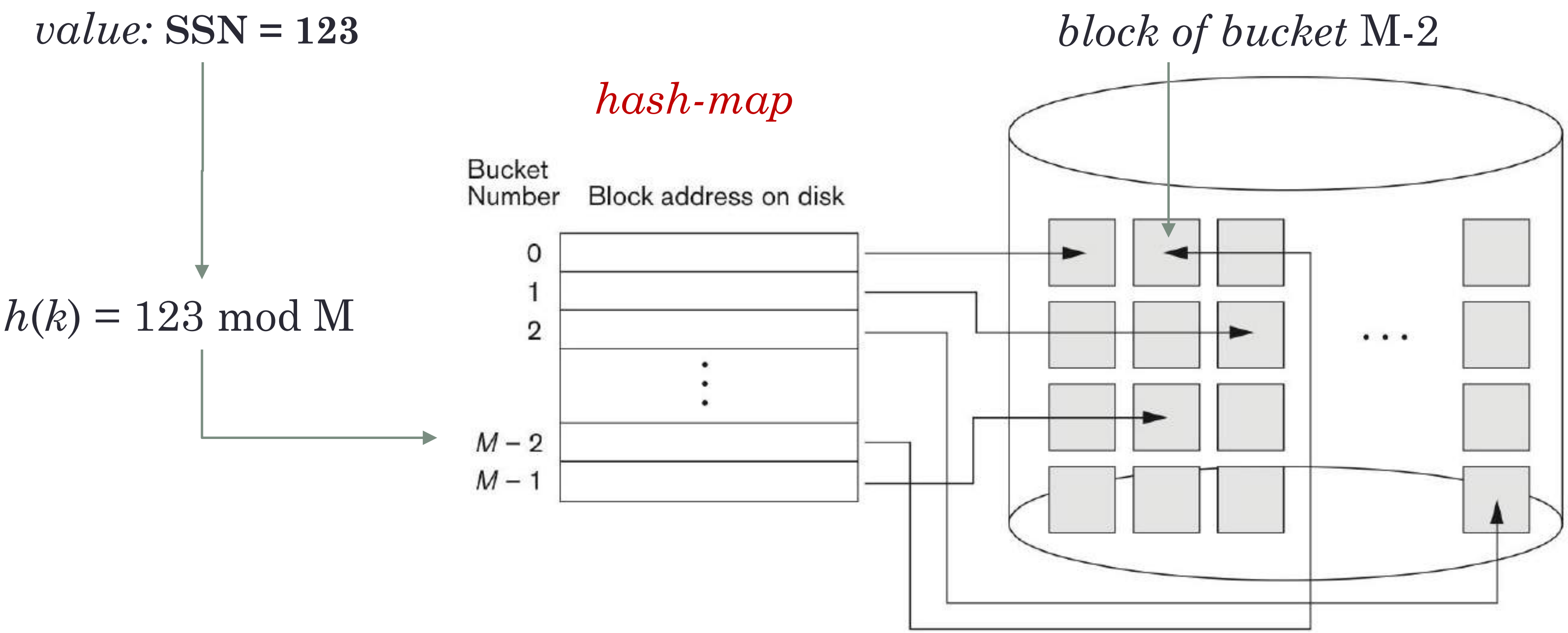
Normally, **collisions occur** i.e., two or more records are mapped to the *same* bucket

Example: Let M = 3, $h(k) = k \bmod 3 \in \{0, 1, 2\}$, thus, we obtain three buckets.

Records with k = **1**, **11**, **2**, and **4** are stored to buckets **0**, **2**, **2**, and **1**, respectively. *Collision* on bucket 2.

Indirect clustering: group tuples together w.r.t. their hashed-values y and not w.r.t. their hash-field values k ☺

External Hashing Algorithm



Retrieve a possible record with SSN = 123:

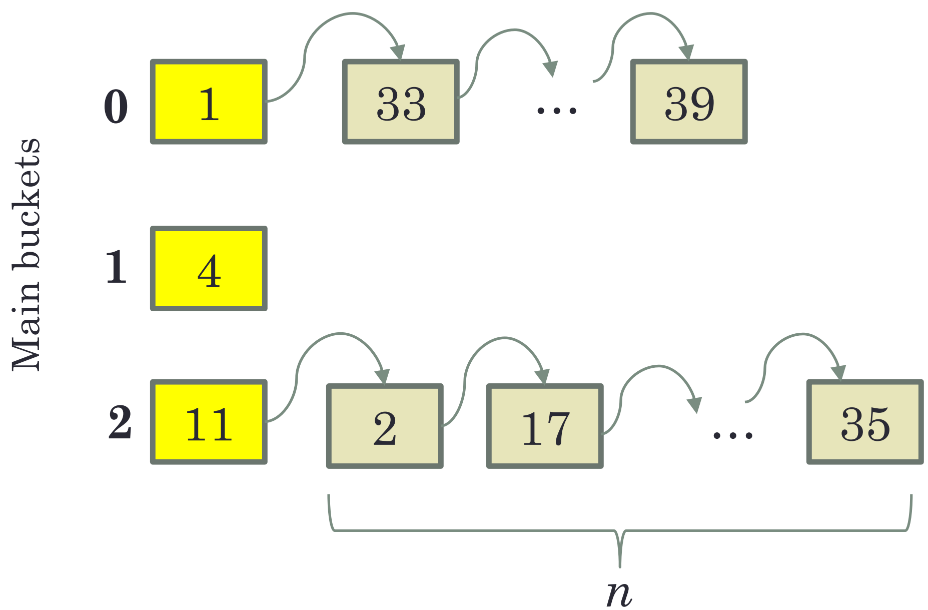
1. Hash k and get the corresponding bucket, e.g., h(SSN) = M-2.
2. Use the hash map to get the block address in disk of the M-2 bucket.
3. Fetch the block from the disk to memory.
4. Linear search in memory to find the record such that: h(SSN) = M-2.

Complexity: O(1) block access, i.e., *directly* get the block containing the record.

External Hashing Overflow

Due to collisions, i.e., more than one record is mapped to the same bucket, the buckets might be *full*.

- **Problem:** How can we insert a new record *hashed* to a full bucket?
- **Solution**: Adopt, *again*, the *chain pointers* method.



k = {**1, 4, 11**, 33, 2, 17, 39, 35, …}

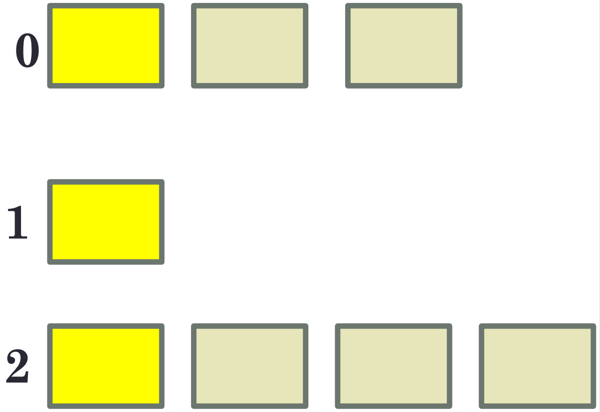
y = k mod 3

Complexity: O(1) + O(n) block accesses; n = number of overflow blocks; $n < b$

External Hashing

Delete a record based on the *hash* field

- If record is in the *main* bucket, delete it **O(1)**
- Else follow the chain to overflow block **O(1) + O(*n*) block accesses.**
- *Periodically* pack together blocks of the *same* bucket to free up blocks with deleted records.



Update a record based on a *non-hash* field

- Locate record in main *or* overflow bucket
- Load block into memory, *update* and *write* it back.
- **O(1)** *or* **O(1) + O(*n*) block accesses.**

Update a record on the *hash* field: change the hashed-value! delete form the *old* bucket and insert to the *new* bucket.

In-Class Example

Number of buckets: M = 3; 1 block per main bucket; *bfr* = 2 records/block

SSN key values *k* = {1000; 4540; 4541; 4323; 1321; 1330}

Task 1: Assign employees to buckets w.r.t. SSN and ***y* = SSN mod 3**

Task 2: Calculate the expected number of block accesses for a random SQL:
`SELECT * FROM EMPLOYEE WHERE SSN =` *k* …in the *worst* and *best* case!

- **Worst case:** the record we are searching for is in the very last block ☹
- **Best case:** the record we are searching for is found immediately in the very first block (main bucket) ☺

Task 3: Compare with the Heap File and Sequential File (order w.r.t. SSN) for the *best*, *worst*, and *average* cases.

In-Class Example

Number of buckets: M = 3; 1 block per bucket; *bfr* = 2 records/block

SSN key values *k* = {1000; 4540; 4541; 4323; 1321; 1330}

Task 1: Assign employees to buckets w.r.t. SSN;

Assignment: 1000 mod 3 = 1; 4540 mod 3 = 1; 4541 mod 3 = 2; 4323 mod 3 = 0; 1321 mod 3 = 1; 1330 mod 3 = 1; *enough…*

1000
4540
1321
1330

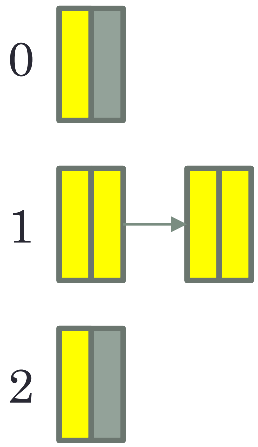
4323 4541

0 1 2

In-Class Example

Task 2 [Worst Case]: Calculate the *expected* number of block accesses

```
SELECT *
FROM EMPLOYEE
WHERE SSN = k
```



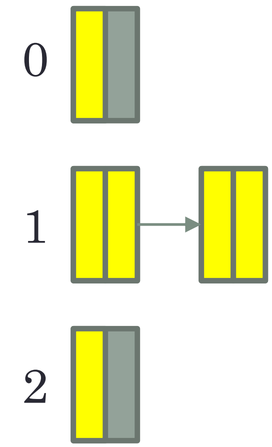
[Theory] Each bucket is equi-probable with probability 1/3 = 0.33 or 33%

Fact: 4 blocks = 3 main + 1 overflown

- M = 0: **1 record**
- M = 1: **2 records + 2 records overflown**
- M = 2: **1 record**

Cost: 0.33***1** + 0.33*(**1 + 1**) + 0.33***1** = **1.32 block accesses**

IN-CLASS EXAMPLE



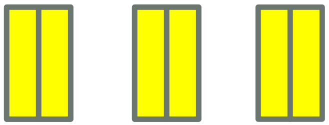
Task 2: [Best Case]

Cost: 0.33***1** + 0.33***1** + 0.33***1** = **1 block access**

Task 3: Comparison

[Worst Case]

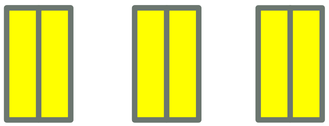
- **Hash:** 1.32 block accesses
- **Heap [3 blocks] linear scan:** 3 block accesses
- **Sequential [3 blocks] binary search:** $\log_2(3)$ = 1.58 block accesses



Heap

[Best Case]

- **Hash:** 1 block access
- **Heap [3 blocks] linear scan:** 1 block access
- **Sequential [3 blocks] binary search:** 1 block accesses



Sequential

[Average Case]

- **Hash:** 0.33***1** + 0.33*(0.5***1 +** 0.5***2**) + 0.33***1** = 1.15 block access
- **Heap [3 blocks] linear scan:** (1+3)/2 = 2 block accesses
- **Sequential [3 blocks] binary search:** $\log_2(3)$ = 1.58 block accesses

CRASH TEST

Hypothesis: Range queries are *inefficient* (Achilles heel)

- **Hash field**: AGE (integer number)
- **Experiment**: Retrieve employees with **20** ≤ AGE ≤ 50

```
SELECT *
FROM EMPLOYEE
WHERE AGE BETWEEN 20 AND 50
```

Methodology:

- Find the bucket, which contains the records with AGE = 20: **O(1) + O(*n*)**
- The *continuous* values: 21, 22, …, 50 are not mapped to the same bucket!
- **Why?** ideal hash function uniformly *distributes* values over buckets

- Thus, each value 21, 22, …, 49, 50 is treated as a separate query!
- Let *m* is be the **number of distinct values** in the range…

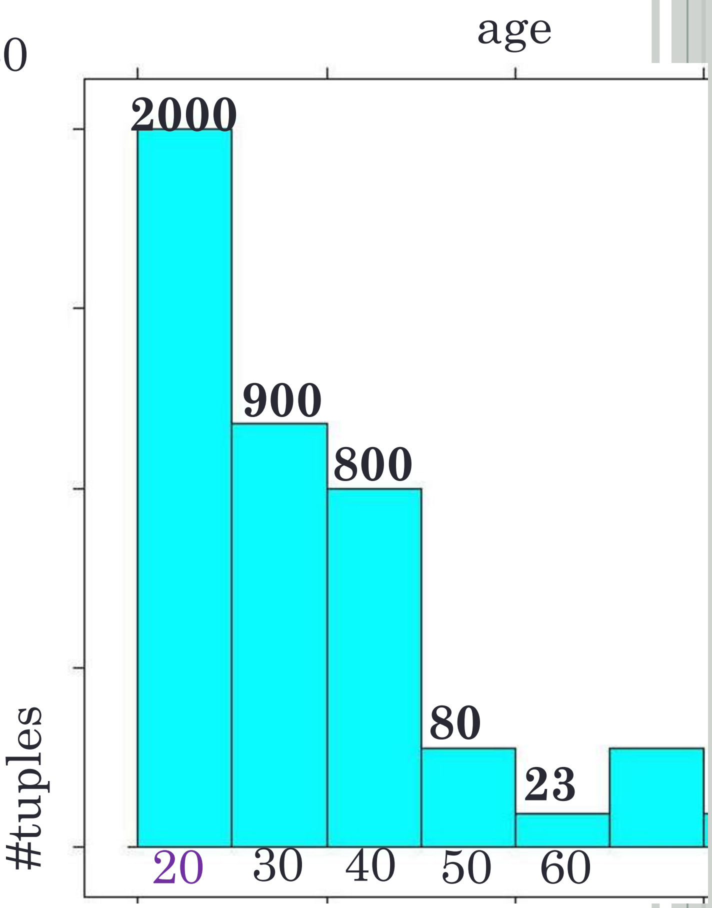
Complexity: O(*m*) + O(*nm*) block accesses; *n* overflow blocks *per* bucket ☹

PREDICTABLE OR UNPREDICTABLE?

Hypothesis: The distribution of the values *influences* the expected cost.

Conjecture: the expected cost is **unpredictable**!

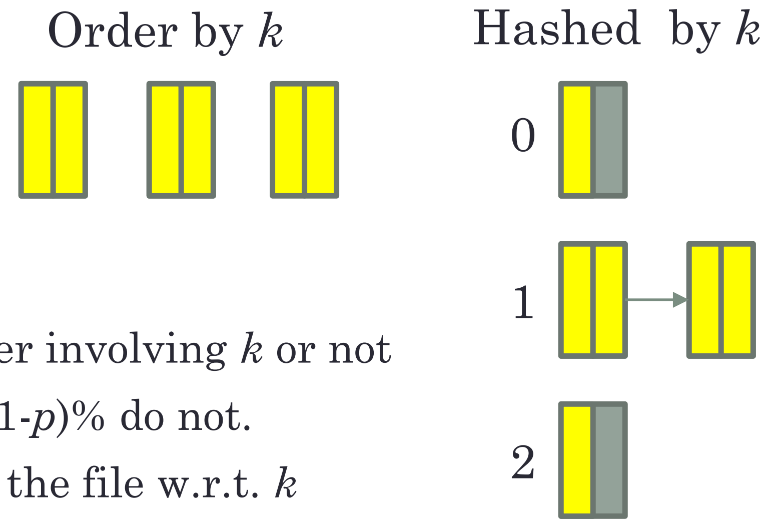
- **Experiment 1**: Retrieve employees with *age* = 20
- **Experiment 2**: Retrieve employees with *age* = 60
- **Blocking factor bfr** = 40 employees per block.



- Find the bucket with age = 20:
- **O(1) + O(*n*) block accesses** including *n* overflow blocks ☹
- ceil(2000/40) = 50 blocks
- 1 main block + 49 overflown blocks

- Find the bucket with age = 60:
- **O(1) = 1 block access** ☺
- ceil(23/40) = 1 block
- 1 main block

Worked Example



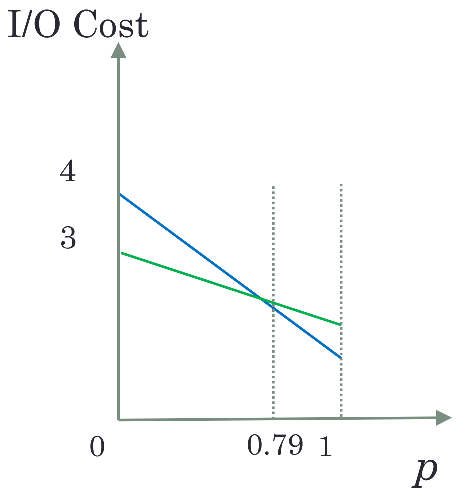
Fact: bfr = 2 records/block; focus on attribute k

Observation: Issuing *many* selection queries either involving k or not

Fact: p% of those queries involve attribute k and (1-p)% do not.

Task: Find a decision rule whether to hash *or* sort the file w.r.t. k

- **Hash worst-case expected cost:**
  - **H1:** p*(0.33*1 + 0.33*2 + 0.33*1) + (1-p)*4 = 4 – 2.68*p
- **Sort worst-case expected cost:**
  - **H2:** $\log_2(3)$*p + (1-p)*3 = 3 – 1.41*p



Decision: IF H1 < H2 **THEN** hashing**; ELSE** sorting

Condition: 4 – 2.68*p < 3 – 1.41*p or $\boldsymbol{p > 0.79}$

i.e., if at least 79% of the queries involve k, then hash file w.r.t. k, otherwise, sort file by k!

@Explore the Heap File solution ☺

Indexing Methodology Part I

Database Systems (H)
Dr Chris Anagnostopoulos

Roadmap

- **Index:** *alternative* access path using *any* field.
- **Primary Index**, **Clustering Index**, **Secondary Index**
    - **Trade-off:** Search Speed *vs* Overhead (Storage, Maintenance)
- **Challenge**: Expedite search by splitting the space in *more than two* subspaces.
- **Multi-Level Index** Structure
    - ISAM Search Tree by IBM® (1975)
    - B Tree & B+ Tree are used by *all* SQL & NoSQL Systems (*next week*)

OBJECTIVES

Physical Design (*last week*)

- **Objective**: given a *specific* file type provide a **primary access** path based on a *specific searching* field, *e.g.*, search only via SSN.

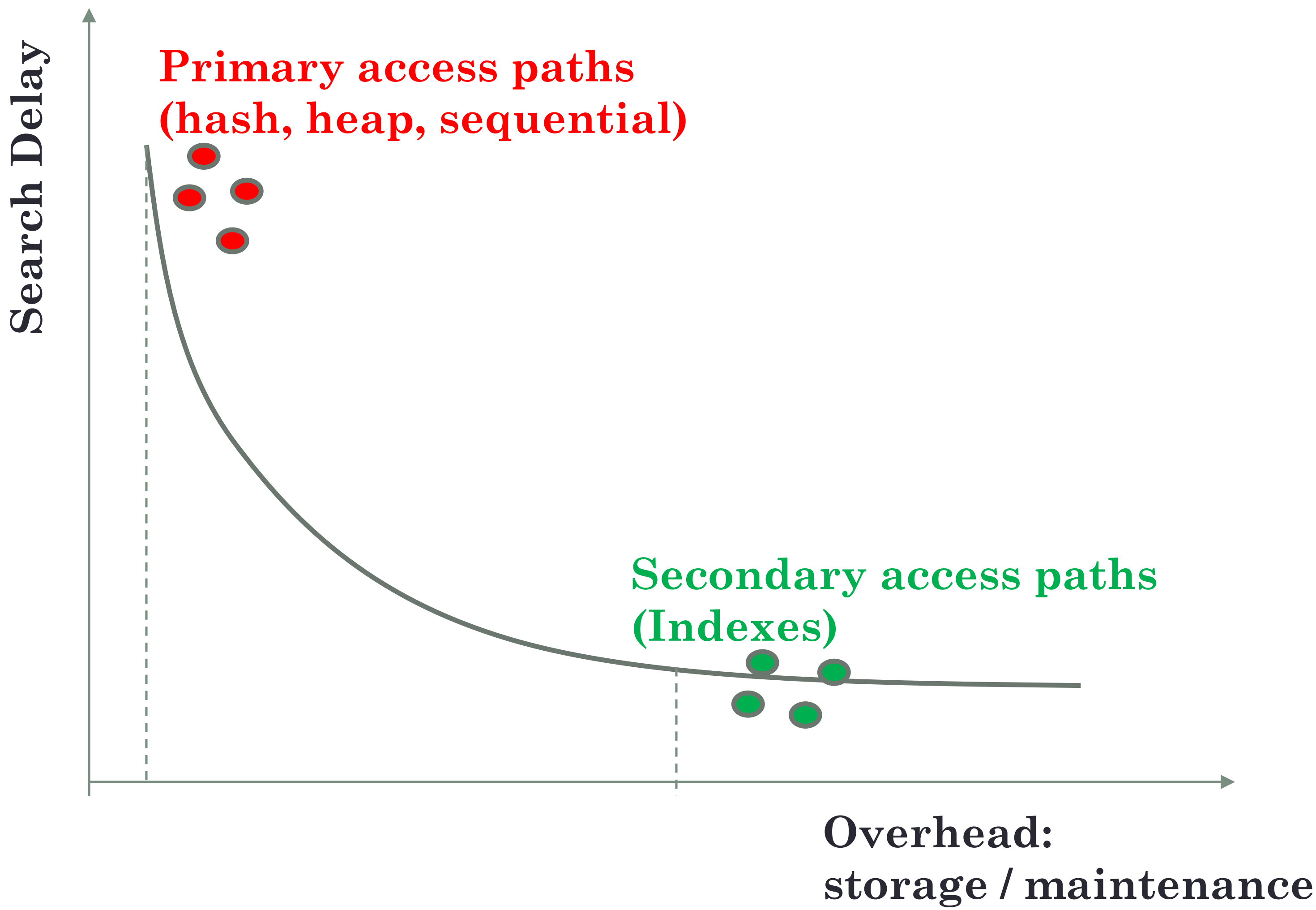
Index Design (*this week*)

- **Objective**: given *any* file type provide a **secondary access** path using *more* than one searching field, *e.g.*, SSN, Salary, Name, etc.

Cost: Additional (**meta-data**) files on the disk & maintenance cost.

Benefit: **Expedite** significantly the search process *avoiding* Linear Scan.

Trade off: Overhead vs Search Speed



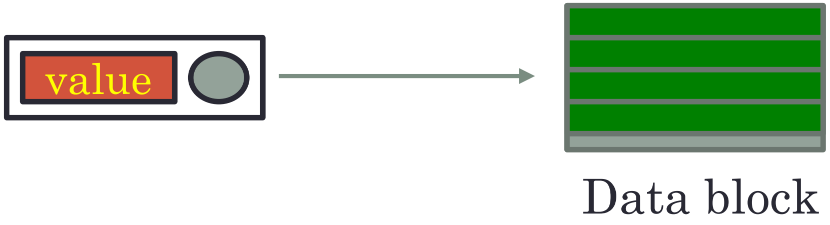
PRINCIPLES

Principle 1: Create *one* index over *one* field: **index field**

Principle 2: An index is an *another separate* **file**

Principle 3: All index entries are **unique** & **sorted** w.r.t. index field

index-entry = (index-value, block-pointer)



Principle 4: *First* search within the index to *find* the block-pointer, *then* access the data block from the data-file.

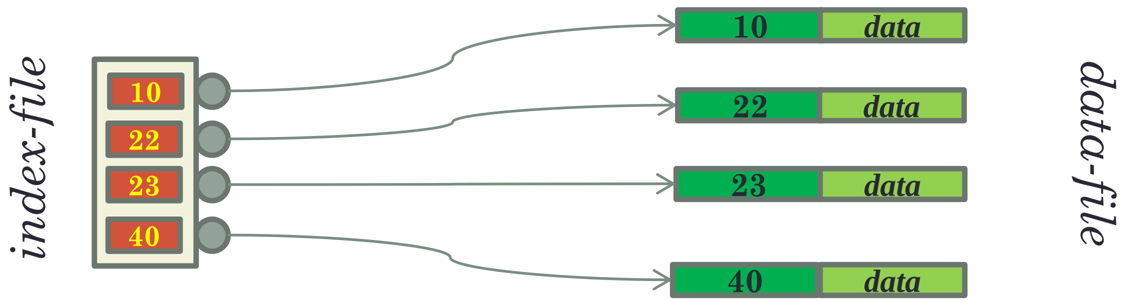
Hypothesis 1: Index file occupies *less* blocks than the data-file.

Fact: index entries are smaller records: (index-value, block-pointer)

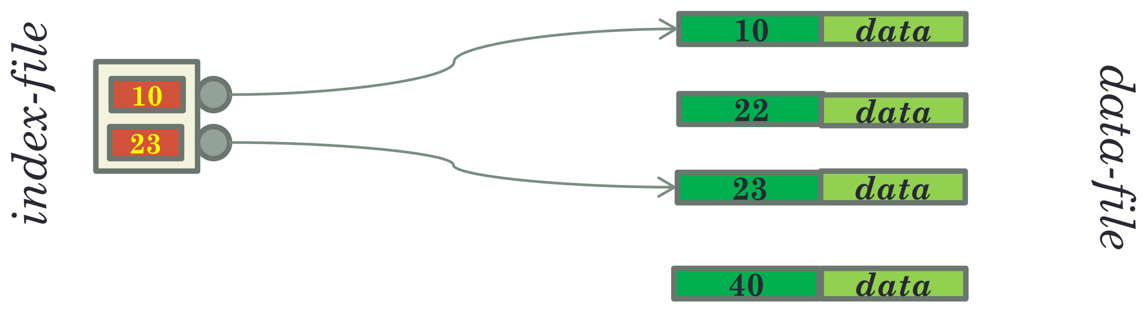
- we fit in *more* index-entries in a block *than* data-records:

  bfr(*index-file*) > bfr(*data-file*)

- **Dense Index:** *an* index entry for *every* record in the file.



- **Sparse Index:** index entries *only* for *some* of records.



Hypothesis 2: Searching over index is *faster* than over file.

Fact: By design, index is an *ordered* file, *thus* we adopt **binary-based** and/or **tree-based** methods to find the *pointer* to the actual data-block.

Now, let's prove Hypotheses 1 and 2.

Index Types

- **Primary Index:** index field is *ordering, key field* of a *sequential file*, e.g., *SSN; file is sorted by SSN*

- **Clustering Index:** index field is *ordering, non-key* field of a *sequential file*, e.g., *DNO; file is sorted by DNO*

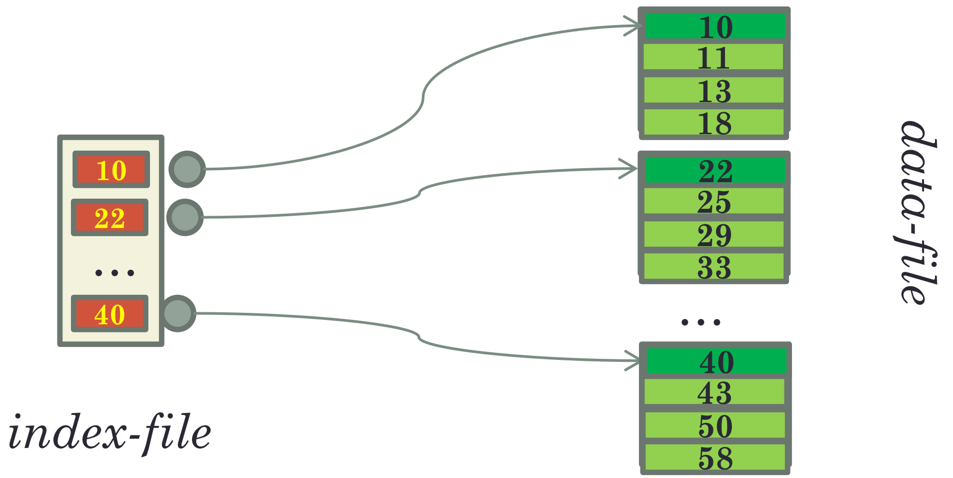
- **Secondary Index:** index field is:
  - *non-ordering, key* field, e.g., unique *passport number,* over an *ordered* (e.g., by SSN) or a *non-ordered* file.

  - *non-ordering, non-key* field, e.g., *salary,* over an *ordered* (e.g., by SSN) or a *non-ordered* file.

PRIMARY INDEX

An *ordered* file over an **ordering, key $\boldsymbol{k}$** of *a sequential* data-file:

- fixed-length index entries:= pair **($\boldsymbol{k_i}$, $\boldsymbol{p_i}$)**.
- $\boldsymbol{k_i}$ is the *unique* value of the index field
- $\boldsymbol{p_i}$ is the *pointer* to the *i*-th block containing the record with key $\boldsymbol{k_i}$



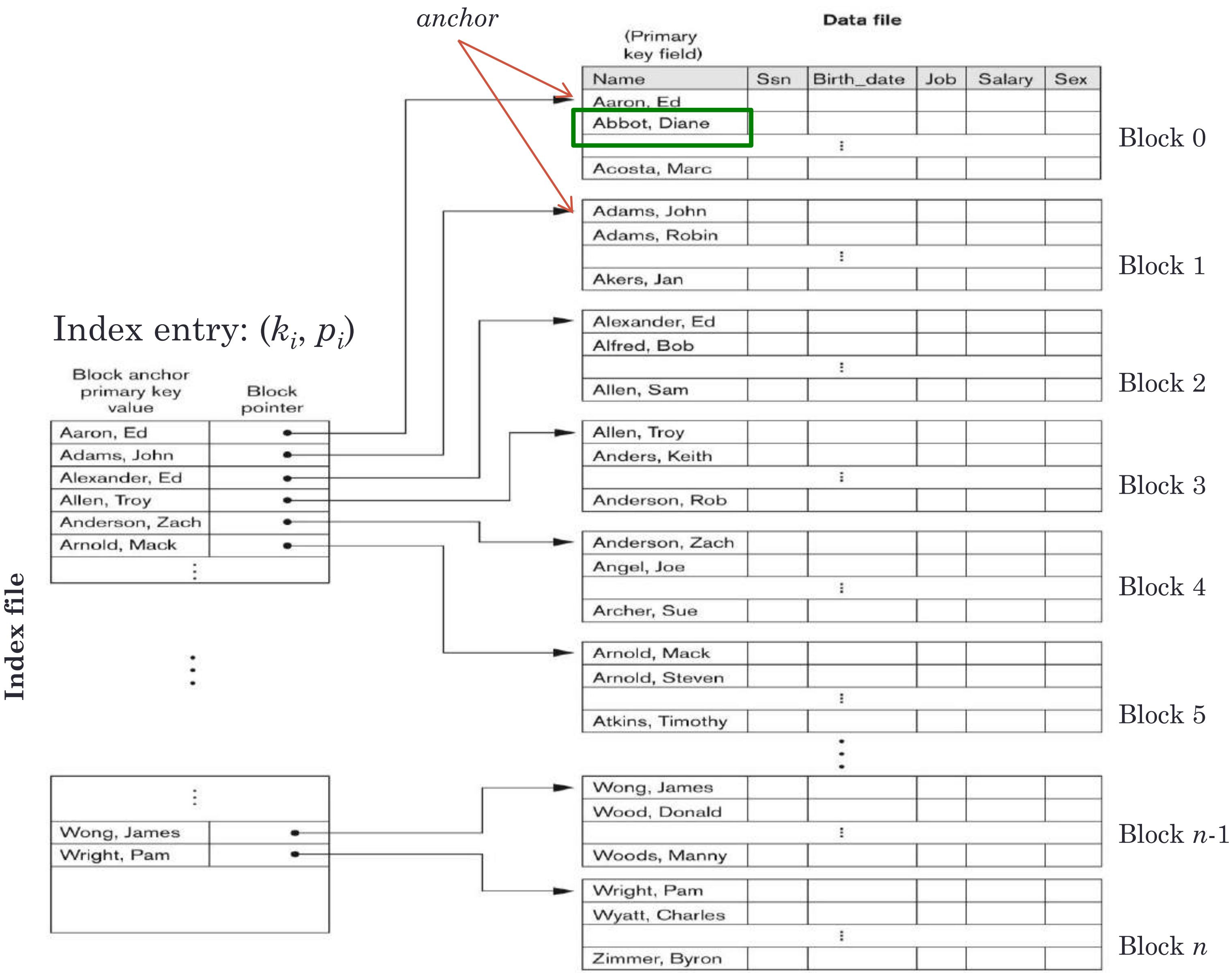
Sparseness: *one* index-entry *per* data block.

- *i*-th index entry **($\boldsymbol{k_i}$, $\boldsymbol{p_i}$)** refers to the *i*-th data block
- k_i is the field value of the *first* record in block *i*
- The *first* data-record in block *i* with value k_i is the ***anchor*** of block *i*.

Primary Index

Q1: What happens if the anchor record is deleted/updated?

Q2: What happens if a non-anchor record is deleted/updated?



IN-CLASS EXAMPLE [E1]

EMPLOYEE: r = 300,000 fixed-length records of size **R** = 100 bytes each; block size **B** = 4,096 bytes; SSN size = 9 bytes; Pointer size = 6 bytes

Task: Expected cost `SELECT * FROM EMPLOYEE WHERE SSN = k`

- **Blocking factor**: *bfr* = floor(B/R) = 40 records per block;
- **File** b = ceil(r/bfr) = **7,500 blocks**

Primary Access Path:

- Linear Search over the File: $b/2$ = **3,750 block accesses**.
- Binary Search/ordered by key SSN: ceil($\log_2(b)$) = **13 block accesses**.

In-Class Example [E1]

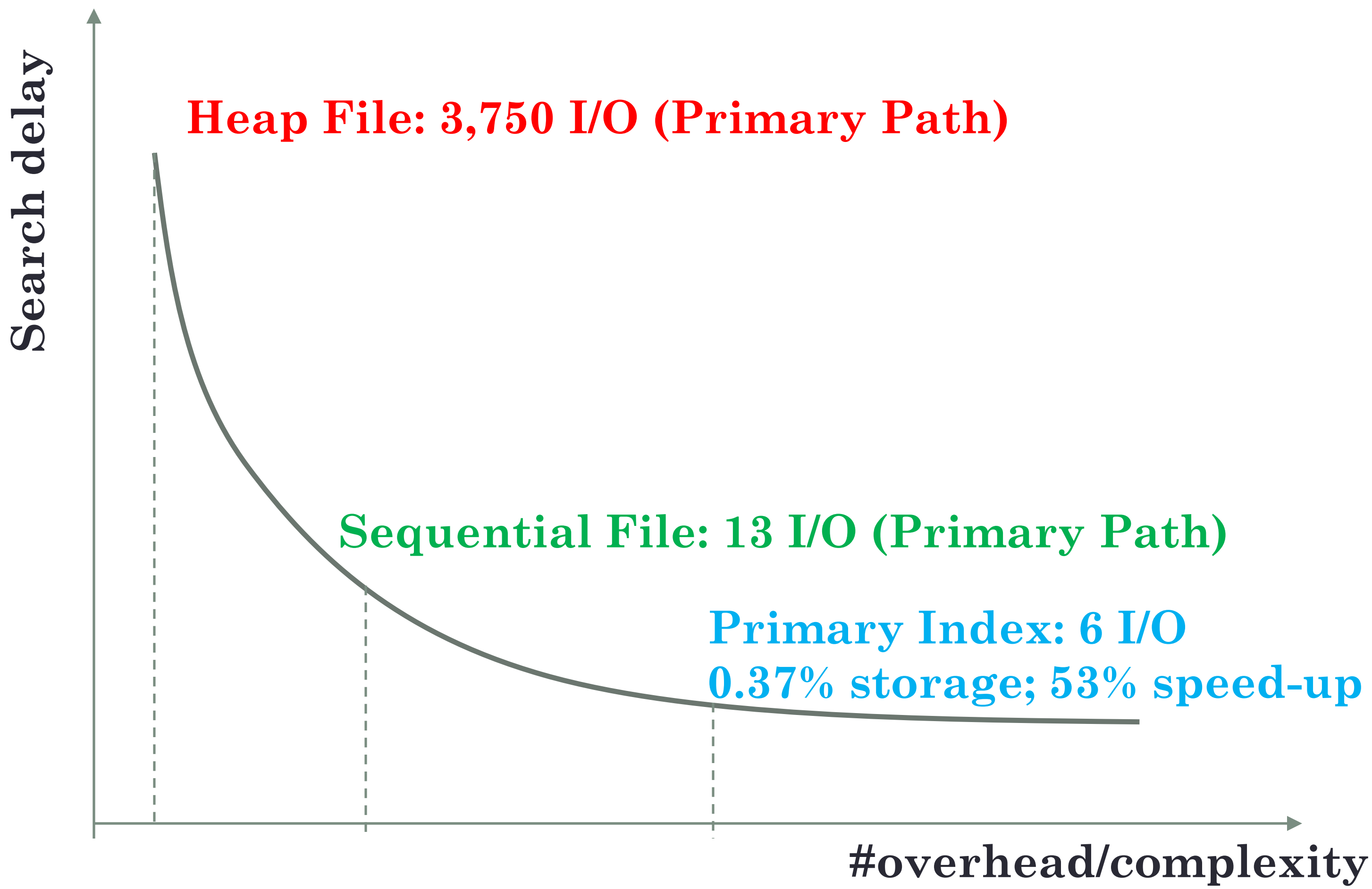
Create Primary Index on SSN: `CREATE INDEX ON EMPLOYEE(SSN);`

- **Index Entry:** {SSN, Pointer}
- **Index Entry Size:** V = 9 bytes for SSN and P = 6 bytes for Pointer.
- **Index Blocking Factor:** ibfr = floor(**B**/(**P+V**)) = 273 entries/block.
- **Primary Index requires** 7,500 entries: one per block (b = 7500).
- **Index blocks**: ib = ceil(7,500/273) = **28 blocks.**

- **Overhead:** 28 blocks more (**0.37% additional storage**)
- **Gain:** Binary Search on Index: ceil($\log_2(ib)$) = **5 block accesses.**
- We need *one* more block access to load the data-block pointed by index:
- **Total: 5 + 1 = 6 block accesses;**

- Binary Search on File: **13 block accesses (53.8% speed-up).**
- Linear Search on File: **3,750 block accesses (99.8% speed-up)**

Trade off: Overhead vs Speed



Clustering Index

Challenge: Index a sequential file on an *ordering*, *non-key* field.

e.g., create an index on EMPLOYEE ordered by DNO (dept. number)

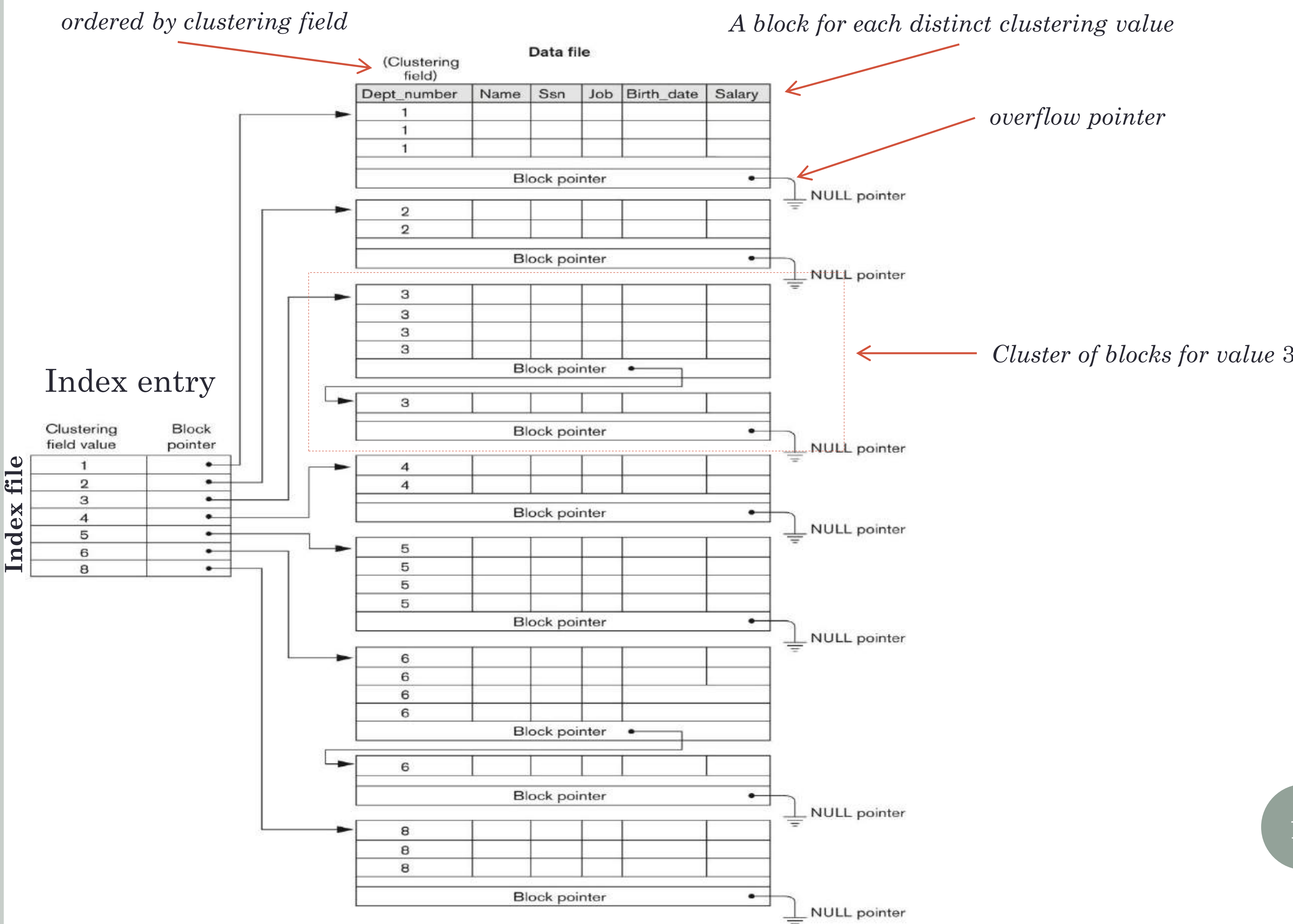
Idea: The file is a *set* of clusters of blocks; a *cluster* per *distinct* value:

index-entry := (**distinct-value, block-pointer**)

- One *index-entry per distinct clustering value*.
- *Block pointer* points at the *first* block of the *cluster*. The other blocks of the *same* cluster are contiguous and accessed via chain pointers.

Q3: Is the clustering index sparse *or* dense?

Clustering Index



Clustering Analysis

EMPLOYEE: r = 300,000 fixed-length records of size **R** = 100 bytes each; block size **B** = 4,096 bytes; DNO size = 9 bytes; P = 6 bytes (pointer), *ordered* by DNO.

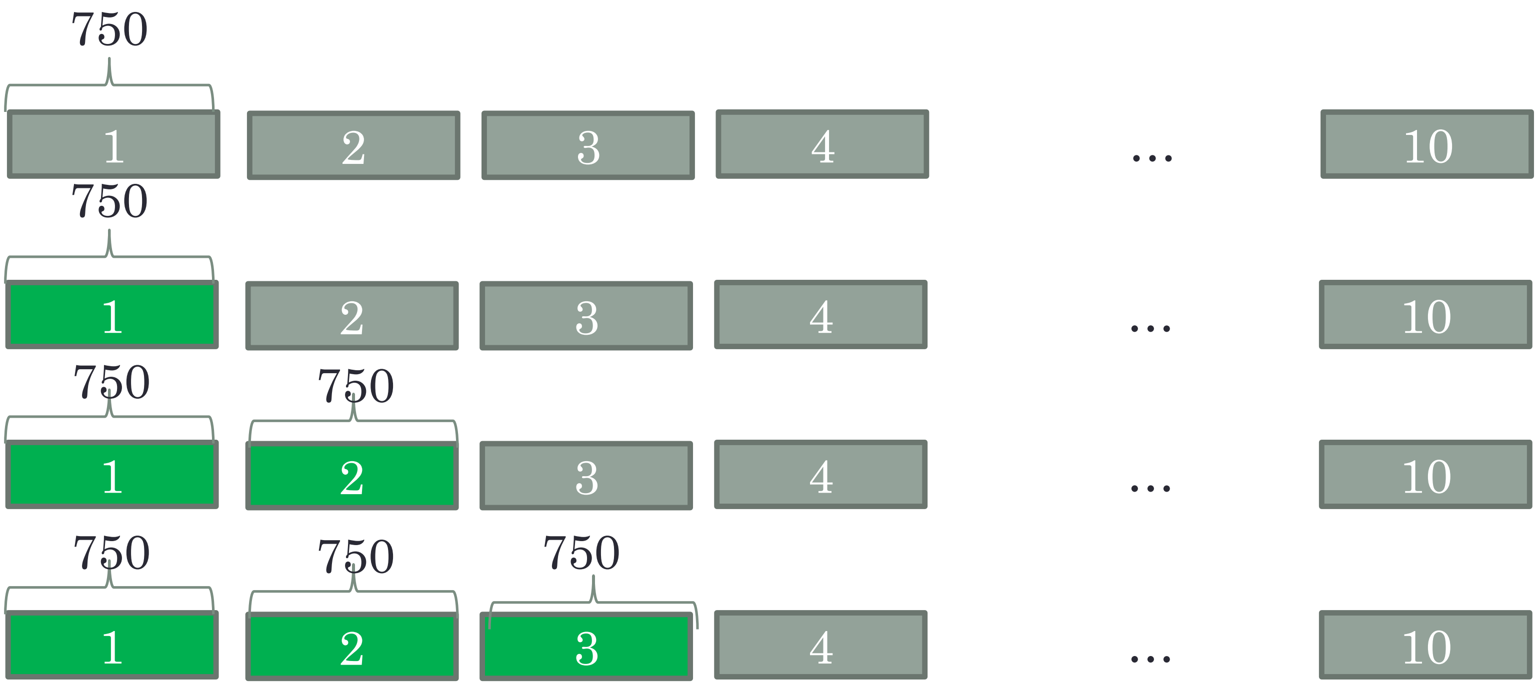
Context: 10 departments; DNO values are *uniformly* distributed over the tuples.
Tasks: **1.** Expected cost `SELECT * FROM EMPLOYEE WHERE DNO = x`
2. Compare with the linear search over a sorted file by non-key field (*exiting feature*).

- **Data File:** b = ceil(r/bfr) = **7,500 blocks**
- **Index Entry:** {DNO, Pointer}
- **Index Entry Size:** V = 9 bytes for DNO and P = 6 bytes for a Pointer.
- **Index Blocking Factor:** ibfr = floor(**B**/(**P+V**)) = 273 entries/block.
- **Index entries:** 10 index entries: one per cluster!
- **Index blocks**: ib = ceil(10/273) = **1 block.**

- **Overhead:** 1 block ☺ **(0.01% additional storage)**
- **Gain**: Search on Index: **1 block access**.
- We load 7500/10 = 750 data blocks belonging to a cluster.
- **Total: 1 + 750 = 751 block accesses.**
- Linear Search (*exiting feature*): **4125 block accesses and NOT 7500 block accesses!**

CLUSTERING ANALYSIS

Linear search over an *ordering* uniformly distributed *non-key field* (*exiting*)



1/10(750) + 1/10(750+750) + 1/10(3·750) + … + 1/10(10·750) = 4125

$$\sum_{k=1}^{n}\left(\frac{b}{n}\right)\left(\frac{1}{n}\right)k = \left(\frac{b}{n^2}\right)\left(\frac{n(n+1)}{2}\right) = \frac{b(n+1)}{2n}$$

n = number of clusters
b = number of blocks

$$\log_2(m) + \frac{b}{n}$$

Clustering index: m blocks

Decision Making

When we decide to create a Clustering Index.

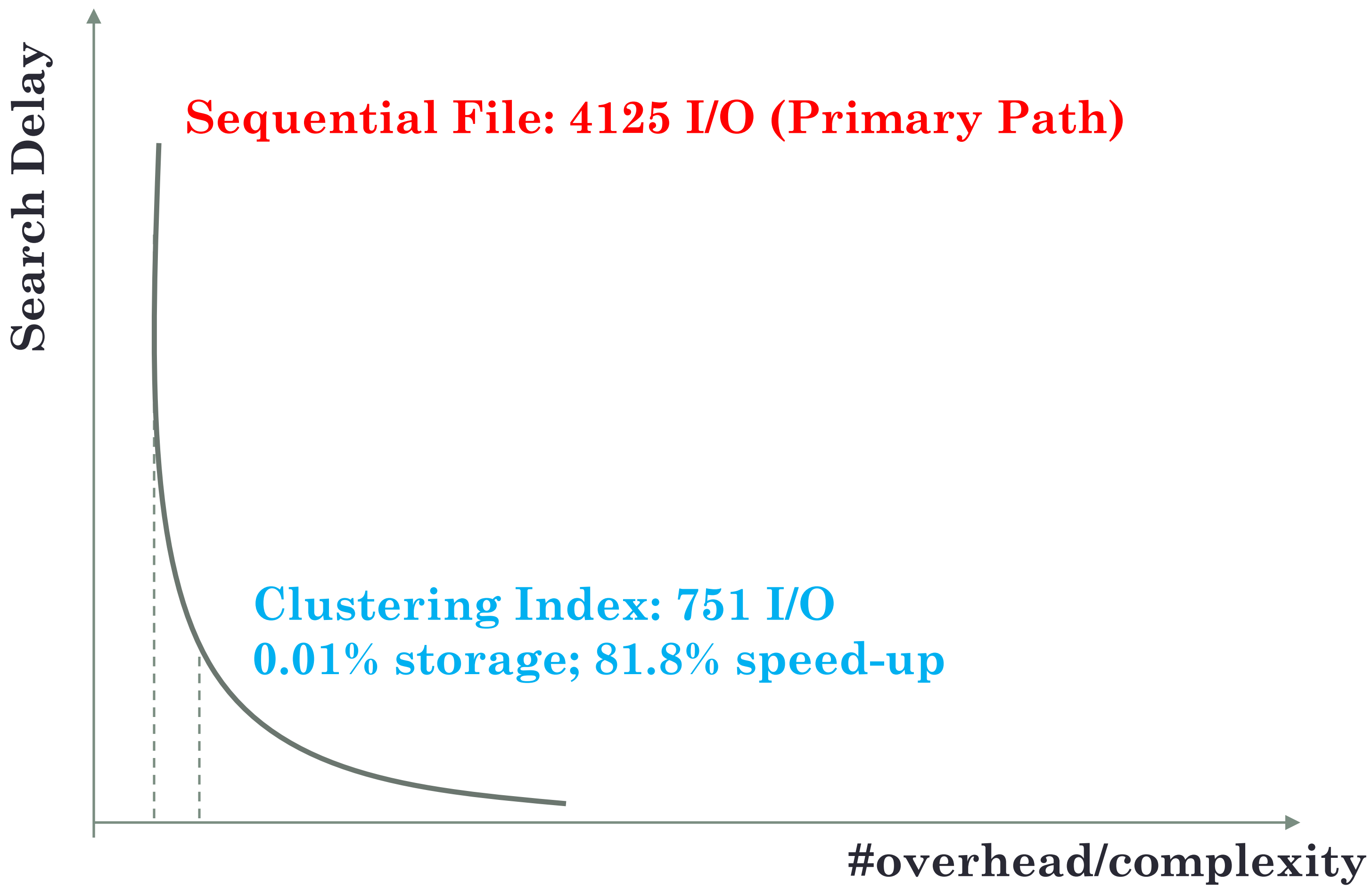
Theorem 1: *A Clustering Index of $m < b$ blocks is created over an ordering non-key field iff:*

$$m < 2^{\frac{b(n-1)}{2n}}$$

Theorem 2: *If $n \rightarrow \infty$, i.e., infinite number of distinct values, then the linear search over an ordering non-key field with exiting feature is bounded by $b/2$, i.e., half of the naïve linear search:*

$$\lim_{n \to \infty} \frac{b(n+1)}{2n} = \frac{b}{2} < b.$$

Trade off: Overhead vs Speed



Secondary Index

Challenge: Index a file on a *non-ordering* field. The file might be unordered, hashed, or ordered ***but not*** ordered w.r.t. the indexing field.

Cases:

- **[S1]** Secondary Index on a *non-ordering*, *key* field; e.g., SSN
- **[S2]** Secondary Index on a *non-ordering, non-key* field; e.g., DNO

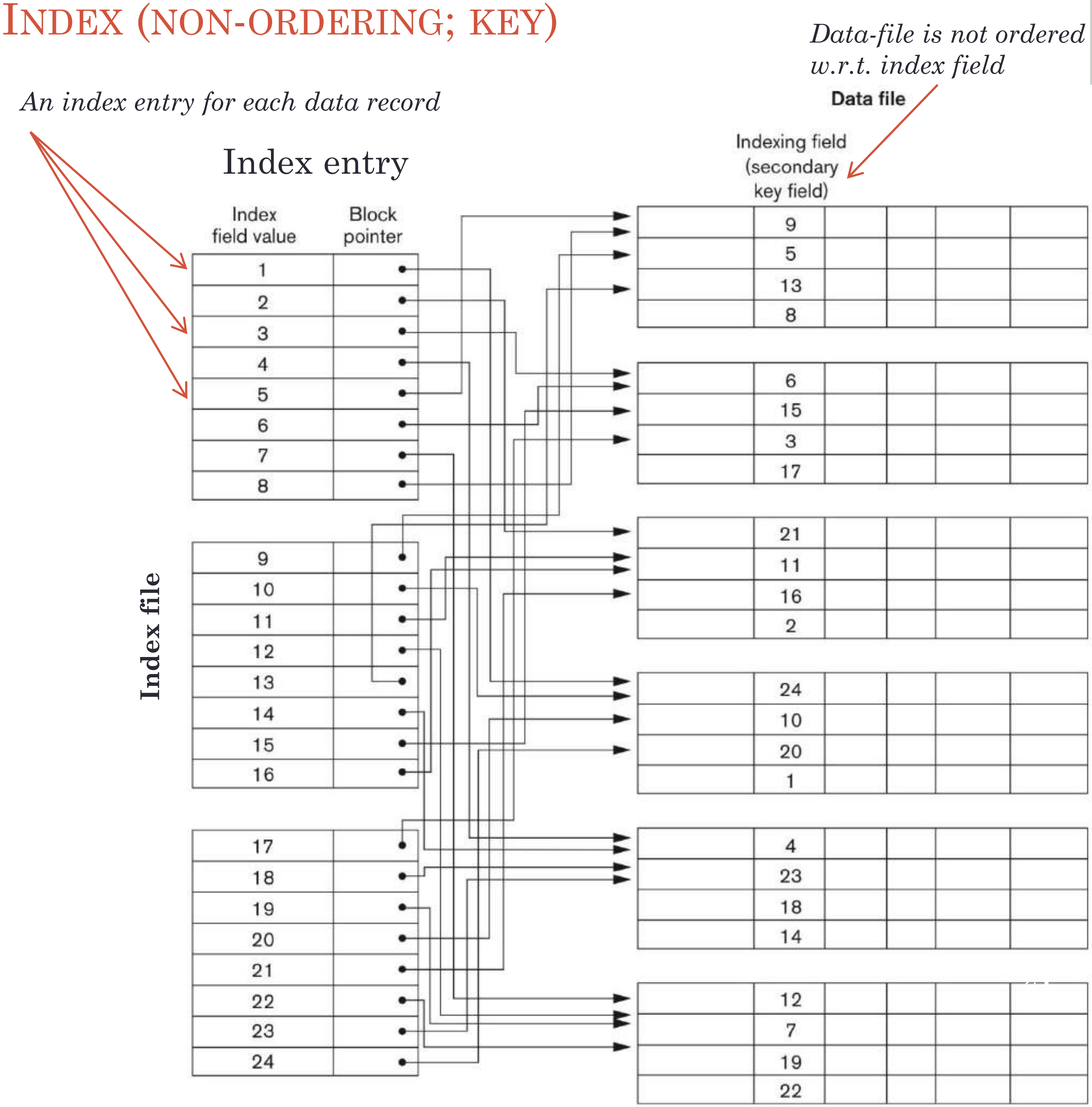
[S1]: One index entry *per* data record, i.e., *dense* index.

Why?

Because: the file is not ordered according to the indexing field, thus, we cannot use anchor records;

index-entry := (**index-value, block-pointer**)

[S1] Secondary Index (non-ordering; key)



In-Class Activity [A1]

Task: Secondary Index on a **non-ordering; key** attribute: SSN

File: r = 300,000 records; **R** = 100 bytes; block size **B** = 4.096 bytes;

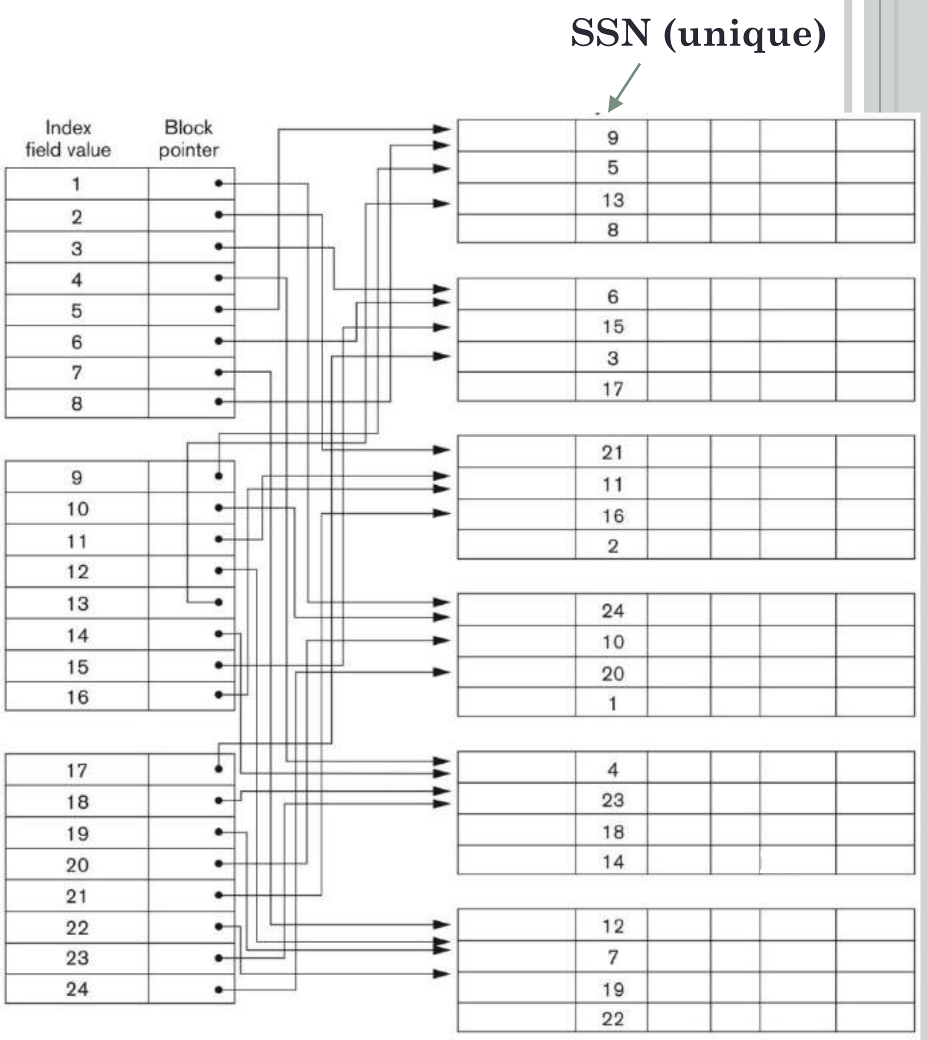
SSN is V = 9 bytes, pointer P = 6 bytes.

Cost: `SELECT * FROM EMPLOYEE`

`WHERE SSN = x`

Context:

- Blocking factor *bfr* = 40 records per block;
- File blocks: ***b* = 7,500 data blocks**.



In-Class Activity [A1]

- Step 1: Index entry: V + P = 9 + 6 = **15 bytes.**
- Step 2: Index Blocking factor: *ibfr* = floor(B/(V+P)) = **273 entries/block**.
- Step 3: Secondary Index is *dense*: *r* = 300,000 index entries
- Step 4: Index File blocks *ib* = ceil(*r* / *ibfr*) = **1,099 blocks (14% overhead)**

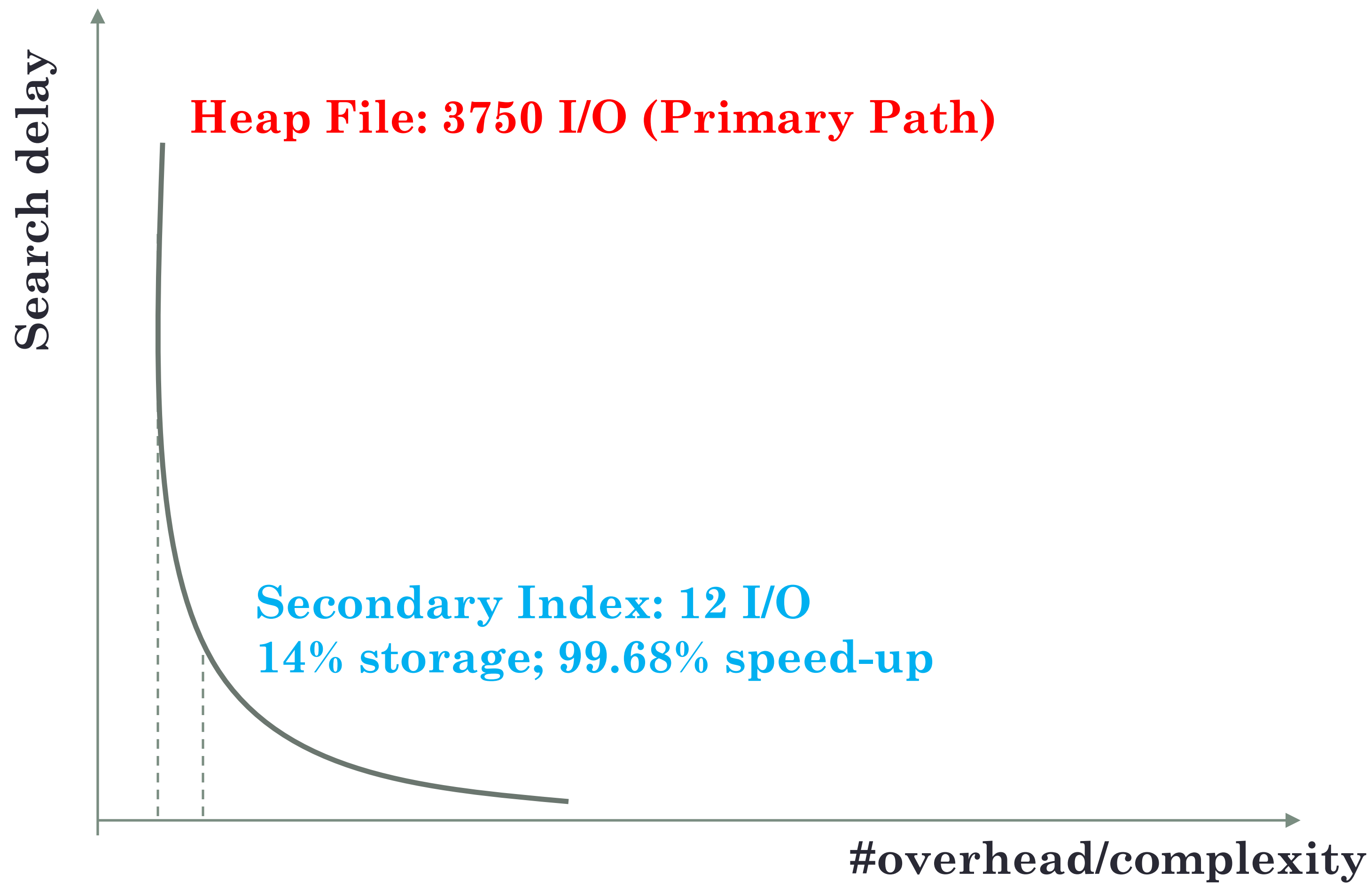
Cost: Binary Search on Index: ceil($\log_2(ib)$) = **11 block accesses.**

One more block access to load the *unique* block pointed by the index-entry:

Total: 11 + 1 = 12 block accesses

Serial Search on File: *b*/2 = **3,750 block accesses (99.68% speed-up)**

Trade off: Overhead vs Speed



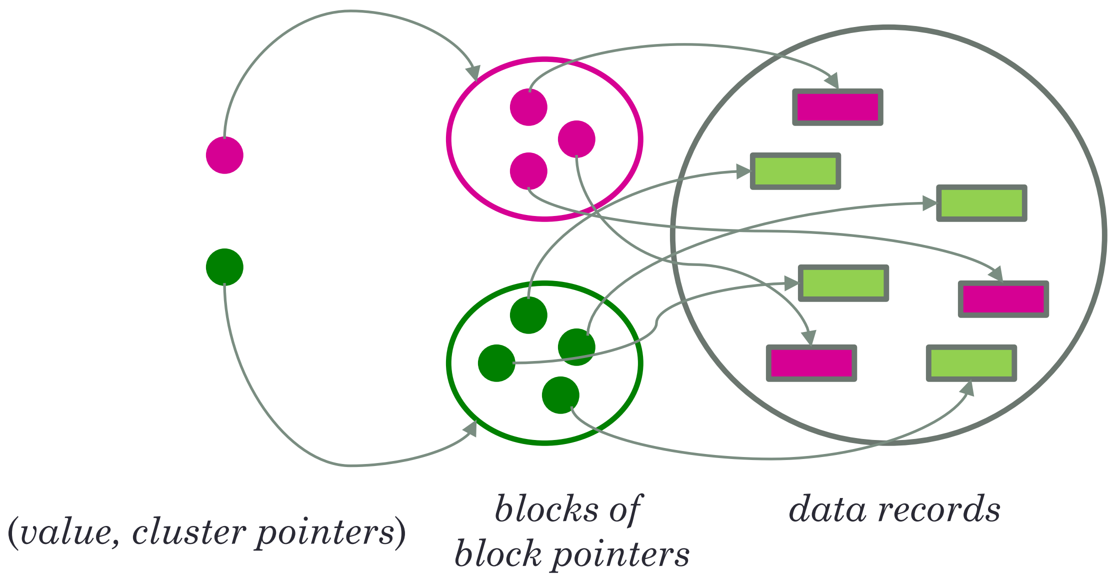
Secondary Index

[S2]: Indexing field is a *non ordering, non key*.

Idea 1: *group* the *block addresses* of those records having the *same* value.

Idea 2: *assign* an index entry *per* group (cluster) of block addresses

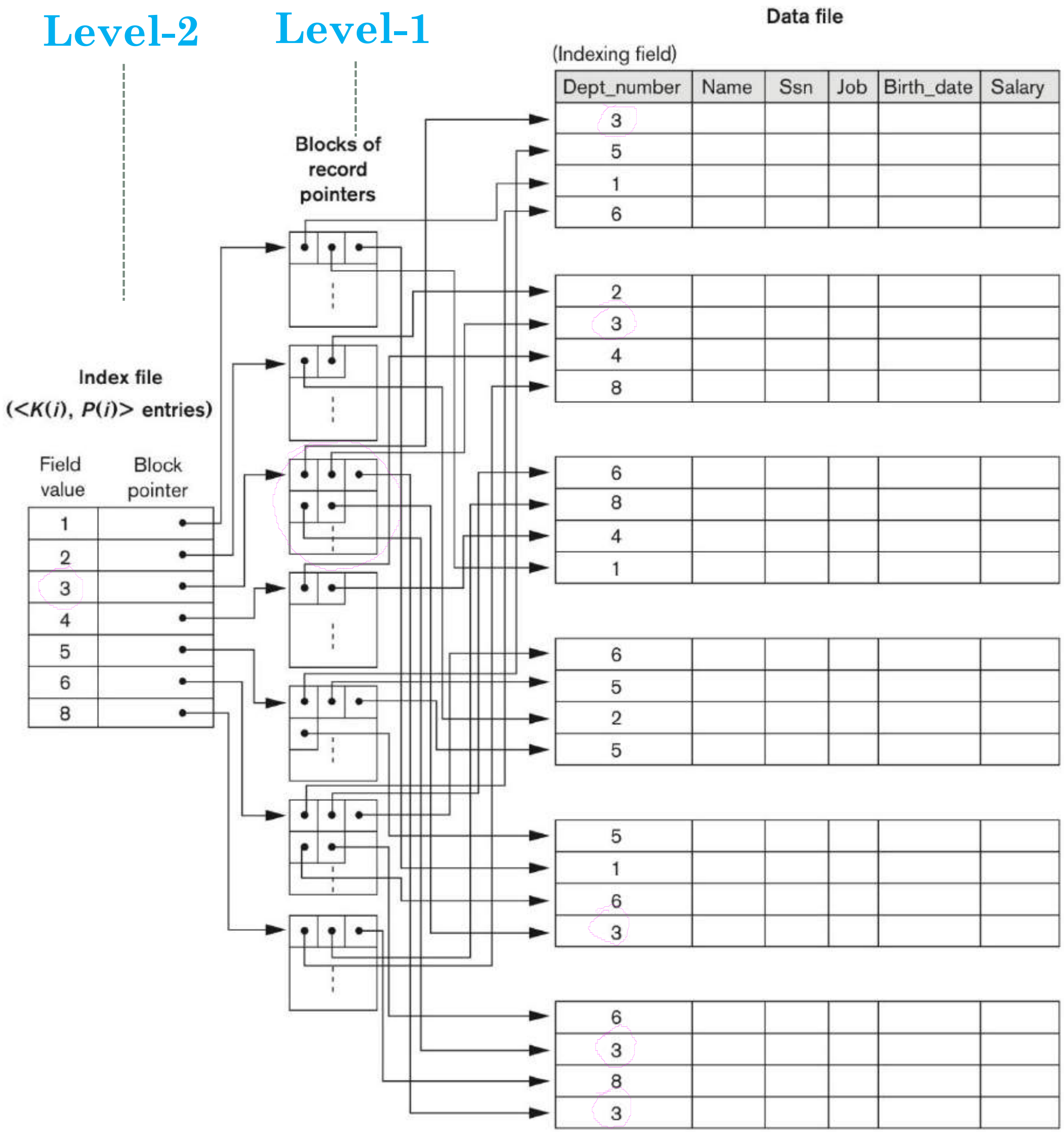
index-entry := (**distinct-value, cluster-pointer**)



A **cluster-pointer** points to (2 *levels of indirection*):

- (**Level 1**) a *block* of {**block-pointers**} of a *cluster*,
- (**Level 2**) a **block-pointer** points to the **data-block** that has records with this *distinct* index value.

[S2] Secondary Index (non-ordering; non-key)



Note:

- Index is *sparse*.
- One entry *per* cluster.
- Level 1 is a *set* of blocks;
- Each Level-1 block contains records of block pointers.

Search for DNO = 3:

1. Binary search in Level-2 (1)
2. Direct access to Level-1 (1)
3. Load *all* the corresponding data-blocks (4)

Total:

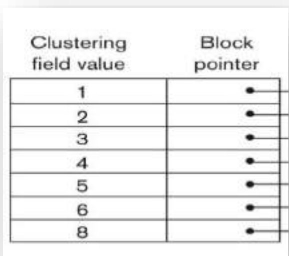
1+1+4 =6 block accesses

Serial Scan: *b* = 7500 block accesses

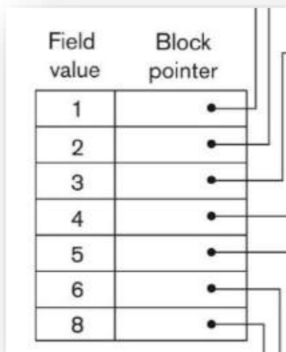
Gain: 99.92% speed-up

MULTILEVEL INDEX

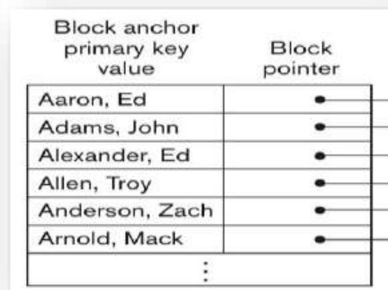
- **Observation**: in *all* index files it holds true that:
  - they are *ordered* on the indexing field;
  - the indexing field has *unique* (*distinct*) values;
  - each index entry is of *fixed* length;



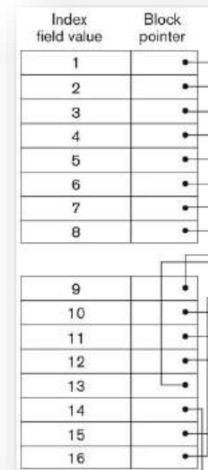
Clustering



Secondary [S2]

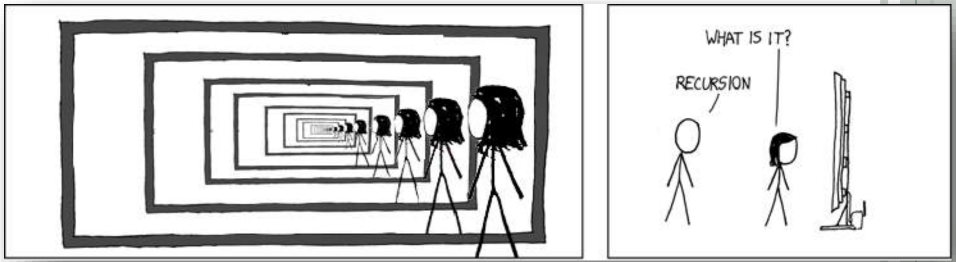


Primary



Secondary [S1]

Conclusion: we can build a *primary index* over any *index* file, since it is an ordered file w.r.t. a key field (*index of an index*)



Multilevel Index

- *index of an index* refers to the *multilevel* index:
  - the original index file is referred to as the **base** or **Level-1 index,**
  - the *additional* index is referred to as **Level-2 index** (*index of an index*)
  - …
  - *if we repeat this to level* > 2 *we obtain*…**Level-*t* index**, *i.e., index of an index of an index …*

Challenge: Find the *best* level *t* of a multi-level index to expedite the search process *trading off* speed-up with overhead.

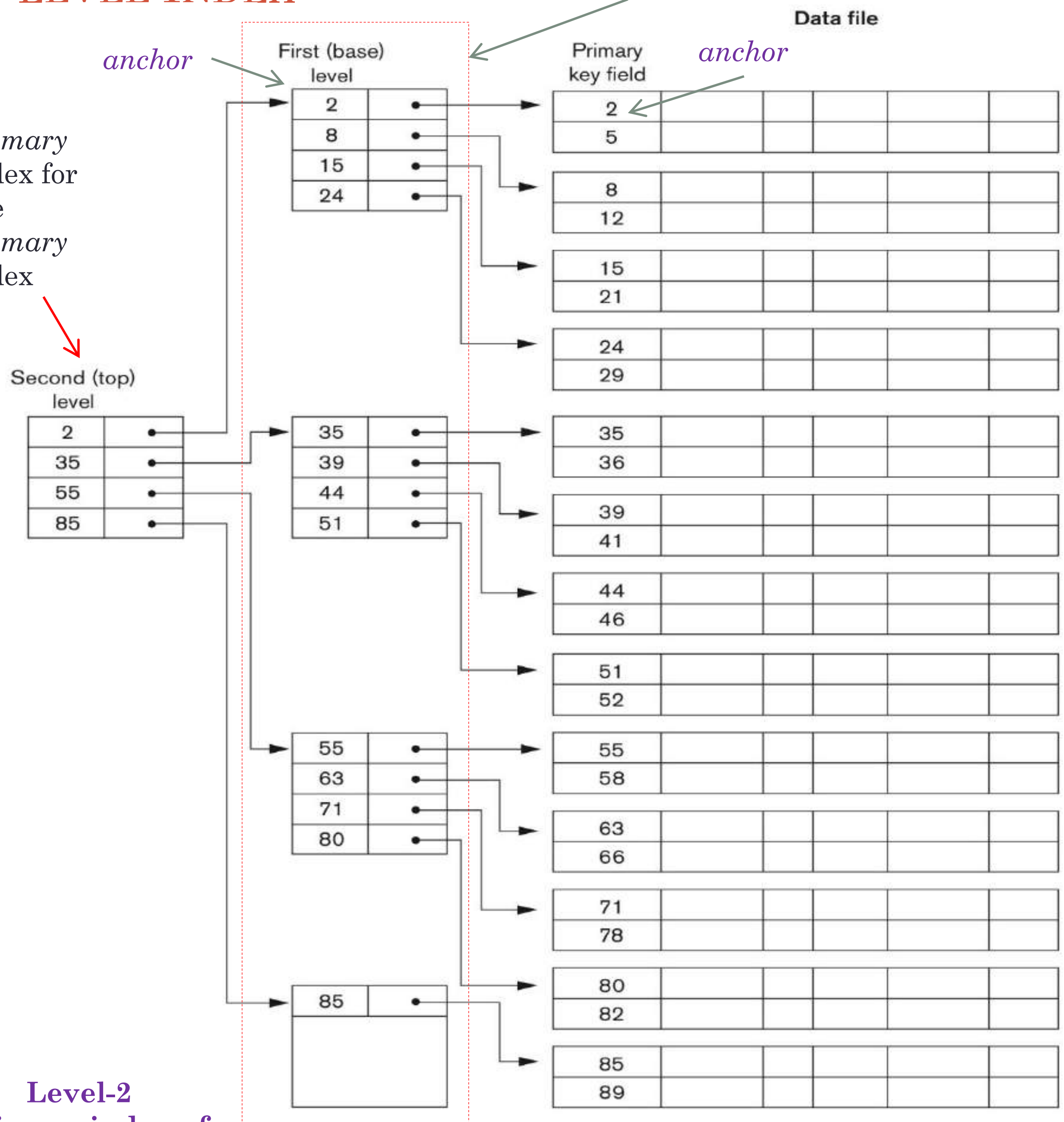
2-LEVEL INDEX

treat this as a file (ordered by key)

anchor

anchor

primary index for the *primary* index

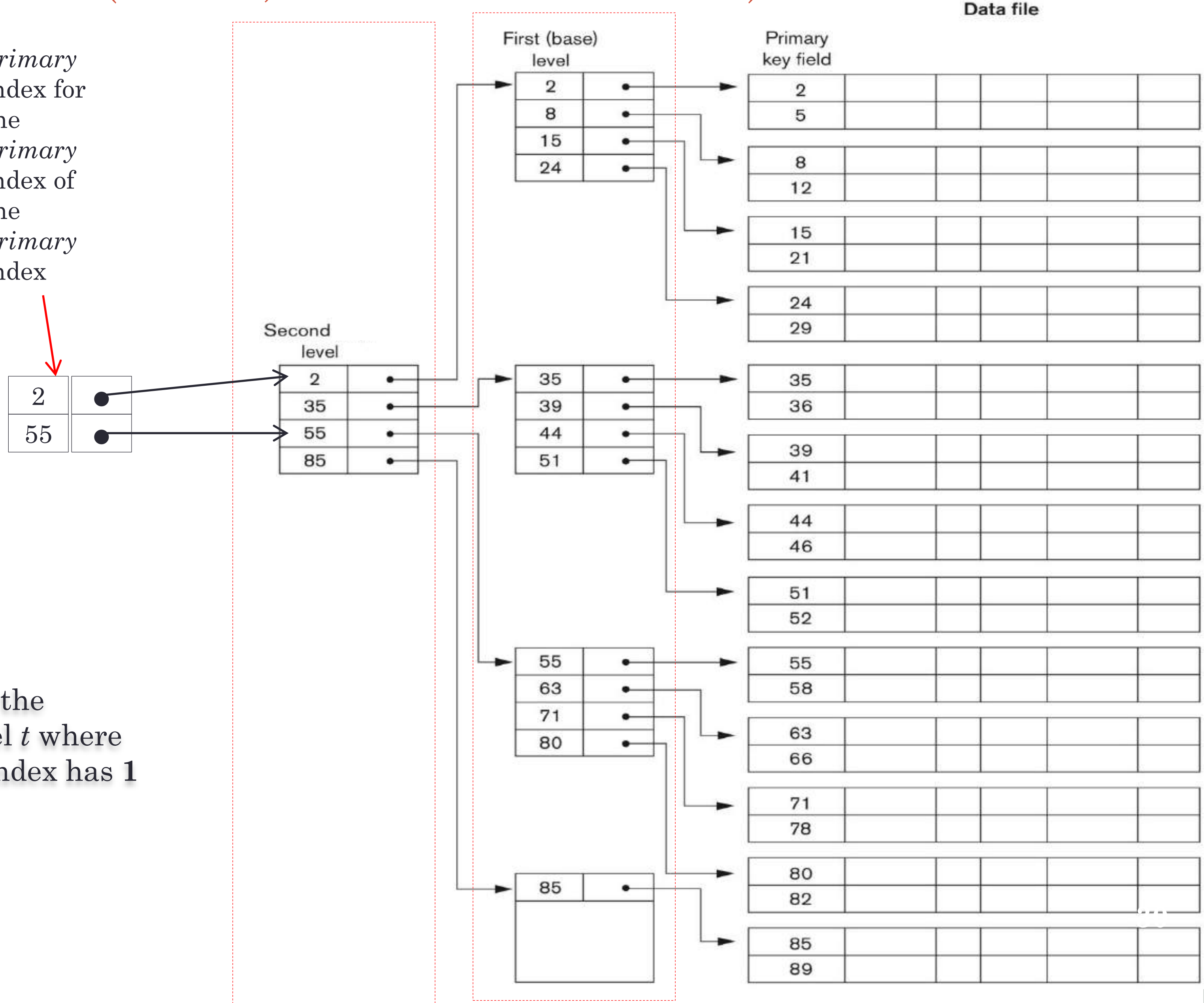


Level-2 (primary index of the index)

Level-1 (primary index of the data-file)

3-LEVEL INDEX (TRIVIAL; ONLY FOR ILLUSTRATION)

primary index for the *primary* index of the *primary* index



Hmm: stop at the *minimum* level *t* where the top-level index has **1** block.

Level-3 **Level-2** **Level-1**

MULTILEVEL INDEX: REASONING

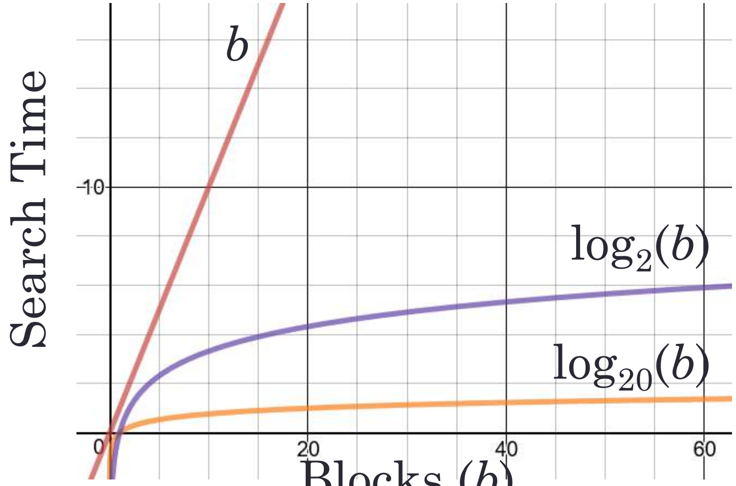
Logarithms by John Napier (1550-1617) in Scotland (Uni St Andrews)

Idea: $\log_m(b)$ with $m > 2$ splits the search space into m sub-spaces until finding the *unique* block!

- Splitting steps: $\log_m(b) < \ldots < \log_2(b)$

Theorem 3: *Given a Level-1 Index with blocking factor m entries/block, the multi-level index is of maximum level $t = \log_m(b)$.*

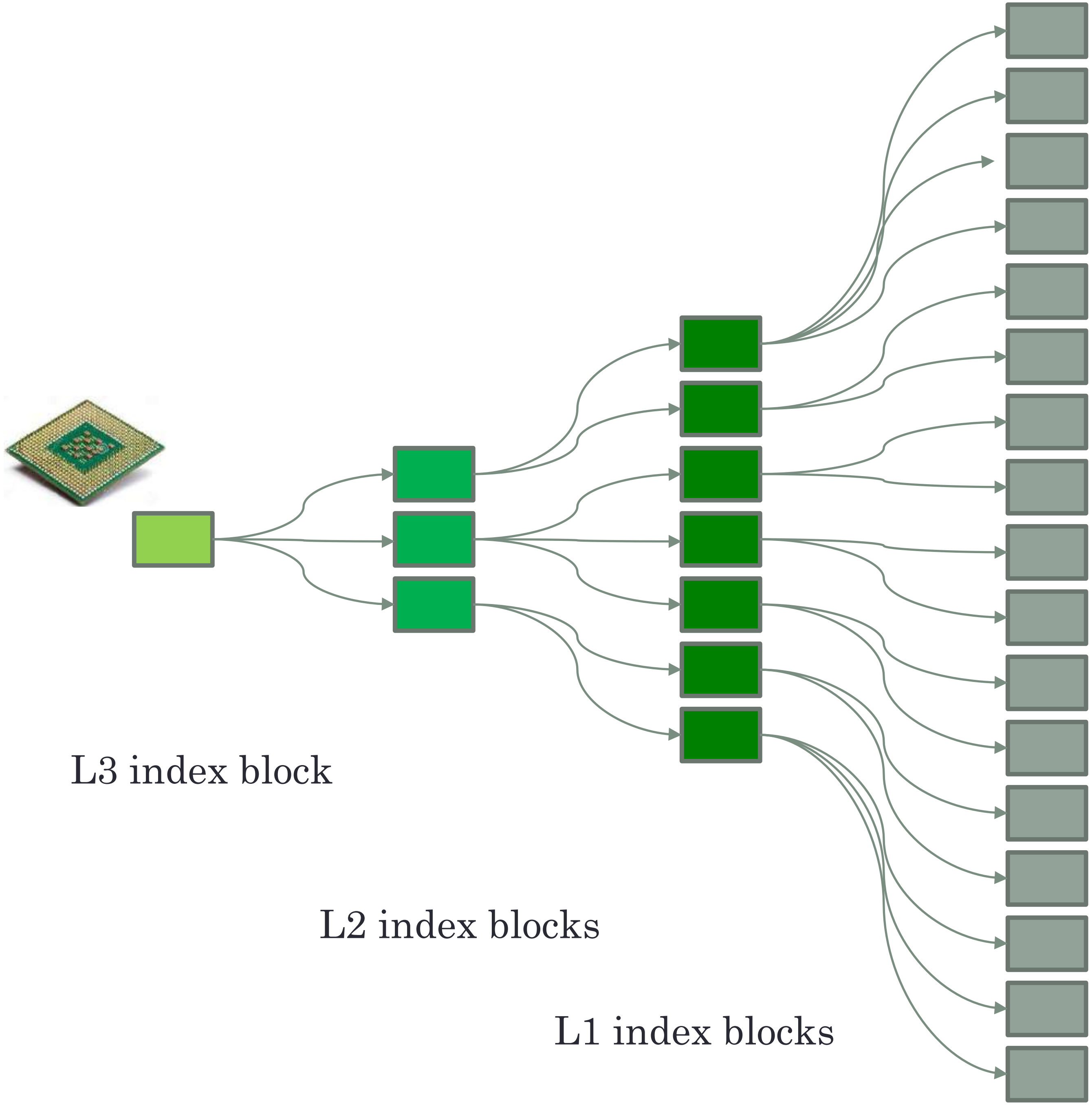
m is known as **fan-out**



Scalable Design: *independent* of data-size!

Welcome to the Big Database Systems!

Fact: In *any* SQL/NoSQL System, always *adopt* multi-level indexes for data access!



L3 index block

L2 index blocks

L1 index blocks

data file blocks

In-Class Example [E2]

Amazing Gain: one block/level plus *one* data-block:= $\boldsymbol{t+1 = \text{ceil}(\log_m(b)) + 1}$ **block accesses**.

Build a *non-ordering/key secondary multi-level index* over a file: $\boldsymbol{b}$ **= 7,500 blocks** and $\boldsymbol{r}$ **=300,000 records** (liaise with in-class activity A1)

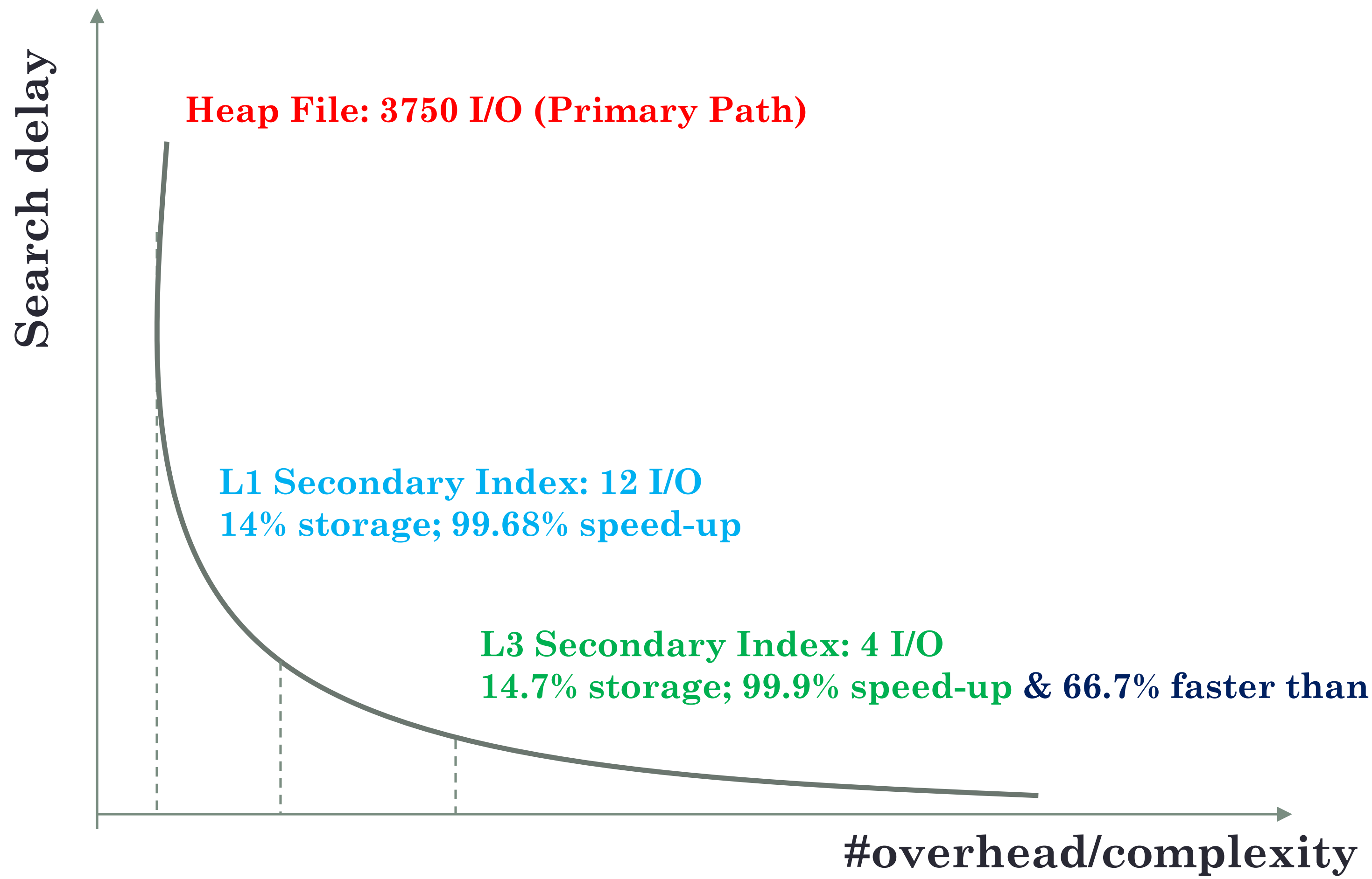
- Fan-out (*index-blocking factor*) $\boldsymbol{m}$ **= 273** index entries/block
- **Level-1** index with $\boldsymbol{b1}$ **= 1,099** index-blocks.
- **Level-2:** index entries: **1,099** thus, number of blocks $\boldsymbol{b2 = \text{ceil}(b1/m)}$ **= 5 blocks;**
- **Level-3:** index entries: **5** thus, number of blocks $\boldsymbol{b3 = \text{ceil}(b2/m)}$ = **1 block;**

Structure := L1: Secondary Index (dense); L2 & L3: Primary Indexes (sparse)

3-level index search cost: **3 + 1 =** **4 block accesses only!**

- Level-1 Secondary Index $\boldsymbol{\text{ceil}(\log_2(1099)) + 1}$ **=** **12 blocks accesses**
- No Index: **3750 blocks accesses**.

Trade off: Overhead vs Speed



Proof of Theorem 3

Block size **B** bytes; File with ***r*** records; data-record has size ***s***.

- *Blocking factor* for the data-file is ***f* = floor(B/*s*)** records/block
- Data file is ***b* = ceil(*r* / *f*)** data-blocks.

Level-1 Primary index: each index entry points to each file-block (*anchor*).

- Let ***l*** be the *size* of the index entry
- The Level-1 index **has *b* entries**, with *blocking factor* ***m* = floor(B/*l*)**
- The Level-1 index has ***b*1 = ceil(*b* / *m*) L1-index-blocks**

Level-2 Primary index: each index entry points to each index-block of Level-1.

- The Level-2 index has *b*1 entries, with *blocking factor* ***m* = floor(B/*l*)**
- The Level-2 index has ***b*2 = ceil(*b*1 / *m*) = ceil(b / m^2) L2-index-blocks** (1/m^2 less blocks)

…

Level-*t* primary index: The *t*-th **top** level will have only **1 block** thus **$1 \leq (b / m^t)$** or **$t = \log_m(b)$**.

- Split the searching space into *m* sub-spaces thus approx. *t* steps to find the desired block.

```
SELECT * FROM EMPLOYEE WHERE DNO >= 3
```

Step 1: Expected cost $\mathbf{log_2(\mathit{m})}$ **block accesses for DNO = 3.**

Step 2: Expected cost **(*b*/*n*) block accesses for DNO = 3.**

Step 3: L := number of extra *clusters* to be retrieved (e.g., L = 7).

Step 4: Expected cost **L·(*b*/*n*) block accesses for DNO > 3.**

Step 5: Total: $\mathbf{log_2(\mathit{m}) + (L+1) \cdot (\mathit{b}/\mathit{n})}$ **block accesses.**

Linear Search

$$\sum_{k=1}^{3}\left(\frac{b}{n}\right) + \sum_{k=3+1}^{n}\left(\frac{b}{n}\right) = b$$

Benefit: index-cost < linear-cost: $\log_2 m < \dfrac{b(n-(L+1))}{n}$

Indexing Methodology
Part II

Database Systems (H)

Dr Chris Anagnostopoulos

ROADMAP

- Multi-Level Index *implementations* (*from Theory to Practice*)
  - B Tree (*better* version); Boeing Research Lab (1972)
  - B+ Tree (*current* popular version) IBM®

- **Use Case:** Secondary Index on *non-ordering, key* field
- Put-All-Together: Primary Access Path, Secondary Index, B+ Tree
  - Examining the trade-off: Speed *vs* Overhead

Multilevel Index

Recall: Search for a record over a t-level index: ***t* + 1 block accesses**
Challenge: Insertions, deletions, and updates are costly! [*]

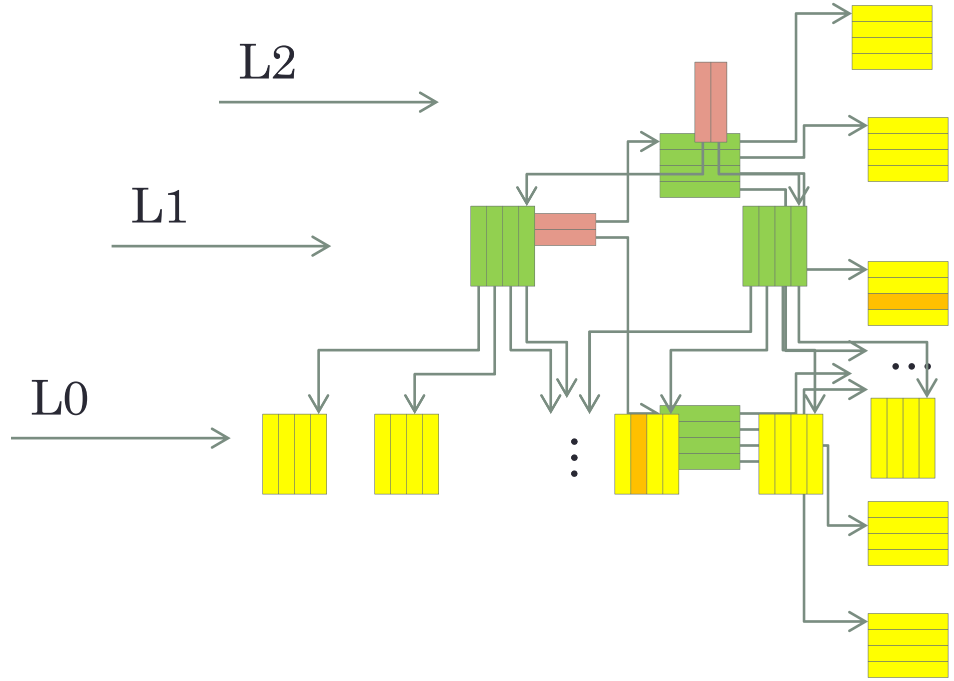
Why?
Because: *all* encapsulated indexes are *physically* ordered files. Hence, *all* updates should be reflected to *all* levels.

Challenge: define *dynamic* multi-level indexes:
- *adjust* to *deletions, insertions* of records
- *expand* and *shrink* following the distribution of the index values
- *be* self-balancing (sub-trees of same depth)

Proposal: B Tree; B+ Tree; B* Tree; *k-d* Tree (d-dimensional space)

[*] Larson, Per-Åke (1981). *Analysis of index-sequential files with overflow chaining*. ACM Transactions on Database Systems. 6 (4).

Multilevel Index as a Tree



Observe: 2-level multilevel index over a *non-ordering*, *key* (secondary index)

Turn: clockwise by 90 degrees

View: A Tree structure:

- **Root** is the **L2**-Index
- Root's **children** are blocks of the **L1**-Index
- **Leaves** are actual data blocks (**L0**)

LIMITATION

It becomes **unbalanced**;

It does not *adjust* to keys' *distribution,* i.e., *leaf-nodes* are at different levels…

- **Worst case**: a *linked-list* of nodes *instead* of a *tree* structure
- *Larger* tree depth t results to *higher* expected search time **O(t)**;

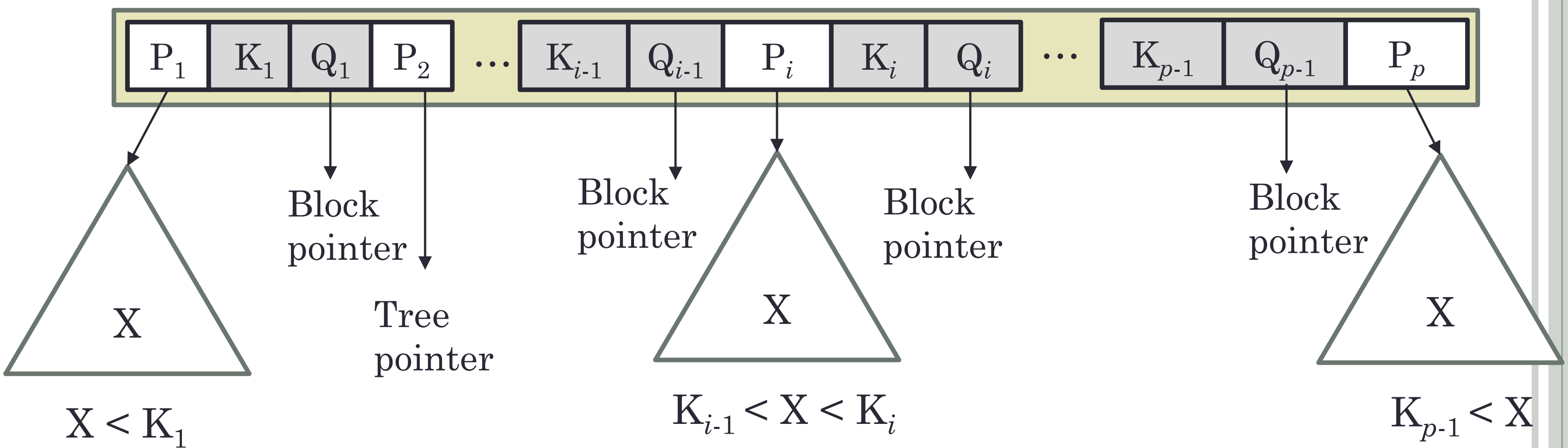
Challenge 1: ensure **balanced tree** by *minimizing* the tree depth t;

- **Challenge 1.1:** *what* happens if key values are inserted in a *full* node? (**split**)
- **Challenge 1.2:** *what* happens if the *key value* in a node is deleted? (**merge**)

B-Tree: Index on Non-ordering Key

B-Tree Node order *p*: splits the searching space up to p subspaces; $p > 2$

$$\textbf{Node} := \{\mathbf{P_1}, (K_1, \mathbf{Q_1}), \mathbf{P_2}, (K_2, \mathbf{Q_2}) \ldots, \mathbf{P_{p-1}}, (K_{p-1}, \mathbf{Q_{p-1}}), \mathbf{P_p}\}$$

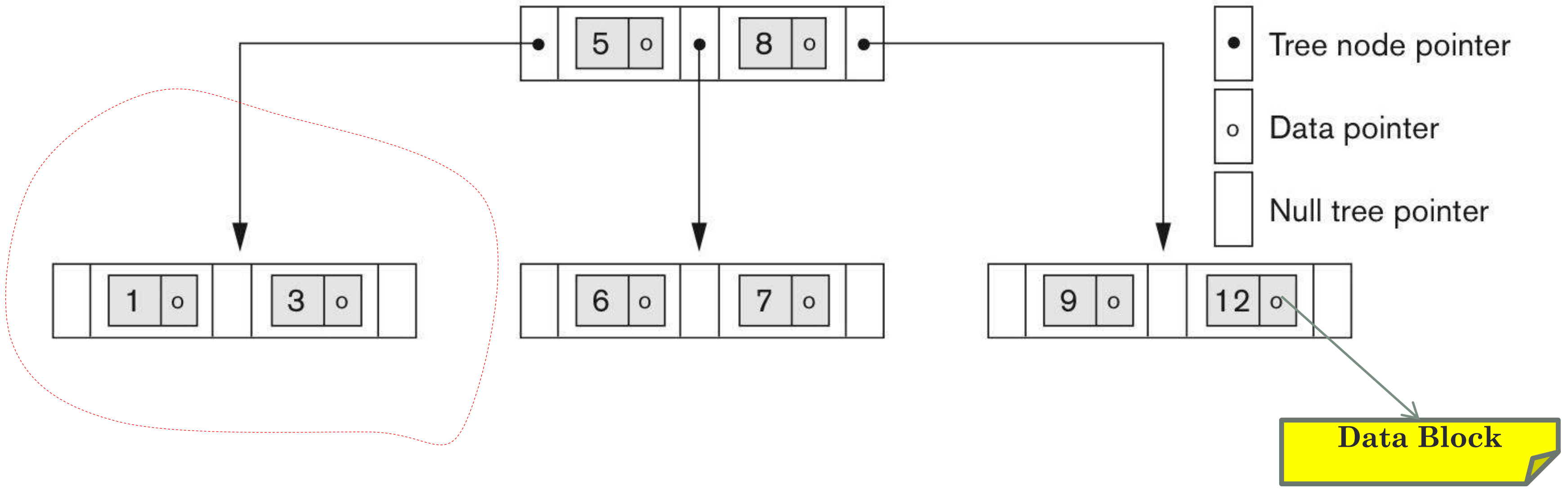


- key values sorted: $K_1 < K_2 < \ldots < K_{p-1}$
- block/data pointer Q_i ponits to the *data-block* holding the value K_i
- tree pointer P_i points to a sub-tree of key values **X**:
  - If $i = 1$, $X < K_1$ and if $i = q$, $K_{q-1} < X$
  - $K_{i-1} < \mathbf{X} < K_i$, for $1 < i < q$

B-Tree

Search: Traversing the tree nodes until finding the key value
Rationale: Immediate access to the block of the searching key!

B-Tree node order p = 3 (**balanced tree**)



Normally: 1 Tree Node *per* block

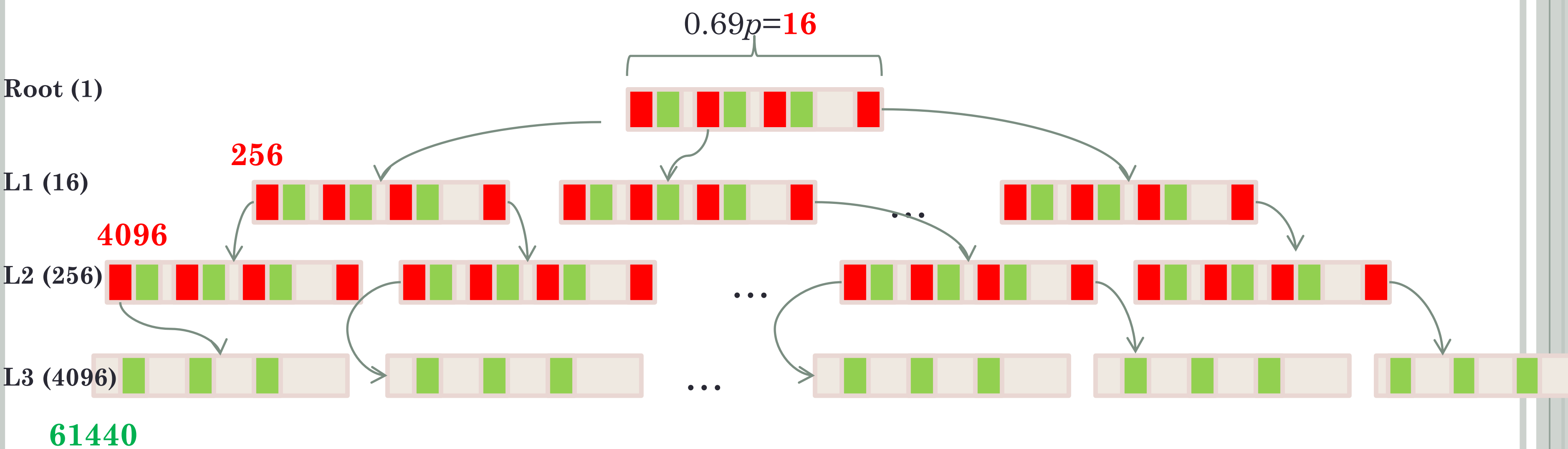
Search 8: 2 block accesses; access the data block *immediately*
Search 7: 3 block accesses; access the data block *immediately*
Search 12: 3 block access *plus* data block access
Search 31: 2 block access; no record thus *no need* to load the block and check!

IN-CLASS EXAMPLE [E1]

Task 1: Create a **3-level B-Tree index** of order ***p* = 23** over a *non-ordering, key field*

Context: Each B-Tree node is 69% *full* of information (pointers/keys).

- On average, each B-Tree node accommodates 0.69*p* = **16 tree pointers/ 15 key values**.
- Average **fan-out** = 16 per tree node, i.e., split the tree space into **16 sub-trees**.



Root: 1 node with **15** keys/data-pointers;**16** pointers to tree nodes;

Level-1: 16 nodes with 16*15=**240** keys/data-pointers;16*16 = **256** pointers to nodes;

Level-2: 256 nodes with 256*15=**3840** keys/data-pointers; 256*16 = **4096** pointers to nodes;

Level-3: 4096 nodes with 4096*15 = **61440** keys/data-pointers; and **null** pointers (leaves);

In-class Example [E1]

Task 2: How many keys can we store?

Structure:

Root: 1 node with **15** keys/data-pointers;**16** pointers to tree nodes;

Level-1: 16 nodes with 16*15=**240** keys/data-pointers;16*16 = **256** pointers to nodes;

Level-2: 256 nodes with 256*15=**3,840** keys/data-pointers; 256*16 = **4096** pointers to nodes;

Level-3: 4096 nodes with 4096*15 = **61,440** keys/data-pointers; and **null** pointers (leaves);

We *can* store: **61440** + **3840** + **240** + **15** = **65,535** key entries pointing to data blocks.

Q: What if our file has **65,536** keys to be indexed?

A: New level: 4096*16 = 65536 tree nodes storing: 983,040 keys (for storing just one extra value!)

The index can store now: **1,048,575 values…*redundancy* (93.3% of leaf nodes space is empty)**

Reflect: does the value of order *p* cause this redundancy?

Challenge: given a file, which is the *best* order value *p* to avoid redundancy?

In-class Example [E1]

Task 3: Block **B = 512** bytes, **data-pointer Q** = 7 bytes,
tree-pointer P = 6 bytes, **key V** = 9 bytes:

(*i*) How many bytes is the index?

(*ii*) How many blocks is the index?

Storage (*i*)

Storage for data-pointers = **65,535*7** = 458,745 bytes (**460KB**)

Storage for key entries= **65,535*9** = 589,815 bytes (**590KB**)

Storage for tree-pointers = (**4096+256+16**)***6** = 4368*6 = 26,208 bytes (**27KB**)

Total storage: 1,074,768 bytes = **1.07MB** index (*only* meta-data!)

Blocks (*ii*)

Tree node size: 16***6** + 15***7** + 15***9** = **336 bytes**;

Blocking factor: floor(512/336) = **1 tree-node per block (*normally*)!**

Number of nodes: **1 + 16 + 256 + 4096** = **4369** nodes, thus, **4,369 blocks!**

MAXIMIZE FAN-OUT & MINIMIZE STORAGE

Hmmm, a B-Tree stores too much meta-data:

- *data-pointers* to blocks (*addresses, e.g.*, URI/L *average* size ~1KB!);
- *tree-pointers* to tree nodes (*structural meta-data*);
- *search key* values (*data values*);

Objective 1: be more *storage* efficient…*free* up space from the nodes; **how?**

Objective 2: be more *search* efficient…maximize the *fan-out* of a node; **how?**

Recall: *fan-out* is the splitting factor of the search-space

Thought: *fan-out* is the node order *p, i.e.*, number of *tree-pointers per* node.

Thus: *maximize* the number of tree-pointers per node to *maximize* fan-out!

Thus: *maximize* the blocking factor by squeezing more tree-pointers

Thus: *remove* the data-pointers from the tree nodes!

B^+ Tree: Index on Non-ordering Key

Ta-dah: B^+ Tree (*give* semantics to the nodes!)

- **Internal Nodes;** guide the searching process (*super-fast*)
- **Leaf Nodes;** point to actual data blocks (*access point*)

Principle 1: Internal Nodes have *no data-pointers to maximize fan-out*.

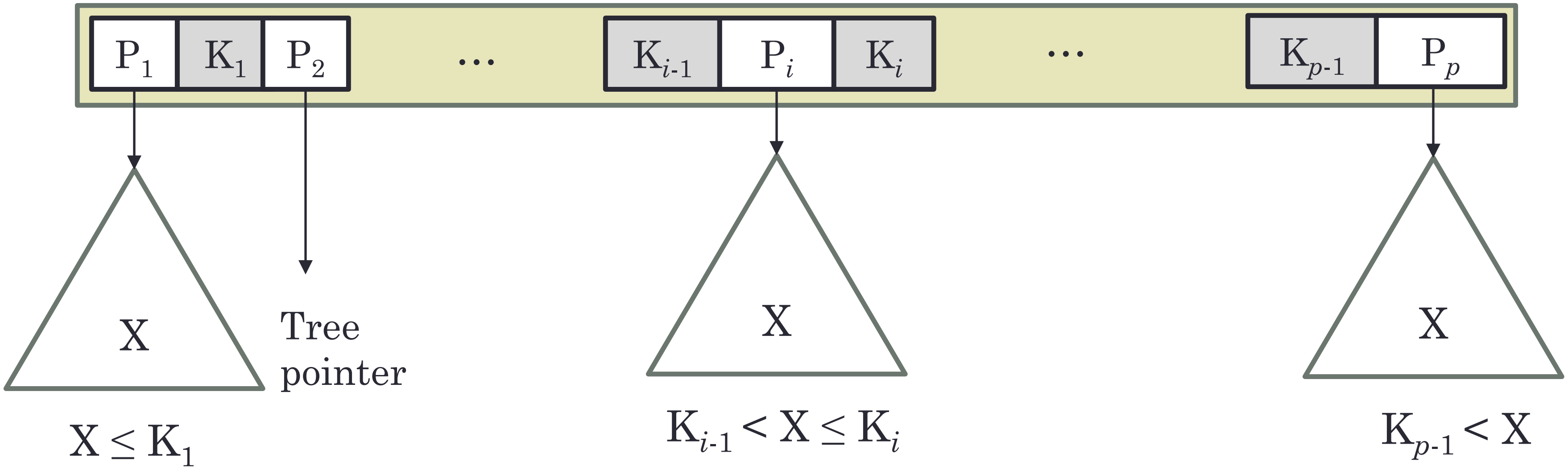
Principle 2: *only* Leaf Nodes hold the *actual* data-pointers.

Principle 3: Leaf Nodes hold ***all*** *the key values sorted* and their *corresponding* data-pointers.

Principle 4: *Some* key values are *replicated* in the Internal Nodes to *guide* & *expedite* the search process ☺ (corresponding to *medians of key values in sub-trees*).

B^+ Tree: Internal Node

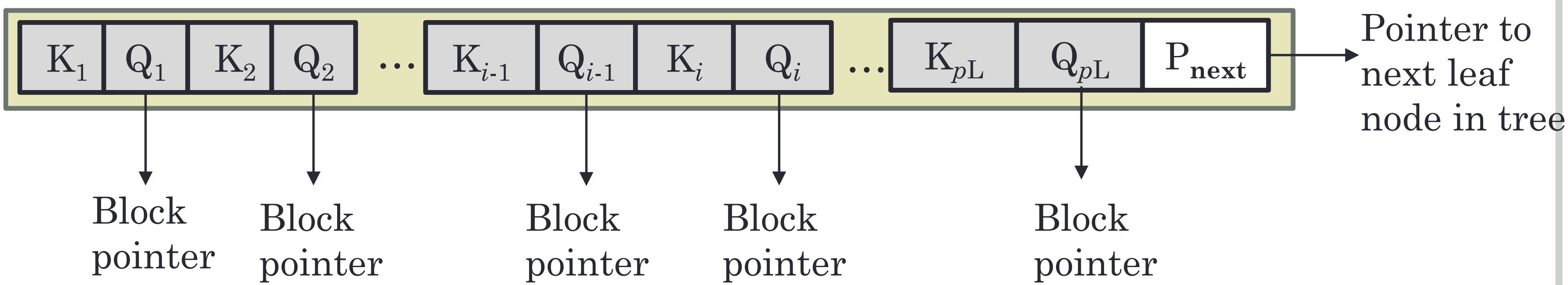
B+Tree Internal Node order *p* := {$\mathbf{P_1}$, K_1, $\mathbf{P_2}$, K_2, …, $\mathbf{P_{p-1}}$, K_{p-1}, $\mathbf{P_p}$}



- key values sorted: $K_1 < K_2 < \ldots < K_{p-1}$
- tree pointer P_i points to a sub-tree of key values **X**:
  - If $i = 1$, $X \leq K_1$ and if $i = q$, $K_{q-1} < X$
  - $K_{i-1} < \mathbf{X} \leq K_i$, for $1 < i < q$
- **Note:** Each internal node with ***p* tree-pointers** has ***p*-1 key values**.

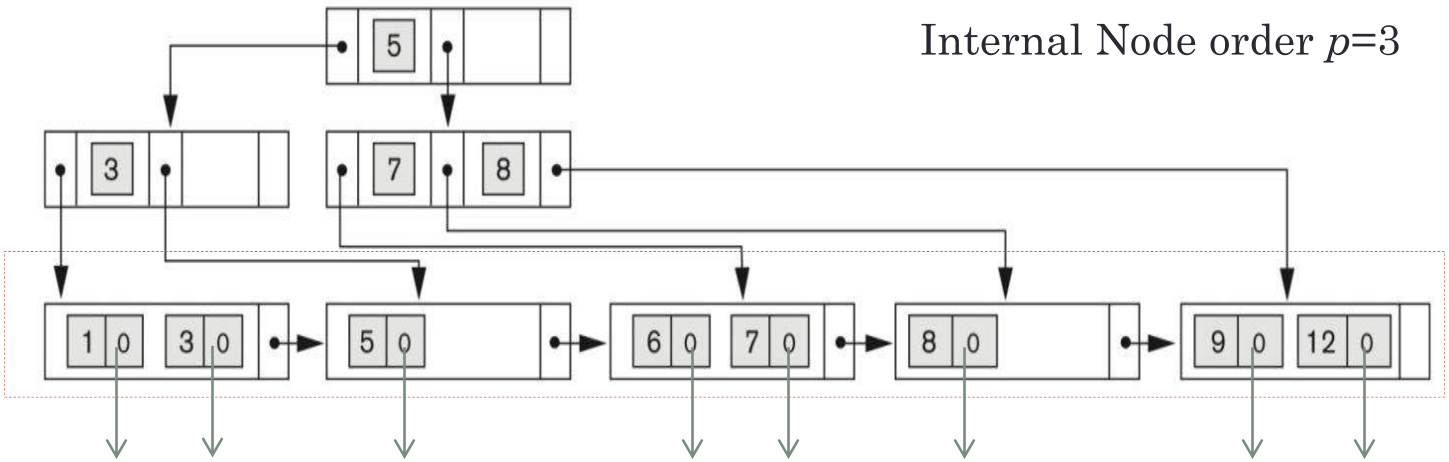
B^+ TREE: LEAF NODE

B+Tree Leaf Node order $\boldsymbol{p}_L$:= {(K_1, $\mathbf{Q_1}$), (K_2, $\mathbf{Q_2}$) …, (K_{pL}, $\mathbf{Q}_{\boldsymbol{p}L}$), $\mathbf{P_{next}}$}



- $\mathbf{Q}_i$ is a **data-pointer** to the *actual* data block holding value $\mathbf{K}_i$
- $\mathbf{P_{next}}$ is a **tree-pointer** to the *next* leaf node (*sibling*).
- **Linked-list of leaf nodes!**
- *All* leaf nodes are at the *same* level, i.e., **tree is balanced.**

B⁺ TREE: EXAMPLE



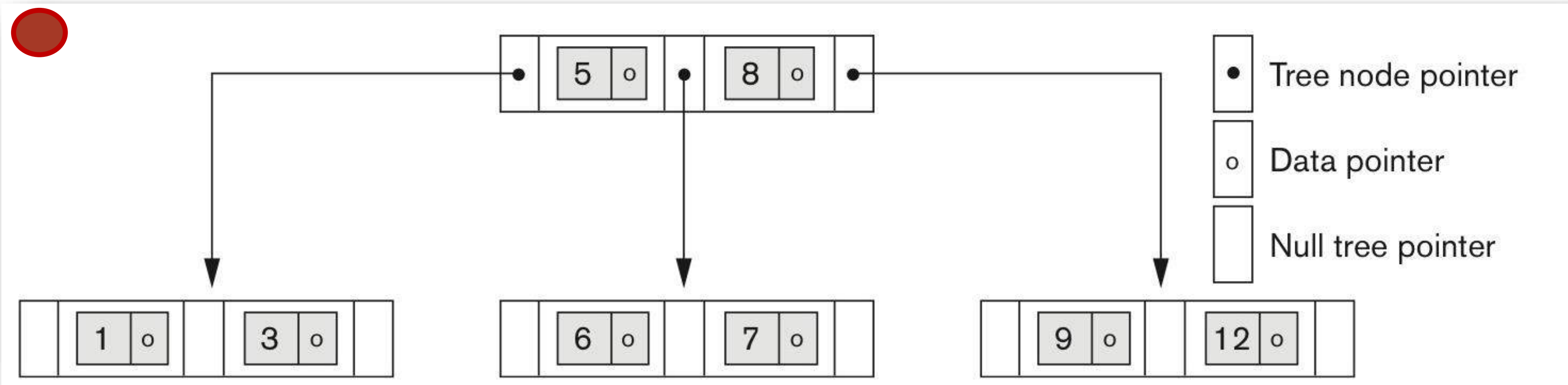
Internal Node order p=3

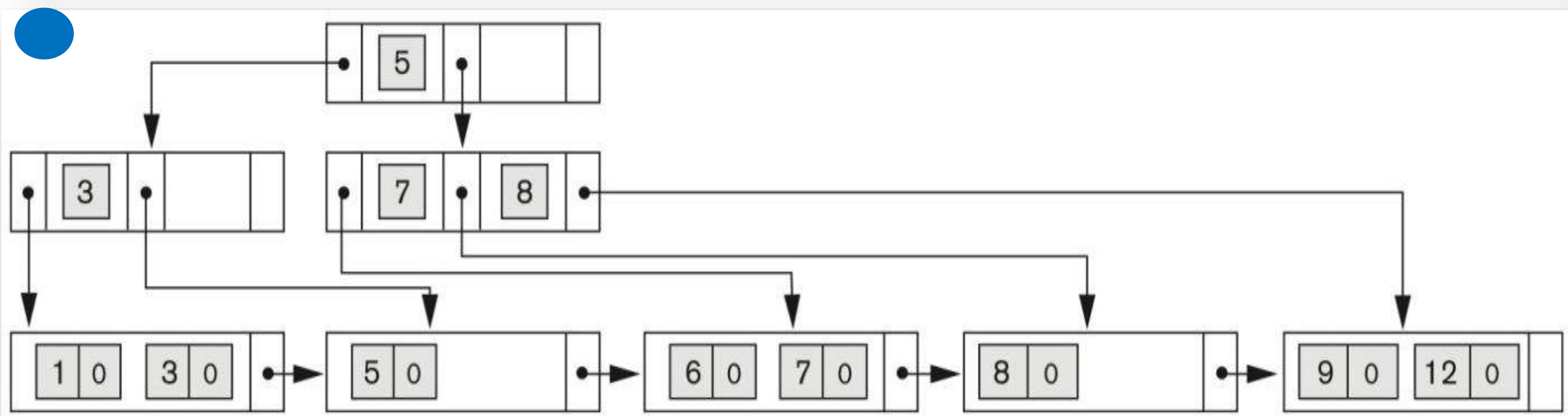
Leaf node order $p_L = 2$

1. Leaf nodes are *linked* & *sorted* by key
2. All *keys* of the file appear at the Leaf nodes!
3. Leaf nodes contain data-pointers *only* (expedite navigation)
4. Leaf nodes are balanced (constant I/O cost)
5. *Some* selected keys are replicated in the internal nodes

B TREE & B+ TREE

- No data-pointers in the Internal Nodes
- *Keys* are distributed *all* over the B Tree
- *Keys* are gathered *only* in the B+ Tree leaves

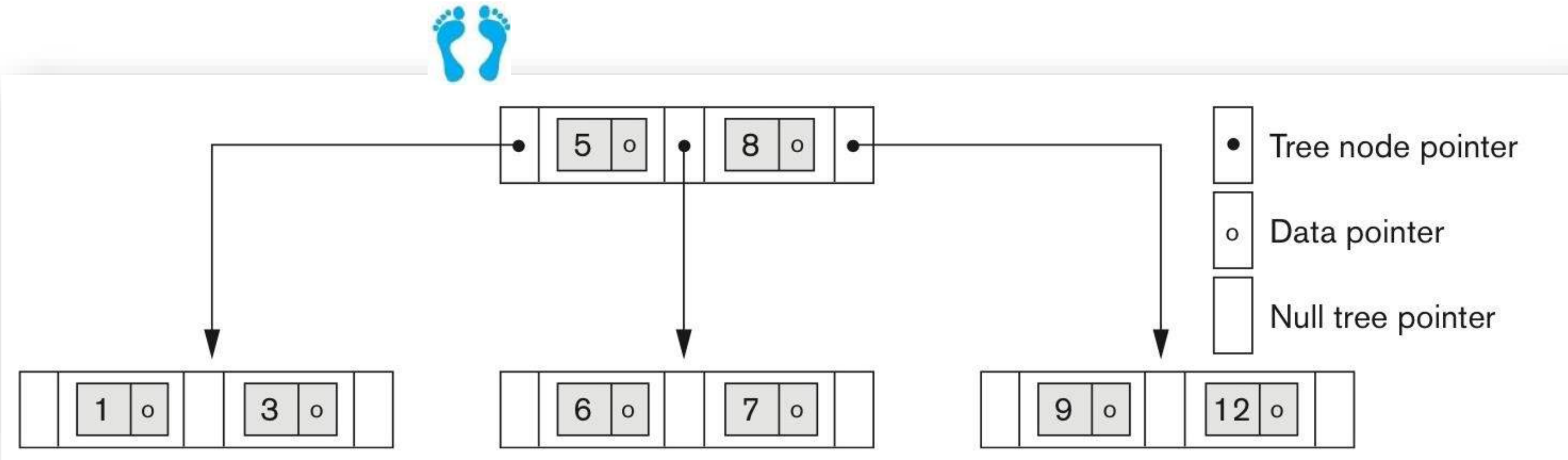


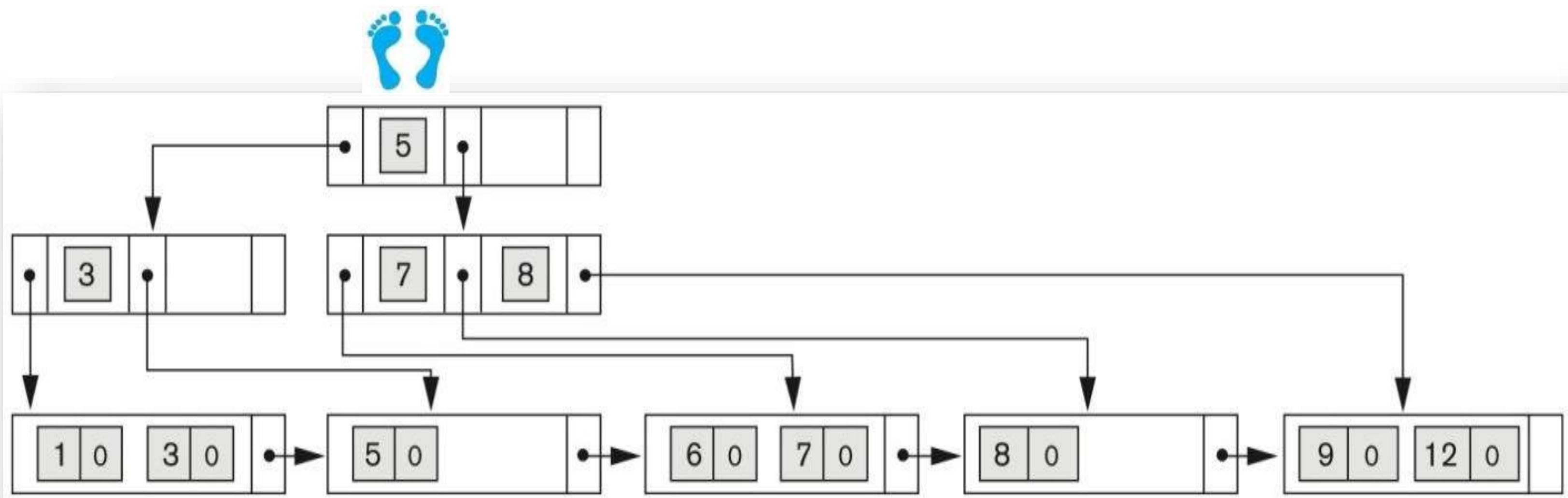


B & B$^+$ TREE

Range Query

```
SELECT * FROM EMPLOYEE
WHERE SSN >= 3 AND SSN <= 10
```





B^+ Tree & B-Tree Example [E2] (1/4)

Hypothesis: By removing the data-pointers from **internal** nodes, we obtain *higher* fan-out, thus, *more* index-entries, thus, *quicker* search process.

Context: Block **B = 512** bytes, key **V = 9** bytes, data-pointer **Q = 7** bytes, tree-pointer **P = 6** bytes.

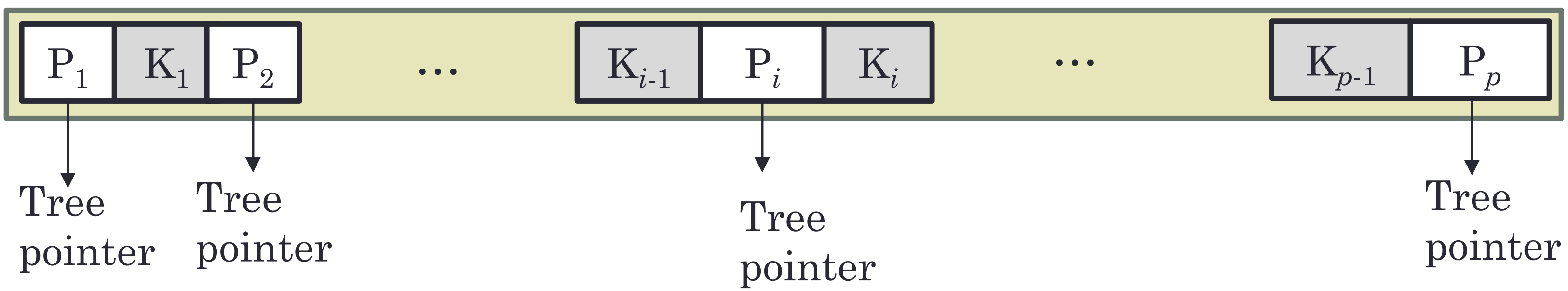
Calculate: *maximum* **order *p*** of B-Tree and B+ Tree *fitting each node in one block.*

Recall: internal B+ Tree node has *p* tree pointers and *p*-1 keys

Recall: B-Tree node has *p* tree pointers; *p*-1 keys; *p*-1 data-pointers.

B+ TREE & B TREE EXAMPLE [E2] (2/4)

[B+ Tree] Internal Node := {$\mathbf{P_1}$, K_1, …, $\mathbf{P_{p\text{-}1}}$, $K_{p\text{-}1}$, $\mathbf{P_p}$}



Step 1: Size of a B+ Internal Node: p*$\mathbf{P}$ + (p-1)*$\mathbf{V}$

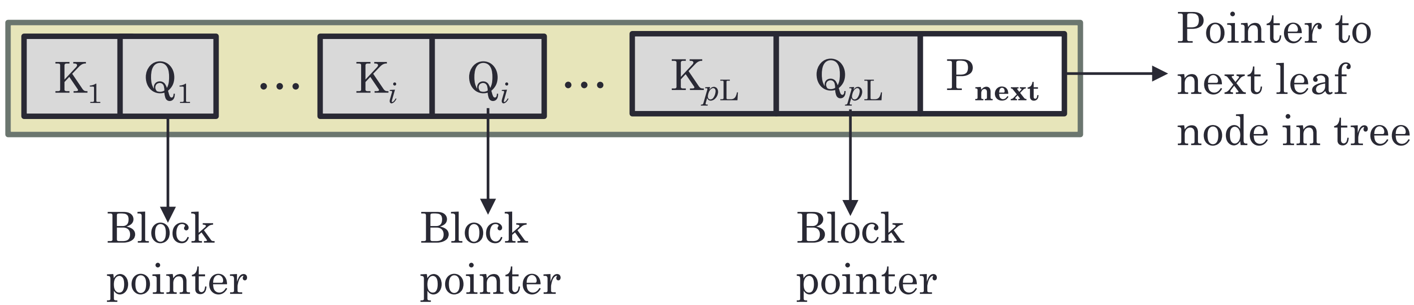
Step 2: To fit into a block we have:

$$p*\mathbf{P} + (p\text{-}1)*\mathbf{V} \leq \mathbf{B} \text{ or } \boldsymbol{p} \leq (\mathbf{B} + \mathbf{V}) / (\mathbf{P} + \mathbf{V})$$

Step 3: The maximum p order is $\boldsymbol{p}$ **= 34** (*i.e.*, 34 tree pointers; 33 key values)

B^+ Tree & B Tree Example [E2] (3/4)

[B+ Tree] Leaf Node := {(K_1, $\mathbf{Q_1}$), (K_2, $\mathbf{Q_2}$), …, (K_{pL}, $\mathbf{Q_{pL}}$), $\mathbf{P_{next}}$}.



Step 1: Size of a B+ Leaf Node:

- p_L data-pointers,
- p_L key values,
- *one* next-tree pointer.

Step 2: To fit into a block we have:

$$\boldsymbol{p}_\mathrm{L} * (\mathbf{Q} + \mathbf{V}) + \mathbf{P} \leq \mathbf{B} \text{ or } \boldsymbol{p}_\mathrm{L} \leq (\mathbf{B} - \mathbf{P}) / (\mathbf{Q} + \mathbf{V})$$

Step 3: Each leaf node can store up to $\boldsymbol{p}_\mathrm{L}$ **= 31** pairs of key/data-pointers.

B^+ Tree & B Tree Example [E2] (4/4)

[B-Tree] Node := {$\mathbf{P_1}$, (K_1, $\mathbf{Q_1}$), …, $\mathbf{P_{\mathit{p}-1}}$, ($K_{p-1}$, $Q_{\mathbf{\mathit{p}-1}}$), $\mathbf{P_{\mathit{p}}}$}

Step 1: Size of a B Tree Node:

- *p*-1 data-pointers,
- *p*-1 key values,
- *p* tree-pointers.

Step 2: To fit into a block we have:

$$p^{*}\mathbf{P} + (p-1)^{*}(\mathbf{V} + \mathbf{Q}) \leq \mathbf{B} \text{ or } \boldsymbol{p} \leq (\mathbf{B} + \mathbf{V}) / (\mathbf{P} + \mathbf{V} + \mathbf{Q})$$

Step 3: The maximum *p* order for B Tree is ***p* = 23 < 34 (for B+ Tree).**

Conclusions: B+ Tree has *higher* fan-out, *higher* search speed-up, *stores more* entries; *splits* the subspace into **34** subspaces at every level instead of **23**!

In-Class Activity [A1]

Task 1: Create a 3-level B+ Tree index, i.e., 3 internal levels and one *leaf* level with internal and leaf order $\boldsymbol{p}$ **= 34** and $\boldsymbol{p}_{\mathrm{L}}$ **= 31**, respectively.

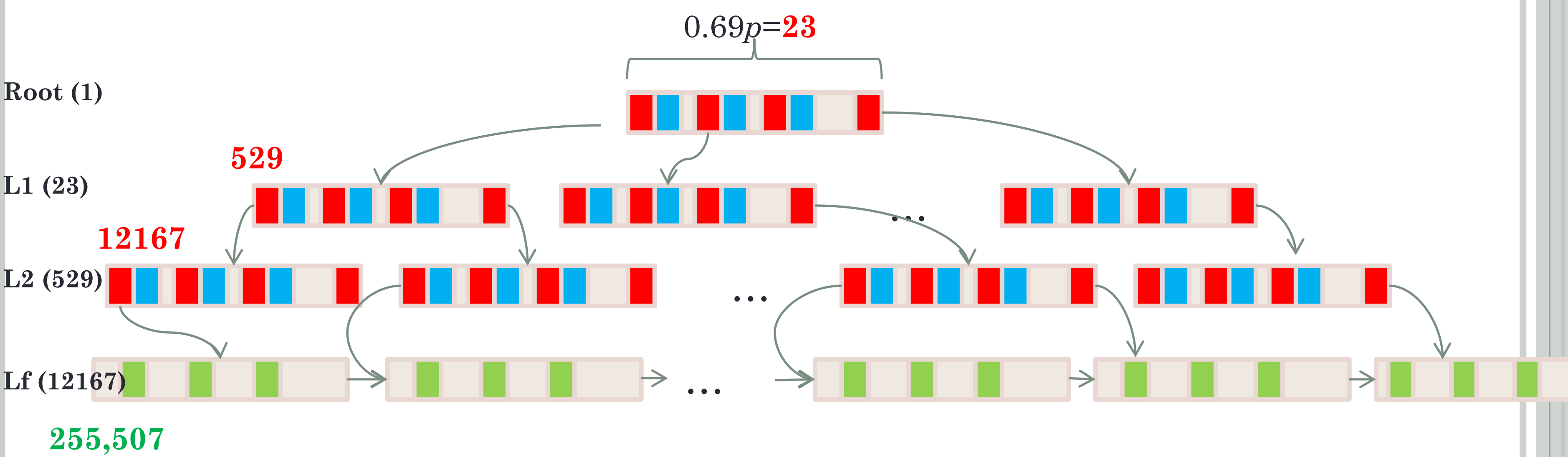
Context: Assume that each internal & leaf tree node is 69% full.

Task 2: Compare with the storage capacity of a 3-level B Tree (*see* E1 example) in terms of #keys.

In-Class Activity [A1]

3-level B+ Tree index: 3 internal levels and one *leaf* level of order $p = 34$ and $p_L = 31$

Context: Assume that each internal & leaf tree node is 69% full.



- Internal tree-node has $0.69p$ = **23 tree-pointers**, i.e., **22 keys** (on average).
- Leaf node has $0.69p_L$ = **21 data-pointers** (on average).

Root: 1 node with **22 keys** and **23** pointers to tree nodes;

Level-1: 23 nodes with 23*22 = **506 keys** and 23*23 = **529** pointers to nodes;

Level-2: 529 nodes with 529*22 = **11638 keys** and 529*23 = **12167** pointers to leaves;

Leaf Level: 12167 nodes with 12167*21 = **255,507 keys**/**data-pointers**.

In-Class Activity [A1]

Task 2: How many keys can we store?

Root: 1 node with **22 keys** and 23 pointers to tree nodes;

Level-1: 23 nodes with 23*22 = **506 keys** and 23*23 = 529 pointers to nodes;

Level-2: 529 nodes with 529*22 = **11638 keys** and 529*23 = 12167 pointers to leaves;

Leaf: 12167 nodes with 12167*21 = **255,507 keys/data-pointers**.

Leaf Level: 255,507 keys/data-pointers (*…as many keys in the leaf nodes*!)

Number of nodes: **1 + 23 + 529 + 12167 = 12720** nodes, thus, **12,720 blocks!**

Compare:

- B Tree (3-Level) stores **65,535 keys/data-pointers.**
- B+ Tree (3-Level) achieves **290%** more capacity *over* the same file!
- B+ Tree occupies 3 times *more* blocks than B Tree

- *And*, if the file needs to store more than 65,535 records, e.g., 65,536 records then we *should* define one more level for the B Tree; B+ Tree level does *not* change!

Decision Making on B+ Tree Use

B+ Tree as Secondary Index of Level t, $t > 1$, over *non-ordering key* SSN

```
SELECT  AVG(SALARY)
FROM    EMPLOYEE
WHERE   SSN >= L AND SSN <= U
```

Context:

- File $\boldsymbol{b}$ **= 1250 blocks**; $\boldsymbol{n}$ **= 1250 employees**, *bfr* = 1 employee *per* block
- SSN = 10 bytes, P = 10 bytes, Leaf order $p_L = q = 10$, Node order $p = 5$,
- *All* leaf nodes are **100% full**
- **SSN ∈ {1, 2, …, 1250}**
- Let a = ratio of employees retrieved = (U-L)/n ∈ [1/n, 1] (range ratio)

Task 1: Which is the B+ Tree level t ?

Task 2: Which is the maximum range ratio a to avoid serial scan?

Decision Making on B+ Tree Use

Task 1: Methodology

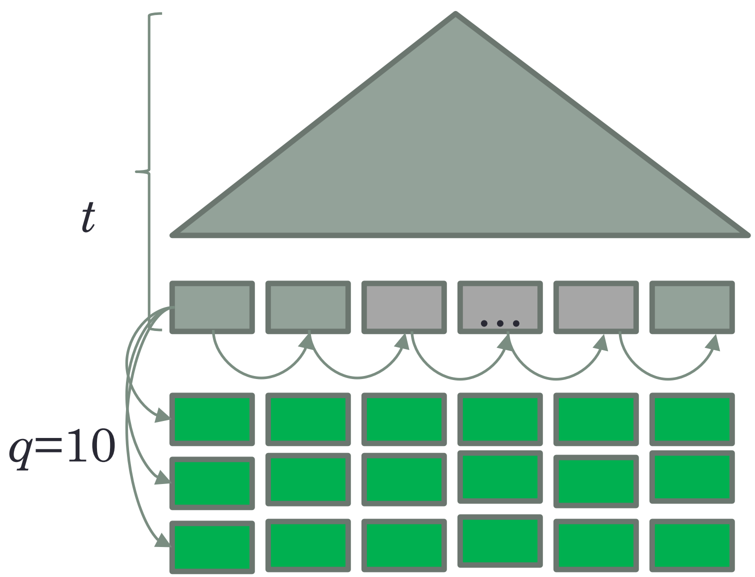
Step 1: Root node has $p = 5$ tree node pointers

Step 2: L1 has 5 nodes having 25 node pointers

Step 3: L2 has 25 nodes having 125 leaf node pointers

Step 4: Leaf Level: 125 nodes storing $q = 10$ SSN values/data pointers each, i.e., $n = 1250$ SSNs.

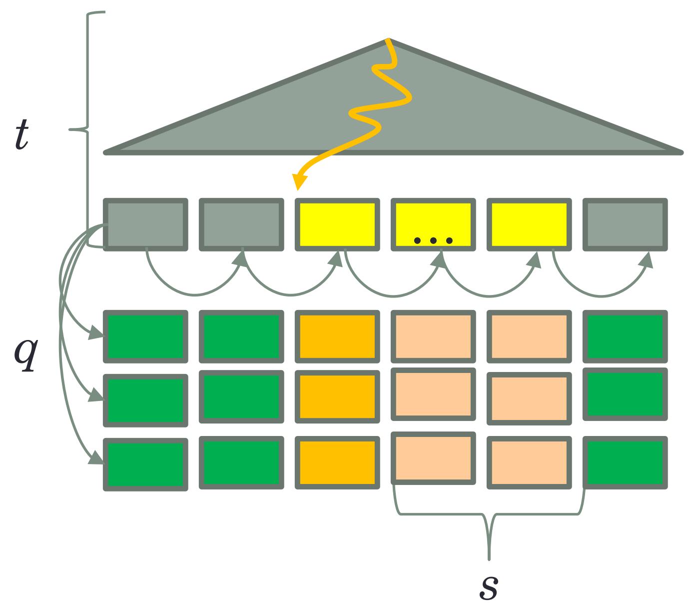
Hence, levels $t = 4$.



Task 2: Expected cost for a given range ratio $a = (\text{U-L})/n$

- $\boldsymbol{t}$ **block accesses** to reach the leaf-node with SSN = L.
- $\boldsymbol{q}$ **block accesses** for loading data-blocks and sum up Salary values.
- Leaf nodes accessed: $(\text{U-L})/q = \frac{an}{q}$ (#values-in-range / #values-in-leaf)
- Visit sibling leaf nodes: $\boldsymbol{s} = \frac{an}{q} - 1$ **block accesses**
- For *each* sibling leaf node, access $\boldsymbol{q}$ **data blocks** and sum up Salaries

Total: $\boldsymbol{C(a) = t + q + \frac{an}{q} - 1 + \left(\frac{an}{q} - 1\right) q = an\left(1 + \frac{1}{q}\right) + (t - 1)}$



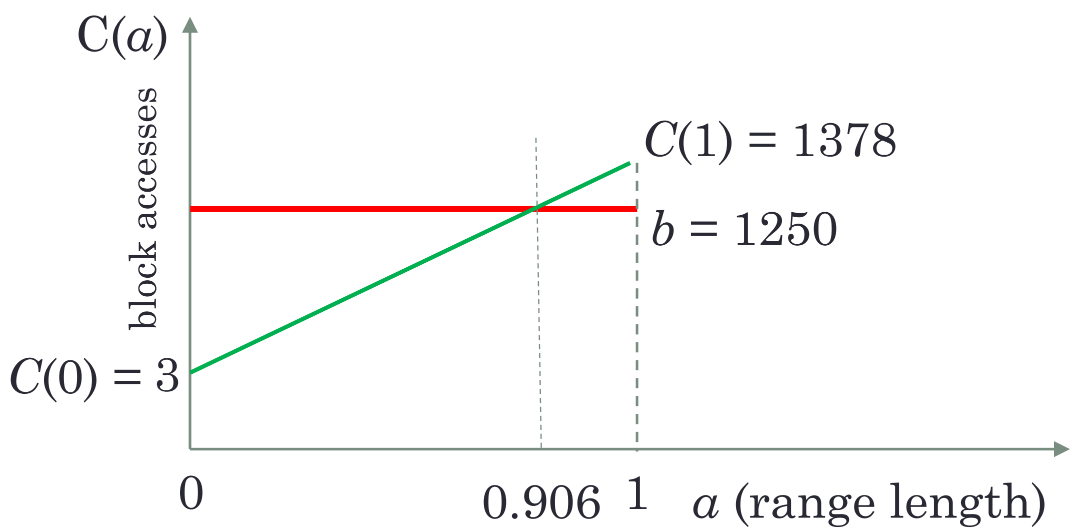
Decision Making on B+ Tree Use

Decision Rule: (B+ tree cost) **C(a) < b** (linear search cost)

C(a) = 1375a + 3 < 1250 = b or $\boldsymbol{a < 0.906}$

IF range ratio is less than 90.6%, **THEN** use B+Tree
ELSE use serial scan

Lessons Learnt: B+ Tree is not a *panacea*!



Query Processing

Database Systems (H)

Dr Chris Anagnostopoulos

ROADMAP

- **External Sorting**: Fundamental Operator

- Strategies for **`SELECT`**
  - Simple, conjunctive, disjunctive

- Strategies for **`JOIN`**
  - *Five fundamental* algorithms for `JOIN`

- **Principle:** firstly, *estimate the cost of each plan*, *choose* the plan with the *minimum* expected cost, and finally *execute*!

FUNDAMENTAL TOOL: SORTING

Almost all SQL queries involve *sorting* of tuples w.r.t. *ad-hoc* sorting requests defined by the user, e.g.,

- `CREATE PRIMARY INDEX ON EMPLOYEE(SSN)` **means** *sort* by SSN,
- `ORDER BY Name` **means** *sort* by Name,
- `SELECT DISTINCT Salary` **means** *sort* by Salary to create clusters and then identify the distinct values,
- `SELECT DNO, COUNT (*) FROM EMPLOYEE GROUP BY DNO` **means** sort by DNO to create clusters.
- …

- **Fundamental Limitation:** we cannot store the *entire* relation into memory for sorting the records ☹ (*bubble sort; quick sort; heap sort; merge sort*; …)

- **External Sorting:** sorting algorithm for large relations stored on *disk* that do not fit *entirely* in main memory.

EXTERNAL SORTING: OVERVIEW

Principle: Divide & Sort (Conquer)

- **Divide:** a file of *b* blocks into L *smaller* sub-files (*b*/L blocks each).

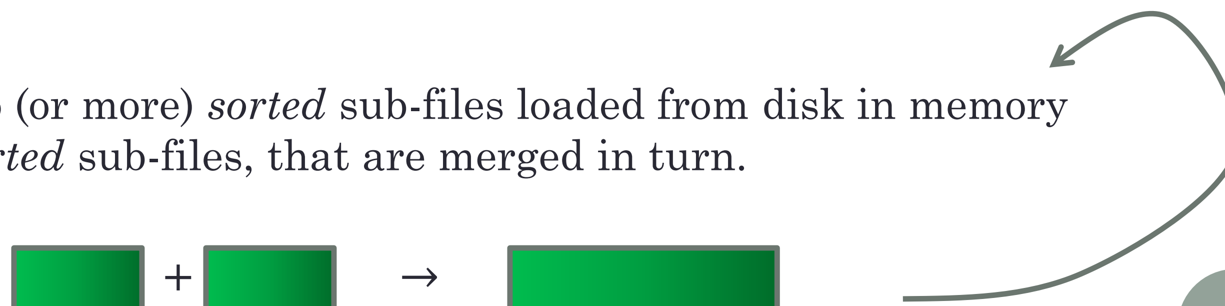
divide: L sub-files

b/L

- **Sort:** load *each small* sub-file to memory, *sort* using e.g., **quick-sort, bubble-sort** and *write* it back to the disk.

loop

- **Merge**: *merge* two (or more) *sorted* sub-files loaded from disk in memory creating *bigger sorted* sub-files, that are merged in turn.



merge sorted sub-files

bigger sorted sub-file

External Sorting: Overview

Lemma 1: The expected cost of the sort-merge strategy in *block accesses* is:

$$2b \cdot (1 + \log_M(L))$$

- *b* is the number of file *blocks*
- **M:** *degree of merging*, i.e., number of *sorted blocks merged* in each loop,
- **L:** number of the initial *sorted sub-files* (before entering merging phase).

Proof: *omitted; beyond the scope of the course* ☺

- M = 2 gives the worst-case performance;
  - Because: *merge* in parallel only a *pair* of blocks at each step;
- M > 2: merge more than *two* blocks at each step; (M-*way merging*)[*]

[*] Knuth, Donald (1998). "Chapter 5.4.1. *Multiway Merging and Replacement Selection*". Sorting and Searching. The Art of Computer Programming. 3 (2nd ed.). Addison-Wesley. pp. 158–168.

STRATEGIES FOR SELECT

`SELECT * FROM` ***relation***

`WHERE` ***selection-conditions***

- **S1. Linear Search over a *key***

Retrieve *every* record; test whether it *satisfies* the selection condition.

```
SELECT * FROM EMPLOYEE WHERE SSN = '12345678'
```

Expected Cost: $b/2$

- **S2. Binary Search over a *key***

```
SELECT * FROM EMPLOYEE WHERE SSN = '12345678'
```

Expected Cost (unsorted file): $\log_2(b) + 2b(1 + \log_M(L))$
Expected Cost (sorted file): $\log_2(b)$

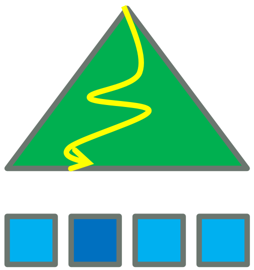
Strategies for SELECT

- **S3. Primary Index** *or* **Hash Function over a *key***

```
SELECT * FROM EMPLOYEE WHERE SSN = '12345678'
```

Precondition (Index): Primary Index of level t over *key* (sorted by key)

Precondition (Hash): File hashed with the *key*



Expected Cost (sorted file): $t + 1$

Expected Cost (hashed file): $1 + \mathbf{O}(n)$ (n = #*overflown buckets*)

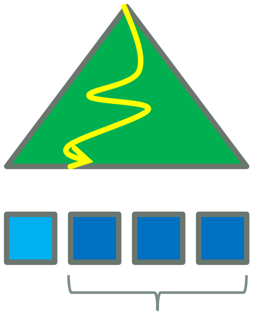
STRATEGIES FOR SELECT

- **S4. Primary Index over a *key* involved in a range query:** involving *a range*: >, ≥, <, ≤

- Use Index to find the record satisfying the *equality* (e.g., DNUMBER = 5) and then *retrieve all subsequent* blocks from the *ordered file*.

```
SELECT * FROM DEPARTMENT WHERE DNUMBER ≥ 5;
```

Precondition: Primary Index of level t over the key (file sorted by key)



Expected Cost (sorted file): t + O(b)

Note: Do *not* use Hashing for range queries!

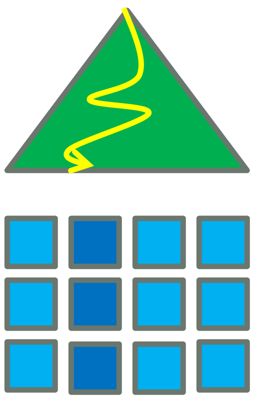
STRATEGIES FOR SELECT

- **S5. Clustering Index over *ordering*, *non-key***

- Retrieve *all contiguous* blocks of the cluster.

```
SELECT * FROM EMPLOYEE WHERE DNO = 5;
```

Precondition: Clustering Index of level t on *non-key* (file sorted by *non-key*)



Expected cost (sorted file): t + O(b/n)

Note 1: n = #distinct values of the *non-key* attribute

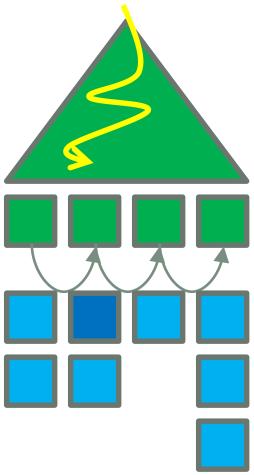
Note 2: attribute is uniformly distributed

Strategies for SELECT

- **S6. Secondary Index (B+ Tree) over *non-ordering key***

```
SELECT * FROM DEPARTMENT WHERE MGR_SSN = '1234567';
```

Precondition: File is ***not*** ordered by *key*.



Expected Cost: *t* + 1

Note: B+ Leaf Node points at the *unique* block

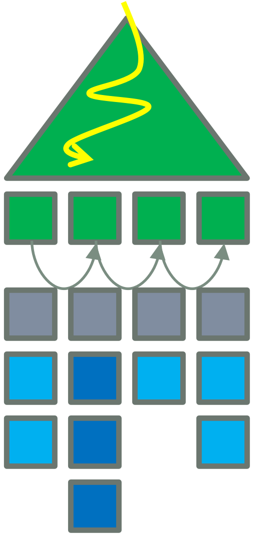
STRATEGIES FOR SELECT

- **S7. Secondary Index (B+ Tree) over *non-ordering, non-key***

Note: retrieve *multiple* records from *different* blocks having the *same* value.

```
SELECT * FROM EMPLOYEE WHERE SALARY = 40000;
```

Precondition: File is ***not*** ordered by *non-key*



Expected Cost: *t* + 1 + O(*b*)

Note: B+ Leaf Node points to a *block of pointers* to data blocks with Salary = 40K (2 levels of indirection)

Strategies for Disjunctive select

Disjunctive Selections: conditions involving **OR**

```
SELECT * FROM EMPLOYEE
WHERE    SALARY > 10000 OR NAME LIKE '%Chris%'
```

Final result: contains tuples satisfying the *union* of *all* selection conditions

Methodology:

- **IF** an *access path* exists, e.g., B+/hash/primary-index for **all** of the attributes:
  - use *each* to retrieve the *set* of records satisfying *each* condition
  - *union* all sets to get the final result.
- **ELSE** if *none* or *some* of the attributes have an access path, *linear search* is unavoidable!

Strategies for Conjunctive SELECT

Conjunctive Selections: conditions involving **AND**

```
SELECT * FROM EMPLOYEE
WHERE   SALARY > 40000 AND NAME LIKE '%Chris%'
```

Methodology:

- **IF** an *access path exists* (index) for *an* attribute, use it to retrieve the tuples satisfying the condition, e.g., `Salary > 40000` **[intermediate result]**
- **GO** *through* this *intermediate result* to check which record satisfies *also* the other condition(s), e.g., `Name LIKE '%Chris%'` **in memory**!

If you have two indexes, *which* index is to be used first?

- **Answer:** *use* the index that generates the **smallest** intermediate result set *hoping* to fit in the memory! [*selectivity* = *#tuples retrieved*]
- **Optimization**: find the execution sequence of conditions that *minimizes* the expected cost.

- **Principle: Predict the *selectivity* beforehand!**

STRATEGIES FOR JOIN

Observation: the *most resource-consuming operator*!
Focus: *two-way equijoin*, i.e., join *two* relations with equality '='

```
SELECT *
FROM   EMPLOYEE E, DEPARTMENT D
WHERE  E.DNO = D.DNUMBER
```

Five fundamental strategies for join processing:

- Naïve join (*no access path*)
- Nested-loop join (*no access path*)
- Index-based nested-loop join (*index*; B+ Trees)
- Merge-join (*sorted relations*)
- Hash-join (*hashed relations*)

NAÏVE JOIN

```
SELECT *
FROM   R, S
WHERE  R.A = S.B --A and B are join attributes, e.g., PK, FK.
```

- **Step 1:** Compute the Cartesian product of **R** and **S**, i.e., *all* tuples from **R** are concatenated (*combined*) with *all* tuples from **S**.
- **Step 2:** Store the result in a file **T** and for each concatenated tuple $t = (r, s)$ with $r \in$ **R** and $s \in$ **S** check *if* r.A = s.B

Algorithm Naïve Join

T = Cartesian R x S

Scan T, a tuple $t \in$ T at a time: $t = (r, s)$

  If r.A = s.B **then** add (r, s) to the result file

  Go to next tuple $t \in$ T.

Outcome: *inefficient*, typically the result is a small *subset* of the Cartesian!

- **What-If:** *no* tuples are actually matched; *predict* the matching tuples in advance!

NESTED-LOOP JOIN

Algorithm Nested-Loop Join

For each tuple $r \in$ **R** **//outer relation**

  For each tuple $s \in$ **S** **//inner relation**

    If r.A = s.B **then** add (r, s) to the result file;

Note: the outer & inner loops are *over* blocks and *not* over tuples!

Note: Re-form the pseudocode in a *block-centric* programming mode (system programming *using* files ☺)

Challenge 1: Which relation should be in the *outer* loop and which in the *inner* loop to *minimize* the join processing cost?
Optimization problem

NESTED-LOOP JOIN: ALGORITHM

Step 1:

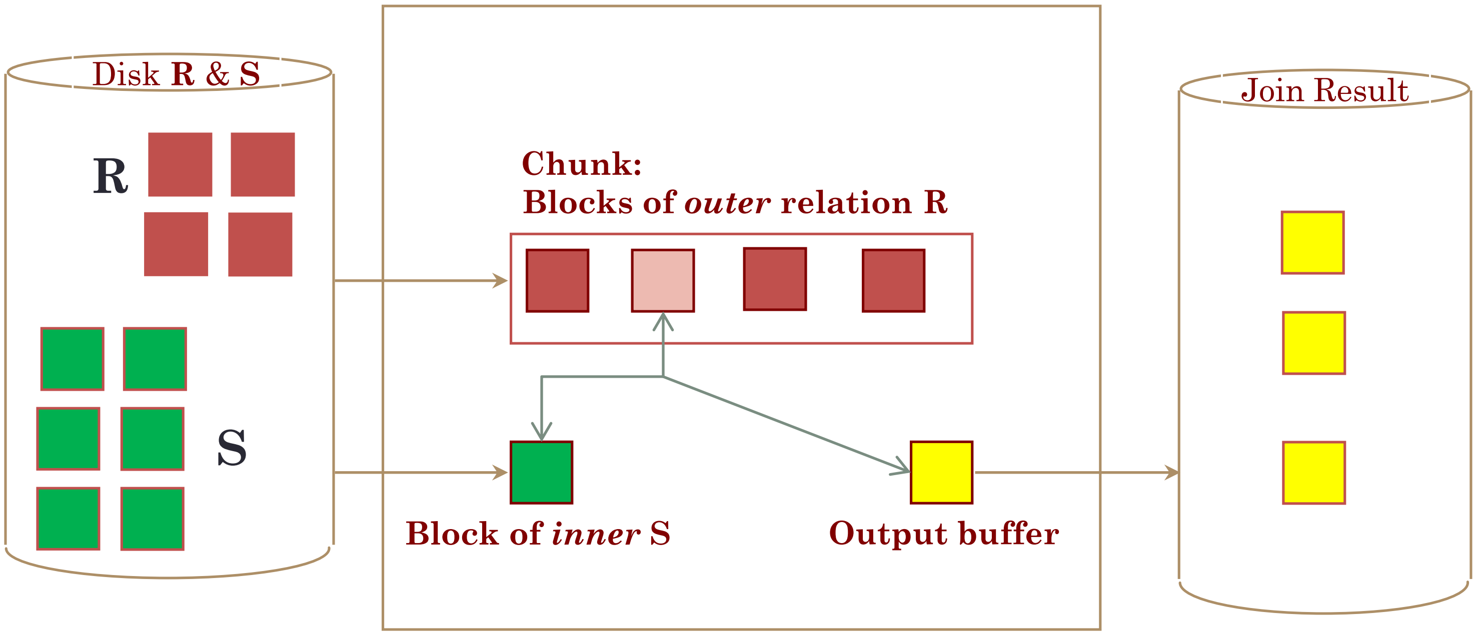
- LOAD a set (*chunk*) of blocks from the *outer* relation **R.**
- LOAD one block from *inner* relation **S**
- Maintain an *output* buffer for the matching (*resulting*) tuples (r, s): r.A = s.B

Step 2:

- JOIN the **S** block with *each* **R** block from the chunk
- FOR each *matching* tuple $r \in$ **R**-block and $s \in$ **S**-block ADD (r, s) to Output Buffer
- IF Outer Buffer is *full*, PAUSE; WRITE the current join result to disk; CONTINUE

Step 3: LOAD *next* **S**-block and GOTO **Step 2**

Step 4: GOTO **Step 1**



Index-based Nested-Loop Join

Idea: Use of an *index* on either A or B joining attributes: **R**.A = **S**.B.

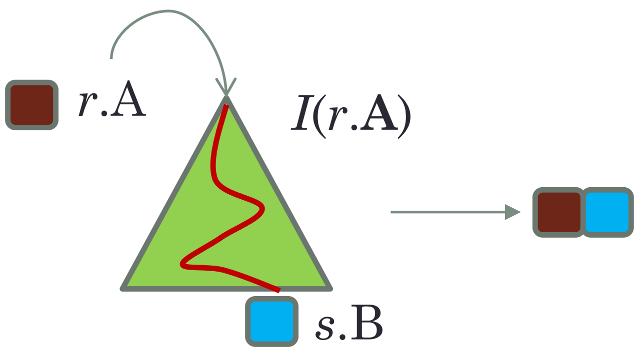
Focus: Assume an *index I* on attribute B of relation **S**.

Algorithm Index-Based Nested-Loop Join

For each tuple $r \in$ **R**

Use *index* of B**:** $I(r.\mathrm{A})$, to *retrieve all* tuples $s \in$ **S** having $s.\mathrm{B} = r.\mathrm{A}$

For each *such* tuple $s \in$ **S**, add matching tuple (r, s) to the result file;



Claim: Much faster compared to the nested-loop join, **why?**

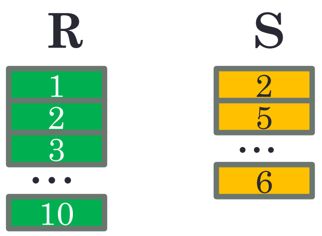
Because: We get *immediate* access on $s \in$ **S** with $s.\mathrm{B} = r.\mathrm{A}$ by *searching* for $r.\mathrm{A}$ using the index I, avoiding linear search on **S.**

Challenge 2: Which index to use to *minimize* the join processing cost?
Optimization problem

SORT-MERGE JOIN

Idea: Use of the *merge-sort algorithm over* two *sorted* relations w.r.t. their joining attributes.

Pre-condition: Relations **R** and **S** are *physically ordered* on their joining A and B;



Methodology:

- **Step 1:** Load a pair {**R.block**, **S.block**} of *sorted* blocks into the memory;
- **Step 2:** Both blocks are *linearly* scanned *concurrently* over the joining attributes (*sort-merge* algorithm in memory);
- **Step 3:** If matching tuples *found* then store them in a buffer.

Gain: The blocks of each file are scanned *only* once!

But: If **R** and **S** are *not* a-priori *physically* ordered on A and B then *sort* them first!

Sort-Merge Join: Example

R

	A	sname	rating	age
$i \rightarrow$	22	dustin	7	45.0
$i \rightarrow$	28	yuppy	9	35.0
$i \rightarrow$	44	guppy	5	35.0
$i \rightarrow$	58	rusty	10	35.0

S

	B	bid	day	rname
$j \rightarrow$	28	103	12/4/96	guppy
$j \rightarrow$	28	103	11/3/96	yuppy
$j \rightarrow$	31	101	10/10/96	dustin
$j \rightarrow$	31	102	10/12/96	lubber
$j \rightarrow$	31	101	10/11/96	lubber
$j \rightarrow$	58	103	11/12/96	dustin

Result Buffer

- (28,yuppy,9,35,103,12/4/96,guppy)
- (28,yuppy,9,35,103,11/3/96,yuppy)
- (58,rusty,10,35,103,11/12/96,dustin)

HASH-JOIN

Pre-condition:

- File **R** is partitioned into M *buckets* w.r.t. **hash function** *h* over joining attribute A.
- File **S** is *also* partitioned into M *buckets* w.r.t. the ***same*** *hash* function *h* over attribute B;

Assumption*:* **R** is the *smallest file* and fits into main memory: M buckets of **R** are in memory.

Algorithm Hash-Join

/*Partitioning phase */

For each tuple $r \in$ **R**,

  Compute $y = h(r.A)$ /* address of bucket*/

  Place tuple *r* into *bucket* $y = h(r.A)$ in memory

/*Probing phase*/

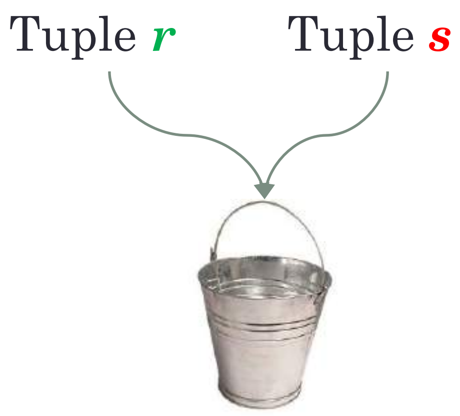
For each tuple $s \in$ **S**,

  Compute $y = h(s.B)$ /*use the same hash function *h**/

  Find the *bucket* $y = h(s.B)$ in memory (of the **R** partition).

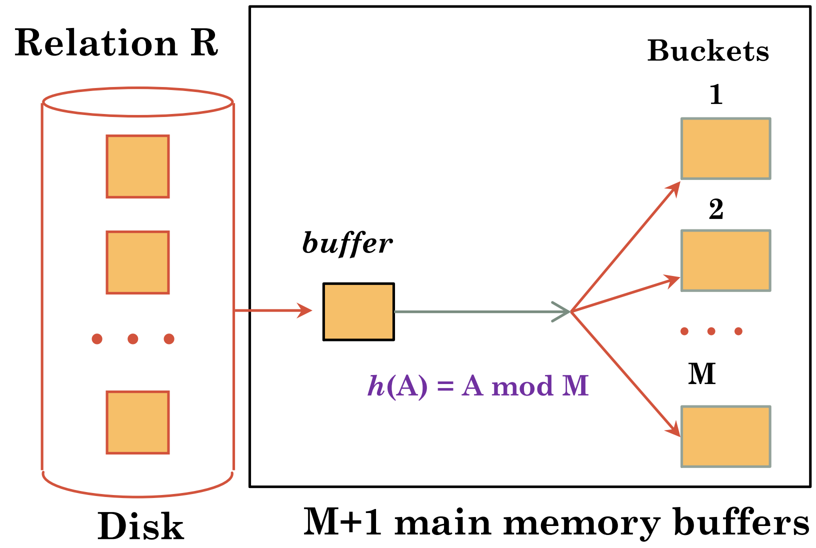
  For *each* tuple $r \in$ **R** in the bucket $y = h(s.B)$

    If $s.B = r.A$ **add** (r, s) to the result file; /*join*/



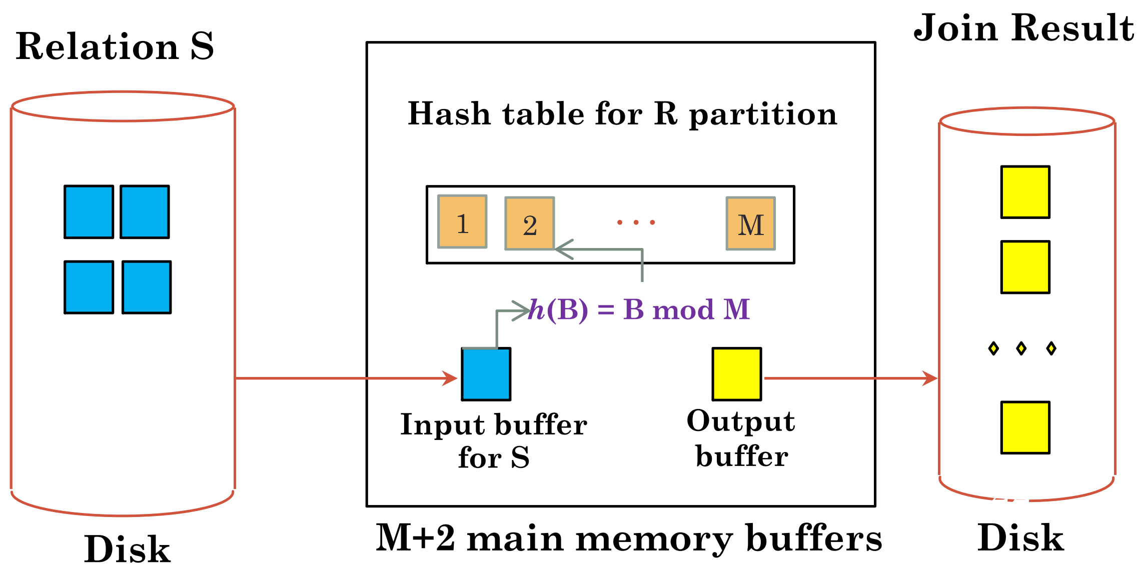
Partitioning Phase

Partition of **R** over attribute A using *hash* ***h*(A) = A mod M** into M buckets.



Probing Phase

Hashing each tuple *s* from **S**, using *hash* ***h*(*s*.B) = *s*.B mod M** to identify the *y* = *h*(*s*.B) bucket in memory.



So Far…

- **Naïve Join:** Exploit *nothing*. Cartesian product and then check…
- **Nested-Loop:** Exploit *nothing*. Computing-oriented join.
  - *Which* relation should be in the *outer* loop? Influences the join cost.
  - Can you *predict* the cost then? **Optimization…**

- **Index-based Nested-Loop:** Exploit at *least* one *index*. Use *index* to find the matched tuples as quick as possible ☺
  - If we have two indexes (over R.A and over S.B), which one to use? Influences the join cost! **Optimization…**

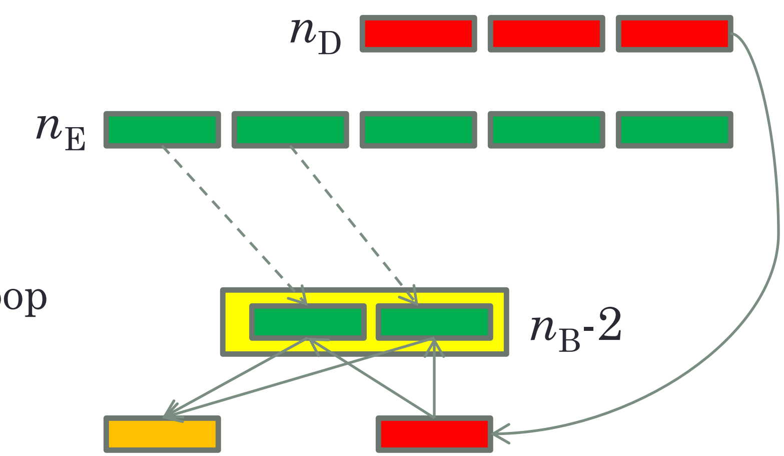
- **Merge-Join**: Exploit *both ordered* relations; otherwise; sort them ☹

- **Hash-Join:** Exploit *hashing*. Hash *one* relation.
- Use the same hashing function to find the matching tuples in the same bucket ☺

- **Challenge 3: *predict* join cost, *choose* best strategy, *execute*!**

Nested-Loop Join Cost Prediction

```
SELECT   *
FROM     EMPLOYEE E, DEPARTMENT D
WHERE    E.DNO = D.DNUMBER
```



Employee (E): n_E blocks used at the *outer* loop

Department (D): n_D blocks used at the *inner* loop

Memory: n_B blocks available:

- 1 block for *reading* the **inner** file D,
- 1 block for *writing* the join **result**,
- n_B-2 blocks for *reading* the **outer** file E: **chunk size**.

Observation 1: Each block of the *outer* relation E is read *once*.

Observation 2: The *whole inner* relation D is read *every time* we read a chuck of (n_B-2) blocks of E.

NESTED-LOOP JOIN COST PREDICTION

- Total number of blocks read for *outer* relation E: $\mathbf{n_E}$
- **Outer Loops:** Number of *chunks* of (n_B-2) blocks of *outer* relation: $\mathbf{ceil(n_E/(n_B-2))}$
- For *each* chunk of (n_B-2) blocks read *all* the blocks of *inner* relation D:
- Total number of block read in all outer loops: $\mathbf{n_D * ceil(n_E/(n_B-2))}$

Total Expected Cost: $\mathbf{n_E + n_D * ceil(n_E/(n_B-2))}$ **block accesses**

Example: $\mathbf{n_E}$ = 2,000 blocks; $\mathbf{n_D}$ = 10 blocks; $\mathbf{n_B}$ = 7 blocks

Strategy Cost 1: (E *outer*; D *inner*) $\mathbf{n_E + n_D * ceil(n_E/(n_B-2))}$ **= 6,000 block accesses**
Strategy Cost 2: (D *outer*; E *inner*) $\mathbf{n_D + n_E * ceil(n_D/(n_B-2))}$ **= 4,010 block accesses**

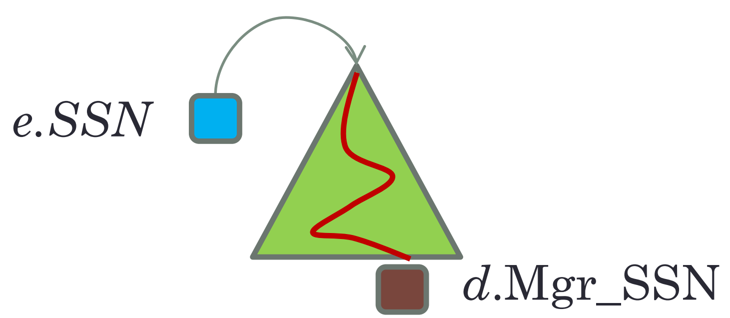
Lesson Learnt: The file with *fewer* blocks goes to the outer loop.

INDEX-BASED NESTED-LOOP COST PREDICTION

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D
WHERE    D.MGR_SSN = E.SSN
```

- B+ Tree on **Mgr_Ssn** with level $x_D = 2$
- B+ Tree on **SSN** with level $x_E = 4$
- E: $r_E = 6000$ tuples; $n_E = 2{,}000$ blocks; D: $r_D = 50$ tuples; $n_D = 10$ blocks

Strategy 1: Employee *e*, *use* B+ Tree (Mgr_Ssn) to find department *d*: *e*.Ssn = *d*.Mgr_Ssn.



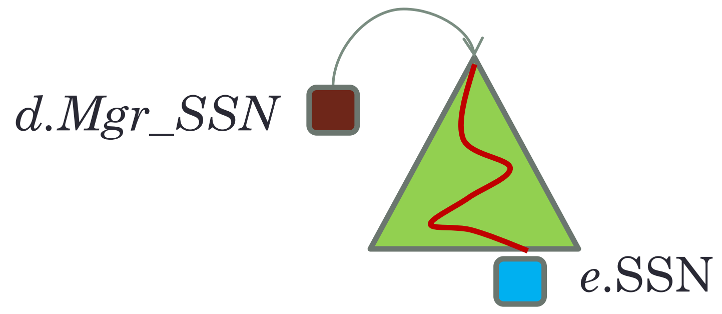
Observation: *not all employees are managers; –search without meaning sometimes…*

Probability an employee *being* manager: 50/6000 = 0.83% (99.16% meaningless searches)

Strategy Cost 1: $n_E + r_E*(x_D + 1) = 20{,}000$ **block accesses;**

Index-based Nested-Loop Cost Prediction

Strategy 2: Department *d*, *use* B+ Tree (SSN) to find Employee *e*: *e*.Ssn = *d*.Mgr_Ssn



Observation: *every department* has *one* manager –*search is fruitful…*

Probability a manager *being* an employee is 100% ☺

Strategy Cost 2: $\boldsymbol{n_D + r_D*(x_E + 1)}$ **= 260 block accesses;**

- **Huge difference (20,000 vs 260 block accesses):**
- *every* record in Department is joined with *exactly* one record in Employee (*manager*)
- *only some employees* from Employee are managers of departments…

Lesson Learnt: Use the index built on the PK

Note: *not* for recursive relationships, e.g., employee-supervisor

Sort-Merge-Join Cost Prediction

Requirement: Efficient if *both* Employee E and Department D are *already* sorted by their joining attributes: SSN and Mgr_Ssn.

Observation: only a *single* pass is made for *each* file.

Strategy Cost: $\boldsymbol{n_E + n_D}$ **= 2,010 block accesses**.

IF *both* files are *not* sorted THEN use external sorting!

Strategy Cost 1: External sorting (2-way merge): $\mathbf{2 \cdot n_E + 2 \cdot n_E \cdot \log_2(ceil(n_E / n_B))}$

- $ceil(n_E / n_B)$: number of *initial* sorted sub-files (each sub-file is n_B blocks)
- n_B: number of available memory blocks.

Strategy Cost 2: External sorting (2-way merge): $\mathbf{2 \cdot n_D + 2 \cdot n_D \cdot \log_2(ceil(n_D / n_B))}$

Sort-Merge-Join Cost Prediction

Total Strategy Cost:

$$\boldsymbol{n_E + n_D + 2 \cdot n_E + 2 \cdot n_E \cdot \log_2(\mathrm{ceil}(n_E / n_B)) + 2 \cdot n_D + 2 \cdot n_D \cdot \log_2(\mathrm{ceil}(n_D / n_B))}$$

Example: $\boldsymbol{n_E}$ = 2,000 blocks; $\boldsymbol{n_D}$ = 10 blocks; $\boldsymbol{n_B}$ = 7 blocks, we get:

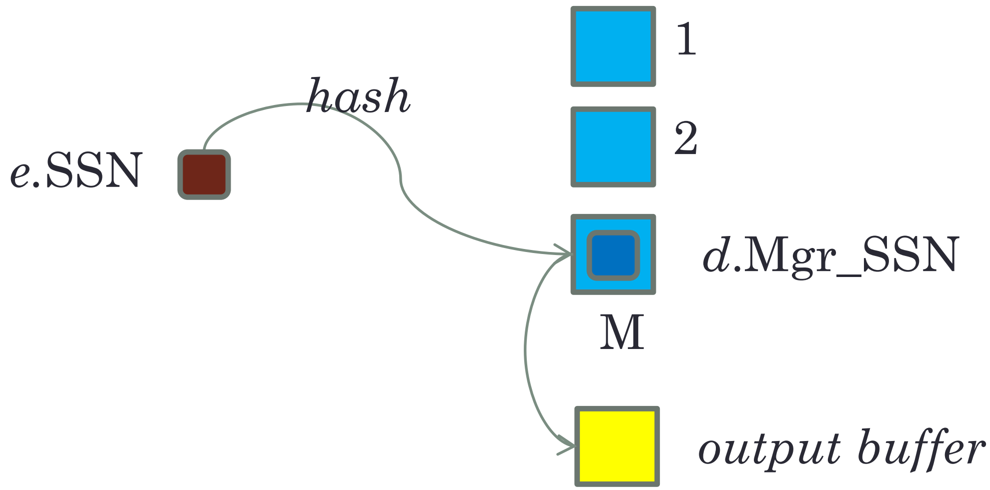
2010 + 4000 + 4000 log(286) + 20 + 20 log(2) = **38,690 block accesses;** only **5.1%** is devoted to join!

Lesson Learnt: Think before sort *only* for joining purposes!

HASH-JOIN COST PREDICTION

Best Case: Memory $\boldsymbol{n_B > n_D + 2}$

n_D: blocks for the *smallest* of the two relations (e.g., Department)



- *Whole* relation Department *fits* in memory and is *hashed* into M buckets.
- *Each* Employee tuple is *loaded* and *hashed* on joining attribute SSN.
- The corresponding *bucket* is found and searched for a matching tuple.
- The *result* is stored in another buffer (that's why $n_B > n_D + 2$)

Best Case: $\boldsymbol{n_E + n_D}$ block accesses.

Normal Case (*smallest* relation **cannot** fit in memory)**:** $\boldsymbol{3(n_E + n_D)}$ block accesses

PUT-ALL-TOGETHER: JOIN COST PREDICTION

- Naïve Join Cost: $n_E * n_D$: **20,000 block accesses**
- Nested-Loop Cost (*best*): $\mathbf{n_D + n_E * ceil(n_D/(n_B\text{-}2))}$: **4,010 block accesses**
- Index-based Nested-Loop Cost (*best*): $n_D + r_D*(x_E + 1)$: **260 block accesses**
- Sort-Merge Cost (*already* sorted): $n_E + n_D$: **2,010 block accesses**
- Hash-Join Normal-Case Cost: $3(n_E + n_D)$: **6,030 block accesses**

Hold on a second: the cost for *writing* the result-set buffer (*block*) from *memory* to *disk* in each strategy is not yet considered!

How many blocks are written? How many tuples are matched?…*next weeks*

IN-CLASS EXAMPLE [A1]

```
SELECT D.NAME, E.NAME
FROM   EMPLOYEE E, DEPARTMENT D
WHERE  E.SSN = D.MGR_SSN
```

- B+ Tree on SSN with level $\boldsymbol{x}_{\mathbf{E}} = \mathbf{2}$
- Employee: $\mathbf{n_E = 100}$ blocks; Department: $\boldsymbol{r}_{\mathbf{D}} = \mathbf{100}$ tuples; $\mathbf{n_D = 10}$ blocks
- Memory: $\mathbf{n_B = 12}$ blocks

Task: propose 2 strategies and choose the best.


IN-CLASS EXAMPLE [A1]

- B+ Tree on SSN with level $\boldsymbol{x_E} = \mathbf{2}$
- Employee: $\mathbf{n_E = 100}$ blocks; Department: $\boldsymbol{r_D} = \mathbf{100}$ tuples; $\mathbf{n_D = 10}$ blocks
- Memory: $\mathbf{n_B = 12}$ blocks

Strategy 1: Use B+ Tree index (SSN)

- For each department *d*, find details of its manager: *d*.MGR_SSN
- **Cost-1: $n_D + r_D*(x_E + 1)$ = 310 block accesses**

Strategy 2: Department relation fits in memory; hash-join (M = 10 buckets).

- Hash Department and store into memory: $\mathbf{n_D = 10}$ block accesses.
- Scan Employee, one block at a time, map an employee in to a bucket, search within the bucket: $\mathbf{n_E = 100}$ block accesses.
- **Cost-2: $n_D + \mathrm{n_E}$ = 110 block accesses** 

IN-CLASS EXAMPLE [A2]

```
SELECT E.NAME, S.NAME
FROM   EMPLOYEE E, EMPLOYEE S
WHERE  E.SUPER_SSN = S.SSN
```

Context:

- b = 2,000 blocks;
- r = 10,000 records (employees);
- B+ Index over SSN of level $x = 5$
- B+ Index over Super_SSN of level $y = 2$

- 10% of employees are supervisors;
- a supervisor does not have any supervisor: **Super_SSN is NULL**;

Task: Propose a plan that minimizes the expected cost using Index-based Nested-loop.

IN-CLASS EXAMPLE [A2]

```
SELECT E.NAME, S.NAME
FROM   EMPLOYEE E, EMPLOYEE S
WHERE  E.SUPER_SSN = S.SSN
```

Facts (Solution 1):

- PK is SSN & FK is Super_SSN.
- Use the B+ Index over SSN (**ssn-index**).

- Scan relation Employee once, i.e., b = 2000 block accesses
- For *each* block from Employee:
  - For *each* tuple *e* check if this employee is a supervisor, i.e., Super_SSN is NULL
  - **IF** employee *e* is NOT supervisor (w.p. 90%)
  - **THEN** use **ssn-index**(e.*super_ssn*)
  - **ELSE** go to next employee

- **Total Cost:** $b + 0.9*r*(x+1)$ = **56,000 block accesses**

IN-CLASS EXAMPLE [A2]

```
SELECT E.NAME, S.NAME
FROM   EMPLOYEE E, EMPLOYEE S
WHERE  E.SUPER_SSN = S.SSN
```

Facts (Solution 2):

- PK is SSN & FK is Super_SSN.
- Use the B+ Index over Super_SSN (**super-index**).

- Scan relation Employee once, i.e., $b = 2000$ block accesses
- For *each* block:
  - For *each* tuple *e* check if this employee is a supervisor, i.e., Super_SSN is NULL
  - **IF** employee *e* is NOT supervisor (w.p. 90%)
  - **THEN** use **super-index**(e.*super_ssn*)
  - **ELSE** go to next employee

- **Total Cost:** $b + 0.9 * r * (y+1)$ = **29,000 block accesses (48% faster)**

Hash-Join Cost Prediction (OPTIONAL)

Normal Case: The *smallest* relation **cannot** fit in memory.

Partitioning Phase

- **Read** both relations E & D first (one *block* at a time);

  Partial Cost: $n_E + n_D$
- **Partition** into M *buckets* using with the *same* hashing function *h*,
  - The M *main* buckets *fit* in memory; *overflown* blocks in disk!
- **Store** the hashed buckets of each relation to the disk.

  Partial Cost: $n_E + n_D$

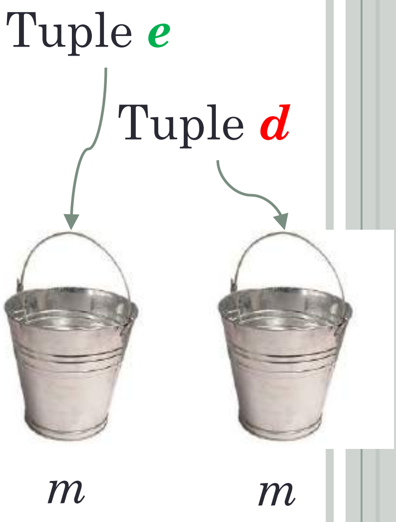
Probing Phase

For each *m* =1…M bucket **Do**

- **Read a pair**: the *m*-th bucket from E and the *m*-th bucket from D

  Partial Cost: $n_E + n_D$
- **Perform** *join* focusing *only* on the tuples from the *same* bucket *m*

  Idea: ***e*** **might be matched with** ***d*** **since** $h(e) = h(d) = m$



Expected Cost: $3(n_E + n_D)$ **block accesses.**

Query Optimization: Part I

Database Systems (H)
Dr Chris Anagnostopoulos

ROADMAP

- *Fundamental* components in Optimization:
  - **Selection Selectivity**: *fraction* of tuples satisfying a condition.
  - **Join Selectivity:** *fraction* of matching tuples in a Cartesian space.
- **Challenge 1**: Predict the selection cardinality, i.e., *predict* the *number* of tuples satisfying a selection condition of a selection query.
- **Challenge 2:** *Predict* the number of blocks we need to retrieve given a selection query.
- **Challenge 3**: *Refine* the expected cost of selection strategies involving the selectivity metric;
  - *expected cost is expressed as a function of selectivity.*

Query Optimization

Input: Query

Output: Optimal *execution* plan

- **Heuristic Optimization:**
  - **Task:** Transform a SQL query into an **equivalent** and **efficient** query using Relational Algebra.

- **Cost-based Optimization:**
  - **Task 1:** Provide *alternative* execution plans and *estimate* (**predict**) their costs
  - **Task 2:** Choose the plan with the **minimum cost;**
  - **Cost Function $c(x_1, x_2, x_3, x_4, \ldots)$ with optimization parameters:**
    - x_1= # block accesses,
    - x_2= memory requirements,
    - x_3= CPU computational cost,
    - x_4= network bandwidth,
    - …

Cost-based Optimization

Exploit: statistical information to *estimate* the execution cost of a query.

- **Information for *each* Relation:**
  - *number* of records (*r*); (*average*) *size* of each record (**R**)
  - *number of blocks* (*b*); *blocking factor* (*f*) i.e., records per block
  - *Primary File Organization*: heap, hash, or sequential file
  - *Indexes*: primary, clustering index, secondary index, B+ Trees.

- **Information for *each* Attribute A of *each* Relation:**
  - Number *of* Distinct Values (NDV) *n* of attribute A
  - Domain range: [min(A), max(A)]
  - Type: *continuous* or *discrete* attribute; **key** or **non-key**
  - Level *t* of Index of the attribute A*, if exists*

probability

Cost-based Optimization

Information for *each* Attribute A:

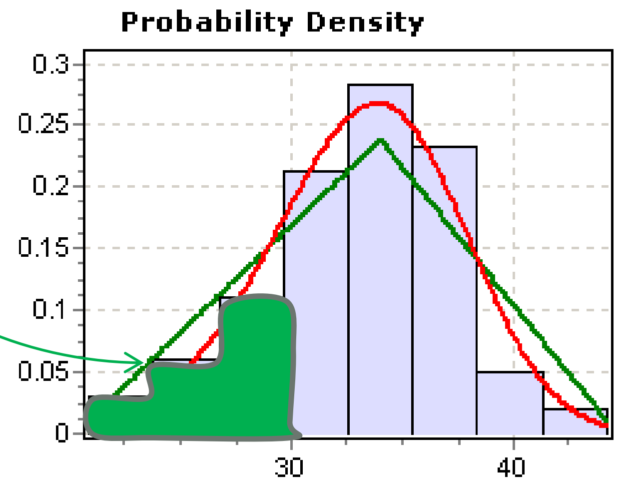
- *Probability Distribution Function* **P(A = *x*)** indicates the *frequency* (*probability*) of each value ***x*** of the attribute **A** in the relation.
- A *good* approximation of a distribution: **histogram**

P(Salary ≤ 30K) = *integral*:

0 to 30 = 0.18 or 18%

Meaning 1: 18% of tuples have Salary ≤ 30K
Meaning 2: Each tuple has Salary ≤ 30K *w.p.* 18%

P(Salary = 30K) = 0.12 or 12%



Histogram of Salary (£K)

SELECTION SELECTIVITY

selection selectivity sl(A) of attribute A is a *real* number: $\mathbf{0 \leq sl(A) \leq 1}$

- sl(A) = 0 : *none* of the records satisfies a condition over attribute A.

```
SELECT * FROM EMPLOYEE WHERE Salary = 1,000,000,000
```

- sl(A) = 1: *all* the records satisfy a condition over attribute A

```
SELECT * FROM EMPLOYEE WHERE Salary > 0
```

- sl(A) = x: x% of the records satisfy a condition over attribute A

```
SELECT * FROM EMPLOYEE WHERE Salary = 40000
```

Hence: $\mathbf{0 \leq sl(A) \leq 1}$ *or* as percentage: $\mathbf{0\% \leq sl(A) \leq 100\%}$

***i.e., probability* that a tuple satisfies a selection condition!**

Selection Cardinality

Challenge 1: Given r tuples and a selection condition over A, *predict the expected number* of tuples satisfying this condition **without scanning the file.**

In other words, *predict*: $\boldsymbol{sl}(\mathbf{A}) \cdot \boldsymbol{r}$ *average* number of tuples satisfying a condition over attribute A.

- *Selection Cardinality*: $\boldsymbol{s} = \boldsymbol{r} \cdot \boldsymbol{sl}(\mathbf{A}) \in [\mathbf{0}, \boldsymbol{r}]$

Example: If r = 1,000 employees, sl(Salary=40K) = 0.3,
then s = 300 employees *on average* have salary 40K.

Note: Selectivity prediction is *indeed* difficult! (sometimes, *intractable*); as scientists, we make assumptions and/or approximations ☺

SELECTIVITY PREDICTION

Solution 1 (*no* assumption; approximation):

- Approximate the distribution of values via a *histogram*
- Gain: *accurate* selectivity estimate; Cost: maintenance *overhead*.
- Then: a *good* selectivity estimate is:

$$sl(A = x) \approx P(A = x), \text{ which } \textit{depends} \text{ on the value of } x \in [\min(A), \max(A)]$$

```
SELECT * FROM EMPLOYEE
WHERE    Salary  = 140K
```

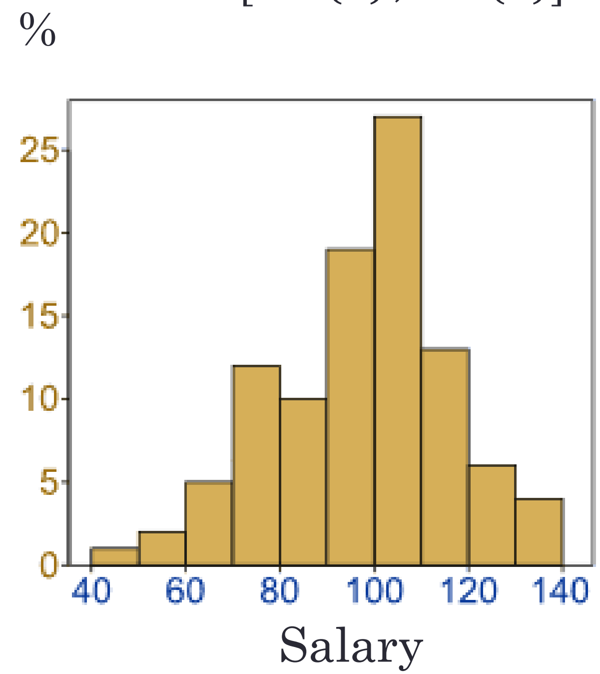
P(Salary=140) = 0.04 = *sl*(Salary = 140);

$s = 0.04 * r$

```
SELECT * FROM EMPLOYEE
WHERE    Salary  = 100K
```

P(Salary=100) = 0.19 = *sl*(Salary = 100);

$s = 0.19 * r$



uniformity assumption

Selectivity Prediction

Solution 2 (uniformity assumption):

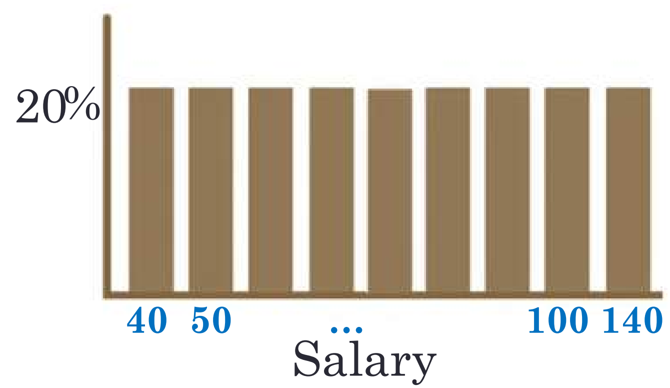
- *All* values are *uniformly* distributed (*equiprobable*), thus, *no* histogram.
- Gain: no need to maintain (update) a histogram ☺
- Impact: provide a *less accurate* prediction for *sl*(A) ☹

$\boldsymbol{sl(\mathrm{A} = x) \approx}$ ***constant*** **independent of the** $\boldsymbol{x}$ **value;** $\forall x \in [\min(A), \max(A)]$

```
SELECT * FROM EMPLOYEE
WHERE    Salary  = 50K
```

P(Salary=50) = 0.2 = $\boldsymbol{sl}$**(Salary = 50);**

$\boldsymbol{s = 0.2 * r}$




```
SELECT * FROM EMPLOYEE
WHERE    Salary  = 100K
```

P(Salary=100) = 0.2 = $\boldsymbol{sl}$**(Salary = 100);**

$\boldsymbol{s = 0.2 * r}$

Selectivity Prediction

uniformity assumption for simplicity

Adopt: Solution 2 (uniformity assumption)

Let an *equality* condition on the **key** attribute A. Then, a *good* estimate is:

$$\boldsymbol{sl(\mathrm{A} = x) = 1/r}, \qquad \forall x \in [\min(A), \max(A)]$$

since *only one* tuple satisfies the condition; *selection cardinality s* = 1 tuple.

Phew…there is *no* meaning to build a histogram.

```
SELECT * FROM EMPLOYEE WHERE SSN = '12345678'
```

Fact: r = 1,000 employees, then sl(SSN) = 1/r = 0.001.

Selectivity Prediction

Equality condition on a **non-key** attribute A, with $\boldsymbol{n}$ **= NDV(A) <** $\boldsymbol{r}$ number of distinct values. Then, a *not-so-good* estimate is:

$$\boldsymbol{sl}(\mathbf{A} = \boldsymbol{x}) = \mathbf{1/NDV(A)} = \mathbf{1}/\boldsymbol{n}, \qquad \forall x \in [\min(A), \max(A)]$$

Because: *all* records are **uniformly distributed** across the *n distinct* values;

- r/NDV(A) = r/n: number of tuples having a distinct value;
- The fraction is: $\mathbf{P(A} = \boldsymbol{x}) = (\boldsymbol{r}/\boldsymbol{n})/\boldsymbol{r} = \mathbf{1}/\boldsymbol{n}$

```
SELECT * FROM EMPLOYEE WHERE DNO = 5;
```

n = NDV(DNO) = 10 departments, r = 1,000 employees **evenly distributed** across the departments. sl(DNO) = 1/10 = **10% (*i.e.,*: 100 employees/department)**

- ***Selection cardinality*** $\boldsymbol{s} = \boldsymbol{r} * \boldsymbol{sel}(\mathbf{A}) = \boldsymbol{r}/\boldsymbol{n}$
- **Check:** If A is a **key**: NDV(A) = n = r; *selection cardinality* = 1 tuple

'…the **probability** of an attribute having *uniform distribution* is *almost* **zero** ☹'

[*] Yannis E. Ioannidis et al; 1996. *Improved histograms for selectivity estimation of range predicates.* ACM SIGMOD'96 NY, USA, 294-305.

uniformity assumption

RANGE SELECTION SELECTIVITY

```
SELECT * FROM RELATION WHERE A ≥ x
```

Definition 1: Domain range: **max(A) – min(A)**; A $\in$ [min(A), max(A)]

Definition 2: Query range: **max(A) – *x*;** $x \in$ [min(A), max(A)]

$$sl(A \geq x) = 0 \text{ if } x > \max(A)$$

$$sl(A \geq x) = (\max(A) - x)/(\max(A) - \min(A)) \in [0, 1]$$

min = 100 x=1000 10000 = max

```
SELECT * FROM EMPLOYEE WHERE Salary ≥ 1000;
```

Salary $\in$ [100, 10000]; r = 1,000 employees **evenly distributed** among salaries:
sl(Salary $\geq$ 1000) = (10000-1000)/(9900) = 0.909 (**90.9%**) *or* s = 909 employees.

uniformity & independency assumption

CONJUNCTIVE SELECTIVITY

```
SELECT * FROM RELATION WHERE (A = x) AND (B = y)
```

$$\boldsymbol{sl}(\mathrm{Q}) = \boldsymbol{sl}(\mathrm{A} = x) \cdot \boldsymbol{sl}(\mathrm{B} = y) \quad \in [0, 1]$$

```
SELECT * FROM EMPLOYEE
WHERE    DNO = 5 AND Salary = 40000;
```

NDV(Salary) = 100, NDV(DNO) = 10, r = 1,000 employees **evenly distributed** among salaries **and** departments:

- *Salary is independent of the department (accept?)*
- sl(Salary = 40K) = 1/NDV(Salary) = 1/100 = 0.01
- sl(DNO = 5) = 1/NDV(DNO) = 1/10 = 0.1

sl(Q) = sl(Salary) · sl(DNO) = (1/10)·(1/100) = 0.001 or only s = 1 tuple.

P(A ∩ B) = P(A)·P(B) = *joint probability an employee satisfying both conditions, given that condition A is statistically independent of condition B.*

DISJUNCTIVE SELECTIVITY

uniformity & independency assumption

```
SELECT * FROM RELATION WHERE (A = x) OR (B = y)
```

$$sl(Q) = sl(A) + sl(B) - sl(A) \cdot sl(B) \quad \in [0, 1]$$

```
SELECT * FROM EMPLOYEE
WHERE    DNO = 5 OR Salary = 40000;
```

NDV(Salary) = 100, NDV(DNO) = 10, r = 1000 employees **evenly distributed** among salaries **and** departments:

- *Salary is independent of the department (accept?)*
- sl(Salary) = 1/NDV(Salary) = 1/100 = 0.01
- sl(DNO) = 1/NDV(DNO) = 1/10 = 0.1

sl(Q) = (10/100) + (1/100) - (1/10)·(1/100) = 0.109 or s = 109 tuples.

P(A ∪ B) = P(A) + P(B) - P(A)·P(B) = *probability an employee satisfying either condition A or condition B; both conditions are statistically independent.*

So Far…

Challenge 1: *Predict the number of tuples satisfying a selection*!

Assumption: The tuples are *uniformly distributed* across the values of attribute A.

Selection Selectivity: 1/NDV(A) = $1/n$

Selection Cardinality: (1/NDV(A)) $\cdot\ r = r/n$

For a **key** attribute, n = NDV(A) = r thus selection cardinality = 1 tuple

For a **non-key** attribute, n = NDV(A) < r thus selection cardinality > 1 tuple

QUIZ: `SELECT * FROM EMPLOYEE WHERE DNO <> 5`

In-Class Activity [A1]

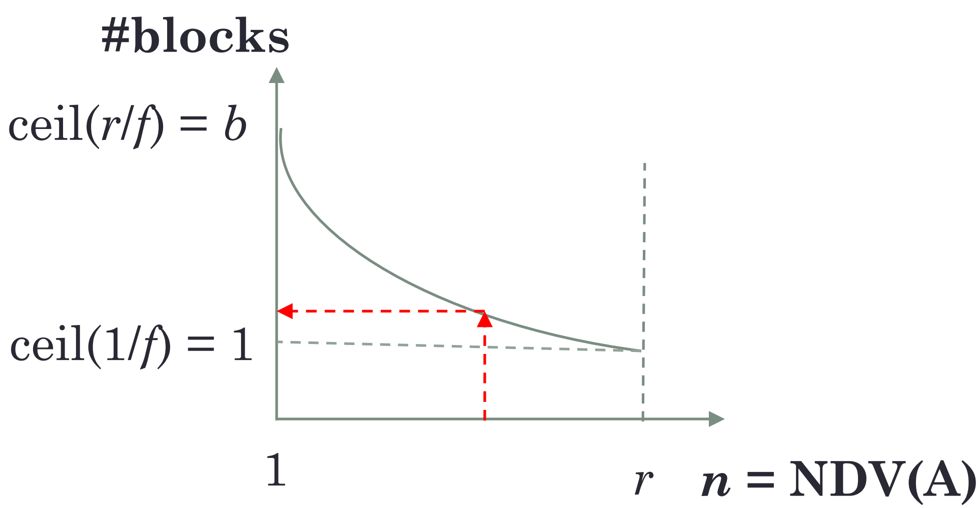
Challenge 2: *Predict the number of blocks satisfying a selection* (predict cost)!

Context: Blocking factor f tuples/block, selection cardinality $s = (1/\mathrm{NDV(A)}) \cdot r = r\,/\,n$

The *predicted* number of blocks retrieved is a *reciprocal function* of n:

$$\mathbf{ceil}(\, s\,/\,f\,) = \mathbf{ceil}(\, r\,/\,(f \cdot \mathrm{NDV(A)})\,) = \left\lceil \frac{r}{nf} \right\rceil \ \textbf{blocks}$$

association of cost with the attribute's characteristic!



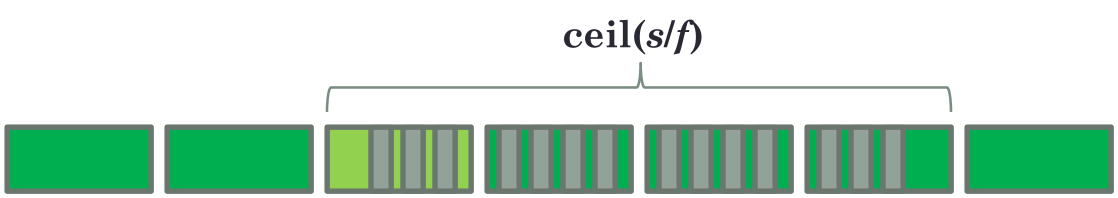
SELECTION COST REFINEMENT

Target: be *more* accurate! **express cost as a *function* of *sl*(A) = *r*/*n***

```
SELECT * FROM RELATION WHERE A = x
```

Context: b blocks, f blocking factor (tuples/block), r records, n = NDV(A)

- **Binary Search where the relation is sorted w.r.t. A:**
  - If A is a **key**, then **Expected Cost: $\log_2(b)$ block accesses (ind. *sl*(A))**
  - If A is **not a key**, then
  - **$\log_2(b)$ block accesses** to reach the ***first*** block with record(s) A = x
  - Access *all* ***contiguous*** blocks whose records satisfy: A = x.
  - *Selection cardinality* **$s = r \cdot sl(A = x)$ tuples** .
  - Blocking factor: f tuples/block: access **ceil(s/f) - 1** more blocks:



Expected Cost: **$\log_2(b)$ + ceil(s/f) -1 = $\log_2(b)$ + ceil($r \cdot sl(A)/f$) -1**

SELECTION COST REFINEMENT

Multilevel Primary Index of level: t over the ***key*** A and *equality* A = x

Expected Cost: t + 1 **(ind. sl(A))**

File Blocks (records)

Hash File Structure

- Apply the hash function h(A) over the ***key*** A and retrieve the block.

Expected cost: 1 **(ind. sl(A))**

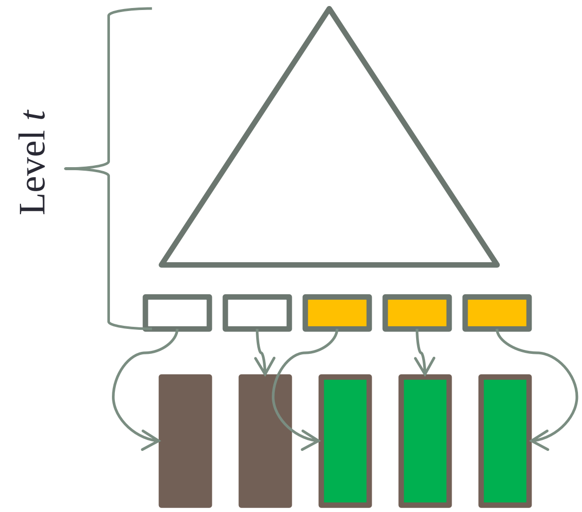
best case; *no* overflown buckets

Selection Cost Refinement

Multilevel Primary Index of level: *t* over the ***key*** A with range A ≥ *x*

```
SELECT * FROM EMPLOYEE WHERE SSN >= 10
```

- Tree traversal: ***t* block accesses (SSN = 10)**
- **Range** selection cardinality: **s = *r* ·*sl*(A)**
- Blocking factor: *f* records/block: **ceil(*s*/*f*) blocks**



File Blocks (records)

Expected Cost: *t* + ceil(*s*/*f*) = *t* + ceil(*r* ·*sl*(A) /*f*) block accesses.

Note: *sl*(A) is the *range* selection selectivity.

Selection Cost Refinement

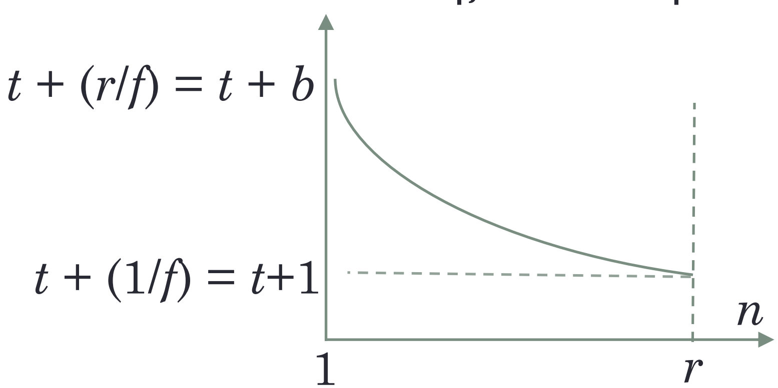
Clustering Index over a ***ordering, non-key***

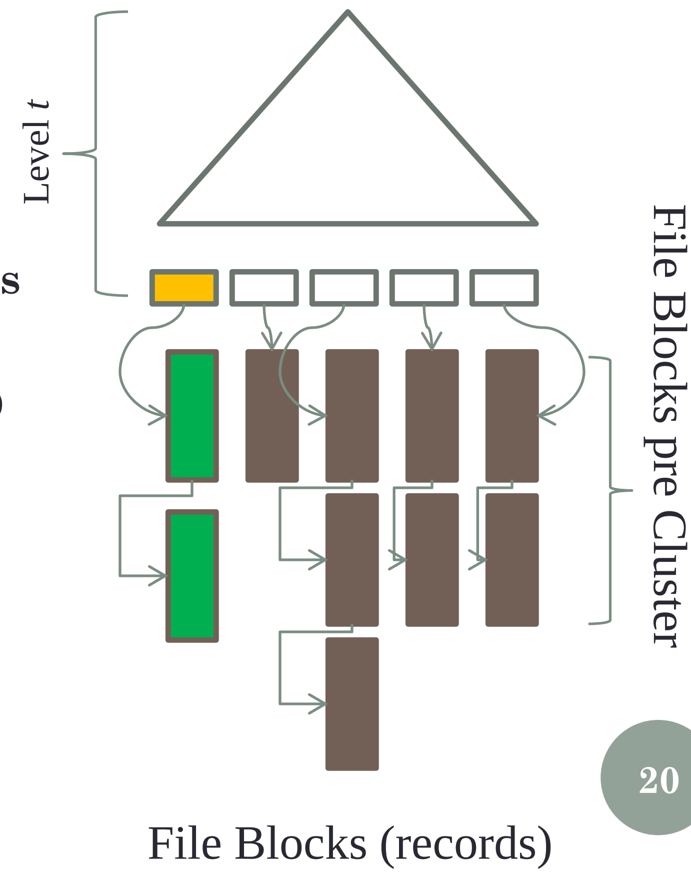
```
SELECT * FROM EMPLOYEE WHERE DNO = 3
```

- Tree traversal: ***t* block accesses.**
- *Selection* cardinality ***s* = *r* ·*sl*(A) tuples**
- Blocking factor: *f* records/block: **ceil(*s*/*f*) blocks**

Expected Cost: t + ceil(s/f) = t + ceil($r \cdot sl$(A) /f)

Fine-grained Cost: $t + \left\lceil \frac{r}{f} P(A = x) \right\rceil$





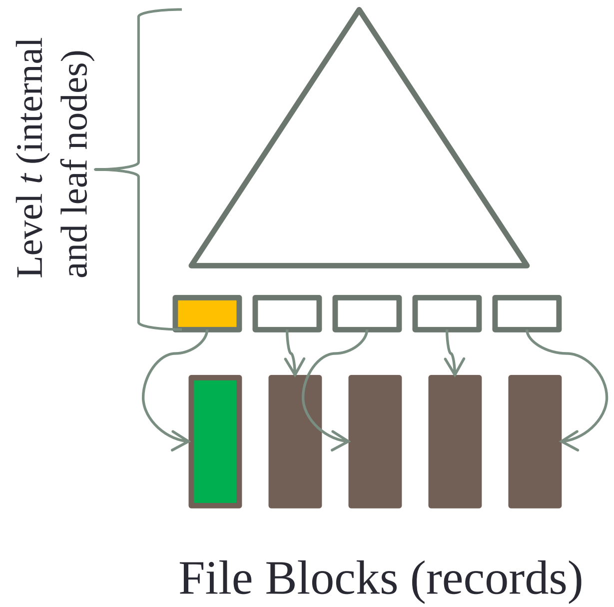
Selection Cost Refinement

B+ Tree over a ***non-ordering, key*** with *equality* A=*x*

```
SELECT * FROM EMPLOYEE WHERE SSN = '12345'
```

- Tree traversal: ***t* block accesses.**
- *One* data block since $s = 1$

Expected cost: $t + 1$ **(ind. *sl*(A))**

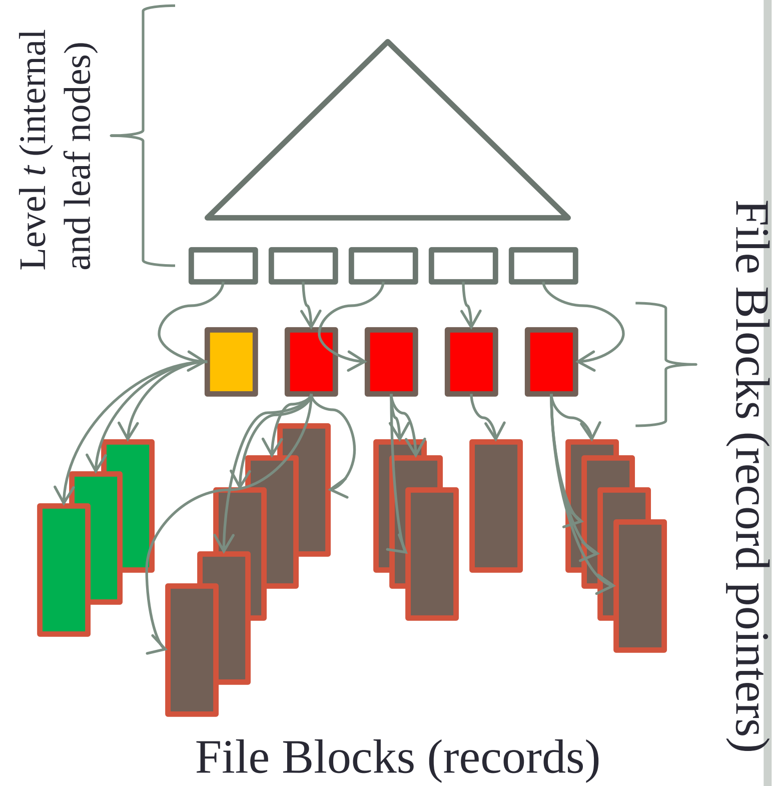


Selection Cost Refinement

B+ Tree over a ***non-ordering, non-key*** with *equality* A = x

```
SELECT * FROM EMPLOYEE
WHERE    Salary = 40000
```

- Tree traversal: ***t* block accesses.**
- **1 block access** to *load* the block of block pointers.
- *Selection* cardinality $\boldsymbol{s} = \boldsymbol{r} \cdot \boldsymbol{sl}(\mathrm{A})$ **tuples**
- Each tuple *may* be in a *different* data block (*worst case*) thus, access up to ***s*** blocks



Expected Cost: $\boldsymbol{t} + 1 + \boldsymbol{s} = \boldsymbol{t} + 1 + \boldsymbol{r} \cdot \boldsymbol{sl}(\mathrm{A})$

Optimization Example [E1]

Employee r = 10,000 records, b = 2,000 blocks, blocking factor f = 5

Access Paths:

- **Clustering Index on non-key Salary**: x_{Salary} = 3 levels
- selectivity sl(Salary) = 0.002, cardinality s_{Salary} = 20 tuples (per salary).

- **B+ Tree on non-key DNO:** x_{Dno} = 2 levels
- NDV(DNO) = 125, cardinality s_{Dno} = r/NDV(DNO) = 80 tuples (employees per department), selectivity sl(DNO) = 1/125=0.008.

- **B+ Tree on non-key EXP:** x_{EXP} = 2 levels
- NDV(EXP) = 2, cardinality s_{EXP} = 5000 tuples, selectivity sl(EXP) = 0.5

Note: Relation is sorted w.r.t. Salary; Clustering Index on Salary ☺

Note: EXP $\in$ {0, 1} stands for *experienced* or *inexperienced* employee

OPTIMIZATION EXAMPLE [E1]


```
SELECT * FROM EMPLOYEE
WHERE    DNO = 5 AND SALARY = 30000 AND EXP = 0
```

Memory: 100 blocks; relation *sorted* by Salary.

Task: Propose alternative plans and choose the *best*.

P0: Linear Search: b = **2,000 block accesses;** *can we do better*?

Tip: cost for *each* condition and *then* reason about the intermediate result size

- **P1:** 'DNO = 5'; B+ Tree non-key: $\boldsymbol{x_{DNO} + 1 + s_{DNO}}$ **= 83 block accesses**
- **Intermediate result:** $\boldsymbol{s_{DNO}}$ **= 80 tuples** or ceil(80/5) = **16 blocks** fit in memory!
- Apply the rest conditions **in-memory:** 'SALARY = 30000 AND EXP = 0'
- **Cost-P1: 83 block accesses**

- **P2:** 'SALARY = 30000', Clustering index: $\boldsymbol{x_{Salary} + \mathrm{ceil}(s_{Salary}/f)}$ **= 7 block accesses**
- **Intermediate result:** $\boldsymbol{s_{Salary}}$ **= 20 tuples** or ceil(20/5) = **4 blocks** fit in memory!
- Apply the rest conditions **in-memory:** 'DNO = 5 AND EXP = 0'
- **Cost-P2: 7 block accesses**

OPTIMIZATION EXAMPLE [E1]


```
SELECT * FROM EMPLOYEE
WHERE   DNO = 5 AND SALARY = 30000 AND EXP = 0
```

Memory: 100 blocks;

P3: ‘EXP = 0’ ; B+ Tree non-key: $\boldsymbol{x}_{\mathbf{EXP}}$ **+ 1 +** $\boldsymbol{s}_{\mathbf{EXP}}$ = **5,003 block accesses**

Intermediate result: $\boldsymbol{s}_{\mathbf{EXP}}$ **= 5000 tuples** or ceil(5000/5) = **1000 blocks** *not* fit in memory, *thus*, write back to disk: 1000-100 = **900 blocks**

Cost-P3: ***at least*** **5,003 + 900 block accesses…** ***stop*** **there** ☹

- **Cost-P0:** 2000 block accesses
- **Cost-P1:** 83 block accesses
- **Cost-P2:** 7 block accesses
- **Cost-P3:** *at least* 5,903 block accesses

Best Plan: Plan 2

Optimization Example [E2] (1/6)


```
SELECT * FROM EMPLOYEE
WHERE    DNO = 5 OR (SALARY >= 500 AND EXP = 0)
```

Memory: 1100 blocks; relation *sorted* by Salary.

Task 1: Estimate the query selectivity and estimate the *ideal* block accesses!

Conjunctive selectivity: *slc* = *sl*(Salary) * *sl*(EXP)

- Salary in [100, 10000]
- *sl*(Salary) = (10000-500)/(9900) = 0.959 (*range selectivity*)
- *sl*(EXP) = 0.5
- *slc* = 0.959*0.5 = 0.4795.

- *sl*(DNO) = 0.008

Disjunctive: *sl*(DNO) + *slc* – *sl*(DNO) * *slc* = 0.4836 or **4,836.6 employees**

Ideally: we *desire* to retrieve *only* ceil(4836.6/5) = **968 blocks (where are they?)**

Challenge: find a plan to be *as close to* **968 block accesses** *as possible*!

Optimization Example [E2] (2/6)


```
SELECT * FROM EMPLOYEE
WHERE    DNO=5 OR (SALARY >= 500 AND EXP = 0)
```

Memory: 1100 blocks

Task 2: Find *the* Optimal Plan

Baseline: Linear Search: ***b* = 2,000 block accesses**

Disjunctive Strategy: *union* of partial results corresponding to OR conditions

Condition 1: `DNO = 5`

Plan 1: B+ Tree (DNO): $\boldsymbol{x_{DNO} + s_{DNO} + 1 = 83}$ **block accesses**

Intermediate result: 80 tuples stored in ceil(80/5) = **16 blocks in memory**

Condition 2: `SALARY >= 500 AND EXP = 0`

Conjunctive Strategy: examine *both* conditions; filter-out intermediate result

- **Plan 2.1: 1st** `EXP = 0` **and 2nd** `SALARY >= 500`
- **Plan 2.2: 1st** `SALARY >= 500` **and 2nd** `EXP = 0`

OPTIMIZATION EXAMPLE [E2] (3/6)

`SELECT * FROM EMPLOYEE`

`WHERE DNO=5 OR (SALARY >= 500 AND EXP = 0);` **Memory: 1100 blocks**

Plan 2.1 **1st** `EXP = 0` **and 2nd** `SALARY >= 500`

Use B+ Tree 'EXP = 0': $x_{EXP} + s_{EXP} + 1$ **= 5,003 block accesses**

Intermediate result: 5000 tuples stored in ceil(5000/5) = **1000 blocks in memory**

Check Available memory: 1100-16 = 1084 blocks free > 1000…*phew*!

Filter: `SALARY >= 500` and *union* with tuples of Plan 1; done!

- **Total Cost A = Plan 1 + Plan 2.1** = 83 + 5,003 = **5,086 block accesses** ☹
- **Memory requirements: 16 + 1000 = 1016 blocks (*affordable*)**
- Linear Search: *b* = **2,000 block accesses** ☺

Plan 2.2: **1st** `SALARY >= 500` **and** **2nd** `EXP = 0` (4/6)

Use Clustering Index 'SALARY >= 500': $\boldsymbol{x_{\text{Salary}}}$ **+ ceil(*s*/*f*) = 1921 block accesses**

Intermediate: *sl*(Salary) = 0.959 or *s* = 9590 tuples stored in **1918 blocks**

- **Check Memory: 1100-16 = 1084 blocks available < 1918 blocks**
- **Store in memory 1084 blocks to be filtered out & write to disk: 1918-1084 = 834 block accesses (write)**
- **would you like to stop? 1921 + 834 = 2755 > 2000 (linear search) …**
- **Remaining:** 834 blocks in disk to be filtered out

1084 in memory | 834 in disk

Filtering Loop 1:

- **Filter**: `EXP = 0` over **1084 blocks & discard 0.5*1084 = 542 blocks**
- **Free-up memory:** read 542 out of remaining 834, thus, **542 block accesses (read)**
- **Remaining:** 834-542 = 292 blocks in disk to be filtered out

542 in memory | 292 in disk

Filtering Loop 2:

- **Filter**: `EXP = 0` over **542 blocks & discard 0.5*542 = 271 blocks**
- **Free-up memory:** read 271 out of 292 from disk, thus, **271 block accesses (read)**
- **Remaining:** 292-271 = 21 blocks in disk to be filtered out

271 in memory	21 in disk

Filtering Loop 3:

- **Filter**: `EXP = 0` over **271 blocks & discard 0.5*271 = 135 blocks**
- **Free-up memory:** read 21 out of 21 from disk, thus, **21 block accesses (read)**
- **Remaining:** 0 blocks in disk to be filtered out
- **Memory**: 16 + 542 + 271 + 135 + 21*0.5 = 975 blocks…*phew*!

21 in memory	0 in disk

Reasoning: (6/6)

- **Condition 1: 83 block accesses**
- **Condition 2: Plan 2.1: 5,003 block accesses**
- **Condition 2: Plan 2.2: 1921 + 834 + 542 + 271 + 21 = 3,589 block accesses**

- **Total Cost A = Condition 1 + Condition 2.1 = 5,086 block accesses** ☹
- **Total Cost B = Condition 1 + Condition 2.2 = 3,672 block accesses** ☹
- **Linear Search: 2,000 block accesses** ☺
- **Ideal Search: 968 block accesses**

Current best plan is: Linear Search.

Action: Increase memory, e.g., from 1100 to 2000 blocks to *avoid* filtering loops

- **Plan 2.2' intermediate result is 1918 blocks fit in memory!**
- **Total Cost B' = Cond. 1 + Cond. 2.2' = 83 + 1921 = 2004 block accesses** ☹

Again, best plan is: Linear Search, *even* if you ask for *infinite* ∞ memory!

Critical Thinking: the source of the problem is the complexity of the selection involving OR's, *ranges*, AND's …over multiple attributes ☹

Query Optimization: Part II

Database Systems (H)
Dr Chris Anagnostopoulos

ROADMAP

- **Join Selectivity**: *fraction* of the matching tuples in Cartesian space.
- **Challenge 1**: *predict* the *number* of matching tuples of a join query, i.e., *join cardinality*.
- **Challenge 2**: *express* the expected cost of the ***all*** strategies as a function of *selection selectivity* and *join selectivity*
- **Optimization Plans:** *selection* and *join* queries; 3-*way* join queries.

EMPLOYEE

SSN	Name	DNO
1	Chris	D1
2	Stella	D1
3	Phil	D2
4	Thalia	D3
5	John	D3

```
SELECT * FROM
EMPLOYEE E, DEPARTMENT D
```

X

DEPARTMENT

DNUMBER	MGR_SSN	DNAME
D1	1	HR
D2	3	R&D
D3	4	SPR

Cartesian cardinality =
$|\mathbf{E} \times \mathbf{D}| = |\mathbf{E}| \cdot |\mathbf{D}| = 15$ *concatenated* tuples

EMPLOYEE x DEPARTMENT

SSN	Name	DNO	DNUMBER	MGR/SSN	DNAME
1	Chris	D1	D1	1	HR
1	Chris	D1	D2	3	R&D
1	Chris	D1	D3	4	SPR
2	Stella	D1	D1	1	HR
2	Stella	D1	D2	3	R&D
…	…	…	…	…	…
5	John	D3	D3	4	SPR

matching tuple

non-matching tuple

15

EMPLOYEE

SSN	Name	DNO
1	Chris	D1
2	Stella	D1
3	Phil	D2
4	Thalia	D3
5	John	D3

```
SELECT * FROM
EMPLOYEE E, DEPARTMENT D
WHERE E.DNO = D.DNUMBER
```

$\bowtie$

DEPARTMENT

DNUMBER	MGR_SSN	DNAME
D1	1	HR
D2	3	R&D
D3	4	SPR

join cardinality = |**E** ⋈ **D**| = 5 *matching* tuples

join selectivity = |**E** ⋈ **D**|/|**E** × **D**| = **5/15 = 0.333**

i.e., probability of selecting a matching tuple out of all tuples in the Cartesian space

EMPLOYEE ⋈ DEPARTMENT

SSN	Name	DNO	DNUMBER	MGR/SSN	DNAME
1	Chris	D1	D1	1	HR
2	Stella	D1	D1	1	HR
3	Phil	D2	D2	3	R&D
4	Thalia	D3	D3	4	SPR
5	John	D3	D3	4	SPR

5

EMPLOYEE

SSN	Name	DNO
1	Chris	D1
2	Stella	D1
3	Phil	D2
4	Thalia	D3
5	John	D3

```
SELECT * FROM
EMPLOYEE E, DEPARTMENT D
WHERE E.SSN = D.MGR_SSN
```

$\bowtie$

DEPARTMENT

DNUMBER	MGR_SSN	DNAME
D1	1	HR
D2	3	R&D
D3	4	SPR

join cardinality = $|\mathbf{E} \bowtie \mathbf{D}|$ = 3 *managers*

join selectivity = $|\mathbf{E} \bowtie \mathbf{D}| / |\mathbf{E} \times \mathbf{D}|$ = **3/15 = 0.2**

i.e., probability of an employee being manager out of all concatenated tuples

EMPLOYEE $\bowtie$ DEPARTMENT

SSN	Name	DNO	DNUMBER	MGR/SSN	DNAME
1	Chris	D1	D1	1	HR
3	Phil	D2	D2	3	R&D
4	Thalia	D3	D3	4	SPR

3

join

JOIN SELECTIVITY & CARDINALITY

Join query: $\mathbf{R} \bowtie \mathbf{S}$ and Cartesian product: $\mathbf{R} \times \mathbf{S}$

- `SELECT * FROM R, S`
- `SELECT * FROM R, S WHERE R.A = S.B`

Definition 1: *join selectivity* (*js*) is the *fraction* of the matching tuples between the relations **R** and **S** *out* of the Cartesian cardinality (#of concatenated tuples):

$$js = |\mathbf{R} \bowtie \mathbf{S}| / |\mathbf{R} \times \mathbf{S}| \text{ with } 0 \leq js \leq 1.$$

Cartesian cardinality: $|\mathbf{R} \times \mathbf{S}| = |\mathbf{R}| \cdot |\mathbf{S}|$

Definition 2: *join cardinality* $jc := js \cdot |\mathbf{R}| \cdot |\mathbf{S}|$

Challenge 1: Predict the join cardinality (*jc*) *without* executing the join query.

JOIN SELECTIVITY THEOREM


join

Theorem 1. Given n = NDV(A, **R**) and m = NDV(B, **S**):

$$js = 1 / \max(n, m)$$

$$jc = (|\mathbf{R}| \cdot |\mathbf{S}|) / \max(n, m)$$

Proof. *Beyond the scope…(last slide)*

Example: Show the dependents of each employee;

Note: an employee might have *zero* to *many* dependents.

```
SELECT * FROM EMPLOYEE E, DEPENDENT P
WHERE    E.SSN = P.E_SSN
```

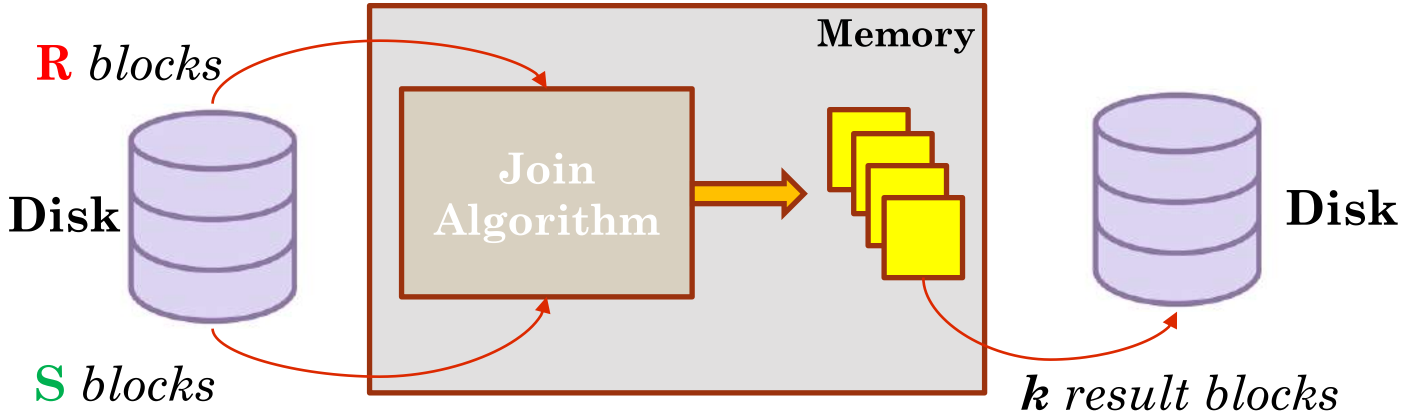
n = NDV(SSN,**E**) = 2000; |**E**| = 2000 employees

m = NDV(E_SSN, **P**) = 3; |**P**| = 5 dependents

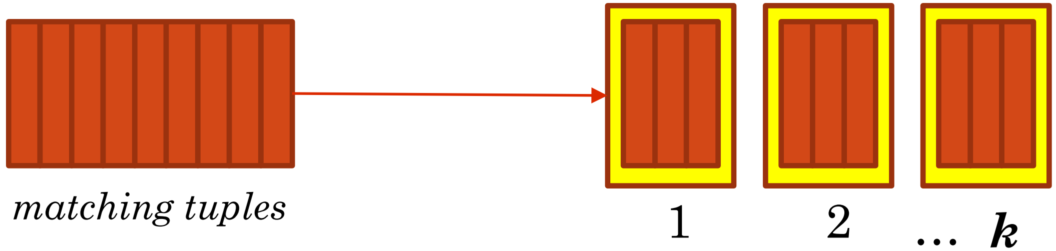
js = 1/max(2000,3) = 1/2000 = 0.0005 or **0.05% (probability of matching tuple)**

jc = 0.0005 * 2000 * 5 = 5 matching tuples (*as expected*)

- Relation **R** and **S** with $\boldsymbol{b}_{\mathrm{R}}$ and $\boldsymbol{b}_{\mathrm{S}}$ blocks; **R.A = S.B**
- Memory: $\boldsymbol{n}_{\mathrm{B}}$ blocks; n = NDV(**R.A**), m = NDV(**S.B**)
- Result block: *blocking factor* $\boldsymbol{f}_{\mathrm{RS}}$ **matching tuples/block**.
- *Size* of matching tuple (r, s) = *size* of tuple $r \in$ **R** + *size* of tuple $s \in$ **S**



- *Write* every *full result block* to disk. **How many result blocks do we *write*?**
- Matching tuples: $jc = js \cdot |\mathbf{R}| \cdot |\mathbf{S}| = (1/\max(n, m)) \cdot |\mathbf{R}| \cdot |\mathbf{S}|$
- #result blocks: $\boldsymbol{k} = (\boldsymbol{js} \cdot |\mathbf{R}| \cdot |\mathbf{S}|) / \boldsymbol{f}_{\mathrm{RS}}$



Join Cost Refinement

Strategy: Nested-Loop Join

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D
WHERE    E.SSN = D.MGR_SSN
```

- **D** is the *outer* relation, i.e., $b_D < b_E$ with *outer* loops: $\mathbf{ceil}(b_D/(n_B\text{-}2))$

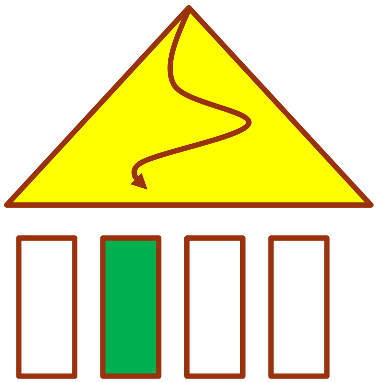
Expected Cost: $b_D + (\mathrm{ceil}(b_D/(n_B-2)) \cdot b_E$

- Matching tuples: $jc = js \cdot |\mathbf{E}| \cdot |\mathbf{D}| = (1/\max(n, m)) \cdot |\mathbf{E}| \cdot |\mathbf{D}|$
- Number of result blocks: $k = (js \cdot |\mathbf{E}| \cdot |\mathbf{D}|) / f_{RS}$
- Include ***k*** result-block ***writes*** from *memory* to *disk* during the execution.

Refined Expected Cost: $b_D + (\mathrm{ceil}(b_D/(n_B-2)) \cdot b_E + (js \cdot |\mathbf{E}| \cdot |\mathbf{D}| / f_{RS})$

Join Cost Refinement

Strategy: Index-based Nested-Loop Join



Primary Index on `MGR_SSN` with x_D levels

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D
WHERE     E.SSN = D.MGR_SSN
```

For *each* employee *e*, use index to *check* if they are a manager.

- Matching tuples: $jc = js \cdot |E| \cdot |D| = (1/\max(n, m)) \cdot |E| \cdot |D|$
- Number of result blocks: $k = (\boldsymbol{js} \cdot |\mathbf{E}| \cdot |\mathbf{D}|) / \boldsymbol{f}_{RS}$

Case: Primary Index on **ordering / key**:

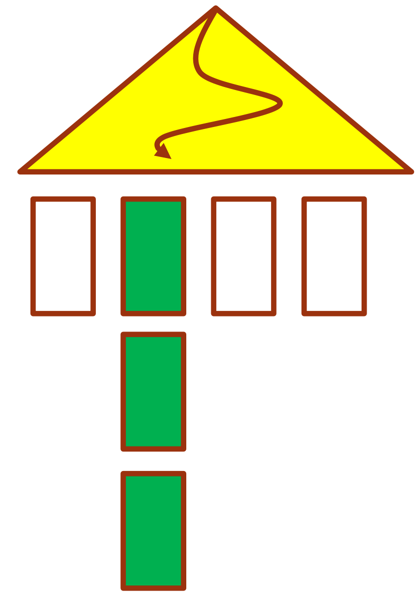
Refined Expected Cost: $\boldsymbol{b}_E + |\mathbf{E}| \cdot (\boldsymbol{x}_D + 1) + (\boldsymbol{js} \cdot |\mathbf{E}| \cdot |\mathbf{D}| / \boldsymbol{f}_{RS})$

Join Cost Refinement

Strategy: Index-based Nested-Loop Join

Clustering Index on `DNO` with $\boldsymbol{x}_{\mathrm{E}}$ levels,
selection cardinality $\boldsymbol{s}_{\mathrm{E}}$, blocking factor $\boldsymbol{f}_{\mathrm{E}}$

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D
WHERE    E.DNO = D.DNUMBER
```



- Selection cardinality of DNO $\boldsymbol{s}_{\mathrm{E}} = (1/\mathrm{NDV(DNO)}) \cdot |\mathbf{E}|$

For *each* department *d*, use index to *load* its employees.

Case: Clustering Index on **ordering / non-key**

- Selection cardinality := employees per department: $\boldsymbol{s}_{\mathrm{E}}$
- Blocks of employees *per* department: $\mathbf{ceil}(\boldsymbol{s}_{\mathrm{E}} / \boldsymbol{f}_{\mathrm{E}})$
- Number of result blocks: $k = (\boldsymbol{js} \cdot |\mathbf{E}| \cdot |\mathbf{D}|) / \boldsymbol{f}_{\mathrm{RS}}$

Refined Expected Cost: $\boldsymbol{b}_{\mathrm{D}} + |\mathbf{D}| \cdot (\boldsymbol{x}_{\mathrm{E}} + \mathbf{ceil}(\boldsymbol{s}_{\mathrm{E}} / \boldsymbol{f}_{\mathrm{E}})) + (\boldsymbol{js} \cdot |\mathbf{E}| \cdot |\mathbf{D}| / \boldsymbol{f}_{\mathrm{RS}})$

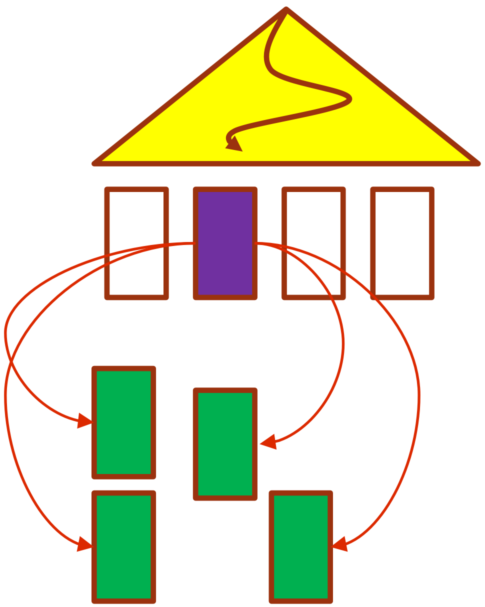
JOIN COST REFINEMENT

Strategy: Index-based Nested-Loop Join

B+ Tree on **DNO** with x_E levels,

selection cardinality s_E, blocking factor f_E

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D
WHERE    E.DNO = D.DNUMBER
```



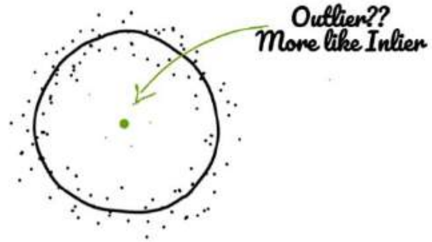
- Selection cardinality of DNO $s_E = (1/\mathrm{NDV(DNO)}) \cdot |E|$

For *each* department *d*, use the index to *load* its employees.

Case: B+ Tree Index on **non-ordering / non-key**

- **1 block** (block of pointers) + **blocks** of employees: s_E
  - *each* employee belongs to a different block (worst case)
- Number of result blocks: $k = (js \cdot |E| \cdot |D|) / f_{RS}$

Refined Expected Cost: $b_D + |D| \cdot (x_E + 1 + s_E) + (js \cdot |E| \cdot |D| / f_{RS})$



OUTLIER SLIDE…

Look at this…

$\boldsymbol{b_D + |D| \cdot (x_E + 1 + s_E) + (js \cdot |E| \cdot |D| / f_{RS})}$

n = NDV(DNO); m = NDV(DNUMBER)

Re-write as:

$\boldsymbol{b_D + |D| \cdot (x_E + 1 + (1/n) \cdot |E|) + ((1/\max(n, m)) \cdot |E| \cdot |D| / f_{RS})}$

Need: predictor should *identify* ***extremely*** *fast the* number of distinct values…
Computing Science: 'The Count Distinct Problem'

- Hash-based Method: **O(*r*)** …does not scale as $r \to \infty$
- HyperLogLog Method: **O(log(log *r*) + log*r*)** …does scale out!

[*] S. Heule et al (2013) *HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm*. 16th ACM EDBT'13. NY, 683-692.

Join Cost Refinement

Strategy: Sort-Merge: *both files are sorted* on joining attributes A and B

- **Refined Expected Cost:** $\boldsymbol{b_R + b_S + (js \cdot |R| \cdot |S| / f_{RS})}$

Strategy: Hash-Join: *both files are hashed* w.r.t. same hash function *h*;

- **Refined Expected Cost:** $\boldsymbol{3 \cdot (b_R + b_S) + (js \cdot |R| \cdot |S| / f_{RS})}$

IN-CLASS EXAMPLE [E1]

Employee r_E = 10,000 employees, b_E = 2,000 blocks.
Department r_D = 125 departments, b_D = 13 blocks.
Result block: f_{RS} = 4 matching records/block (blocking factor).
Memory: n_B = 10 blocks

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D
WHERE    E.DNO = D.DNUMBER
```

Access Paths:

- Primary Index on **DNUMBER** $x_{Dnumber}$ = 1 level.
- B+ Tree Index on **DNO** x_{Dno} = 2 levels.
- Selectivity sl(DNO) = 1/ NDV(DNO) = 1/125 = 0.008.
- Selection cardinality $s_{DNO} = sl(\text{DNO}) * r_E$ = 80 employees per department.

Task: Express the join cost involving *selection* & *join* selectivities.

In-class Example [E1]

Join selectivity & Join cardinality:

- js = 1/ max(NDV(**DNO**), NDV(**DNUMBER**))
- n = NDV(**DNO**) = 125; m = NDV(**DNUMBER**) = 125
- js = 1/max(125,125) = 1/125 = 0.008

Note: 0.008 is the probability of finding a matching tuple in Cartesian space

- $jc = js * r_E * r_D$ = **10,000 matching tuples**
- Number of result blocks: jc/f_{RS} = 10,000/4 = **2,500 result blocks (write)**

EXAMPLE

In-class Example [E1]

Nested-loop Join:

- Department *outer*: $b_D + (\text{ceil}(b_D/(n_B\text{-}2)) \cdot b_E + (js \cdot r_E \cdot r_D / f_{RS})$ = **6,513 block accesses**

Index-based Nested-loop Join *with* Employee as outer

- **Primary Index (DNUMBER):** $b_E + r_E \cdot (x_{Dnumber}+1) + (js \cdot r_E \cdot r_D / f_{RS})$ = **24,500 block accesses**

Index-based Nested-loop Join *with* Department as outer

- **B+ Tree Index (DNO):** $b_D + r_D \cdot (x_{DNO} + s_{DNO} + 1) + (js \cdot r_E \cdot r_D / f_{RS})$ = **12,888 block accesses**

Hash-Join: $3 \cdot (b_D + b_E) + (js \cdot r_E \cdot r_D / f_{RS})$ = **8,539 block accesses**

Sort-Merge: Cannot be used. *Why? (DNO is not a sorting attribute)*

Best strategy: Nested-loop Join with Department as *outer* relation

3-WAY JOIN OPTIMIZATION (1/5)

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D, DEPENDENT T
WHERE    T.E_SSN = E.SSN AND E.SSN = D.MGR_SSN
```

- DEPENDENT r_T = 50 dependents, b_T = 3 blocks.
- Clustering Index (E_SSN) x_{E_SSN} = 2 levels, NDV(E_SSN) = 10, s_{E_SSN} = 5 dependents *per* employee.

- DEPARTMENT r_D = 125 departments, b_D = 13 blocks.
- Primary Index (MGR_SSN) x_{MGR_SSN} = 1 level.

- EMPLOYEE r_E = 10,000 employees, b_E = 2,000 blocks, B+ Tree (SSN) x_{SSN} = 4 levels

- Blocking factor: f = 10; Result block: f_{RS} = 2; Memory: n_B = 100 blocks

Observation: 3-way join ☺

Plan 1: *find* employees having at least one dependent; *check* if they are managers.

Plan 2: *find* employees being managers; *check* if these managers have dependents.

3-WAY JOIN OPTIMIZATION (2/5)

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D, DEPENDENT T
WHERE    T.ESSN = E.SSN AND E.SSN = D.MGR_SSN
```

Plan 1: *find* employees having at least one dependent; *check* if they are managers.

Plan 1.1: Join employees with their dependents:

- ***js* = 1/max(10,10000) = 0.01%** or ***jc* = 50 matching tuples.**

- For *each* dependent, get *their* employee
- B+ Index (SSN): $b_T + r_T \cdot (x_{SSN}+1) + jc/f_{RS}$ = **278 block accesses**

- For *each* employee, get *their* dependent(s)
- Clustering Index (E_SSN): $b_E + r_E \cdot (x_{E\_SSN} + \text{ceil}(s_{E\_SSN}/f)) + jc/f_{RS}$ = **32,025 block accesses**

- **Intermediate results: r_{ET} = 50 tuples (employees-with-dependents)**
- Memory requirement: **$b_{ET} = \text{ceil}(r_{ET}/ f_{RS})$ = 25 blocks in memory**

3-WAY JOIN OPTIMIZATION (3/5)

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D, DEPENDENT T
WHERE    T.ESSN = E.SSN AND E.SSN = D.MGR_SSN
```

Plan 1: *find* employees having at least one dependent; *check* if they are managers.

Plan 1.2: Join employees-with-dependents (b_{ET} = 25 blocks) with department.

- Take a *joint* employee at a time, and ask if they are a manager.

- js = 1/max(10000, 125) = 0.01% *statistical probability* of an employee being a manager.
- $jc_{ETD} = js * r_{ET} * r_D$ = **0.625 matching tuples** or ceil(0.625/2) = **1 block**

- Primary Index (MGR_SSN): $b_{ET} + r_{ET} \cdot (x_{MGR\_SSN} + 1) + (jc_{ETD} / f_{RS})$?
- **Note 1:** b_{ET} is already in the memory, thus, *excluded* from the *read* cost.
- **Note 2:** (jc_{ETD} / f_{RS}) result block is only 1, thus, *excluded* from the *write* cost.
- **Note 3:** Cost is *only* for asking the Primary Index = **100 block accesses**

Total Cost Plan 1: 278 + 100 = 378 block accesses ☺

3-WAY JOIN OPTIMIZATION (4/5)

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D, DEPENDENT T
WHERE    T.ESSN = E.SSN AND E.SSN = D.MGR_SSN
```

Plan 2: *find* employees being managers; *check* if these managers have dependents.

Plan 2.1: Join employees with departments:

- ***js*** **= 1/max(125,10000) = 0.01%** or ***jc*** **= 125 managers or 63 blocks.**

- For *each* department, get *its* manager's info
- B+ Index (SSN): $b_D + r_D \cdot (x_{SSN} + 1) + (jc / f_{RS})$ = **701 block accesses**

- For *each* employee, check if they are a *manager*
- Primary Index (MGR_SSN): $b_E + r_E \cdot (x_{MGR\_SSN} + 1) + (jc / f_{RS})$ = **22,063 block accesses**

- **Intermediate results:** $\boldsymbol{r_{ED}}$ **= 125 managers**
- Memory requirement: $\boldsymbol{b_{ED}}$ **= ceil(**$\boldsymbol{r_{ED} / f_{RS}}$**) = 63 blocks in memory**

3-WAY JOIN OPTIMIZATION (5/5)

```
SELECT * FROM EMPLOYEE E, DEPARTMENT D, DEPENDENT T
WHERE    T.ESSN = E.SSN AND E.SSN = D.MGR_SSN
```

Plan 2: *find* employees being managers; *check* if these managers have dependents.

Plan 2.2: Join managers (b_{ED} = 63 blocks; r_{ED} = 125 managers) with dependents.

- Take a *manager* at a time, and ask if they have (*at least*) a dependent.

- js = 1/max(10000, 10) = 0.01% *statistical probab*ility of employee having dependent(s).
- $jc_{EDT} = js * r_{ED} * r_T$ = **0.625 matching tuples** or ceil(0.625/2) = **1 block**

- Clustering Index (E_SSN): $b_{ED} + r_{ED} \cdot (x_{E\_SSN} + \text{ceil}(s_{E\_SSN}/f)) + (jc_{EDT} / f_{RS})$?
- **Note 1:** b_{ED} is already in the memory, thus, *excluded* from the *read* cost.
- **Note 2:** (jc_{EDT} / f_{RS}) result block is only 1, thus, *excluded* from the *write* cost.
- **Note 3:** Cost is *only* for asking the Clustering Index = **375 block accesses**

Total Cost Plan 2: 701 + 375 = 1076 block accesses ☹ > 378 block accesses ☺

HOLISTIC OPTIMIZATION (1/5)

```
SELECT * FROM EMPLOYEE, DEPARTMENT
WHERE    Salary = 1000 AND SSN = MGR_SSN
```

- EMPLOYEE r_E = 10,000 records, b_E = 2,000 blocks.
- Clustering Index (Salary) x_{Salary} = 3 levels, NDV(Salary) = 500.
- B+ Tree (SSN) x_{SSN} = 4 levels

- DEPARTMENT r_D = 125 records, b_D = 13 blocks.
- Primary Index (MGR_SSN) x_{MGR_SSN} = 1 level.

- Blocking factor: f = 10; Result block: f_{RS} = 2; Memory: n_B = 100 blocks

Task: Propose *different* plans and select the *best*.

HOLISTIC OPTIMIZATION (2/5)

```
SELECT * FROM EMPLOYEE, DEPARTMENT
WHERE    Salary = 1000 AND SSN = MGR_SSN
```

Plan 1: first *select* and then *join*; might *reduce* number of matched tuples

Plan 2: first *join* and then *select*; might generate *big* intermediate results

Plan 1.1: Clustering Index for *selection*; Plan 1.2: Primary Index for *joining*

Plan 1.1: Find employees with Salary = 1000

- sl(Salary) = 0.002; cardinality $s_{\text{Salary}} = r \cdot sl(\text{Salary})$ = 20 employees/salary.
- Cluster has ceil(s_{Salary} /f) = **2 blocks with Salary = 1000.**
- Clustering Index (Salary): x_{Salary} + ceil(s_{Salary}/f) = **5 block accesses.**

- **Memory: 2 blocks of employees with Salary 1K in-memory**
- **Plan 1.1 Cost: 5 block accesses**

Holistic Optimization (3/5)

```
SELECT * FROM EMPLOYEE, DEPARTMENT
WHERE    Salary = 1000 AND SSN = MGR_SSN
```

Plan 1.2: Join r_{FE} = 20 employees (b_{FE} = 2 blocks) with departments.

- Index-based Nested-loop Join with the *filtered* employees in outer-loop
- Take a *filtered* employee at a time, and ask if they are a manager.

- js = 1/max(10000, 125) = 0.01% *statistical probability* of employee being manager.
- $jc_F = js * r_{FE} * r_D$ = **0.25 matching tuples** or ceil(0.25/2) = **1 block**

- Primary Index (MGR_SSN): $b_{FE} + r_{FE} \cdot (x_{MGR\_SSN}+1) + (jc_F / f_{RS})$?
- **Note 1:** b_{FE} is already in the memory, thus, *excluded* from the *read* cost.
- **Note 2:** (jc_F / f_{RS}) result block is only 1, thus, *excluded* from the *write* cost.
- **Note 3:** Cost is *only* for asking the Primary Index = **40 block accesses**

Total Cost Plan 1: 5 + 40 = 45 block accesses

Holistic Optimization (4/5)

```
SELECT * FROM EMPLOYEE, DEPARTMENT
WHERE    Salary = 1000 AND SSN = MGR_SSN
```

Plan 2: first *join* and then *select*; might generate *huge* intermediate results

Plan 2.1:B+ Tree (SSN) for *joining*; *P*lan 2.2: *filtering* in memory (I hope, so!).

Plan 2.1: Find the blocks with the managers

- For *each* department, get *its* manager's info.
- *js* = 1/max(10000, 125) = 0.01% statistical probability of employee being manager.

- B+ Tree (SSN): $b_D + r_D \cdot (x_{SSN} + 1) + (js \cdot r_E \cdot r_D / f_{RS})$ = **701 block accesses**

- **Memory: 125 tuples (managers) or ceil(125/f_{RS}) = 63 blocks in-memory**
- **Plan 2.1 Cost: 701 block accesses**

HOLISTIC OPTIMIZATION (5/5)

```
SELECT * FROM EMPLOYEE, DEPARTMENT
WHERE    Salary = 1000 AND SSN = MGR_SSN
```

Plan 2.2: Scan 125 managers and *keep* those with Salary = 1000

Matching tuples: *sl*(Salary)*125 = 0.25 tuples; **the same as in Plan 1.1!!**

Total Cost Plan 2: 701 block accesses

Conclusion:

- **Plan 1: 45 block accesses; intermediate result size: 2 blocks**
  - Computation and storage *efficient*
- **Plan 2: 701 block accesses; intermediate results size: 63 blocks**
  - Computation and storage *inefficient*

Lessons: first *select* and then *join* (reduce the tuple space *before* joining!)

We obtain the *same* heuristic rule in Heuristic Optimization…

PROOF OF THEOREM 1 (OPTIONAL)

Proof: Attribute A has selectivity 1/NDV(A,**R**), thus, each tuple from **S** *will* match, on average, |**R**|/ NDV(A,R) tuples from **R**.

SSN has selectivity 1/|**R**|. Department tuple joins with |**R**| · 1/|**R**| = 1 manager.

We have |**S**| tuples in **S**, hence, the total join result is: |**S**|·(|**R**| / NDV(A,**R**)) = |**S**|·|**R**|/ NDV(A,**R**).

|**S**| departments join with |**S**| (|**R**| · 1/|**R**|) = |**S**| managers.

Similarly, |**S**|·|**R**|/ NDV(B, S).

Obtain the *smaller* of the above two quantities for estimating the join result size, i.e., min(1/NDV(A,**R**), 1/NDV(A,**R**)) or 1/max(NDV(A,**R**), NDV(B,**S**)). Hence:

$$jc = (|\mathbf{R}|\cdot|\mathbf{S}|) / \max(\mathrm{NDV}(A,\mathbf{R}), \mathrm{NDV}(B,\mathbf{S}))$$

Correctness: If attribute B is *key* then NDV(B,**S**) = |**S**|, thus,

$js = 1 / \max(\mathrm{NDV}(A,\mathbf{R}), \mathrm{NDV}(B,\mathbf{S})) = 1 / \max(\mathrm{NDV}(A,\mathbf{R}), |\mathbf{S}|) = 1/|\mathbf{S}|$

Q.E.D.