



CS 3042 Database Systems

CS 3272 Embedded Database Systems

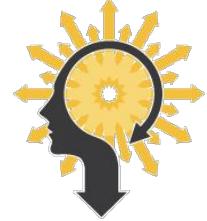
Chapter 1: Introduction

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use

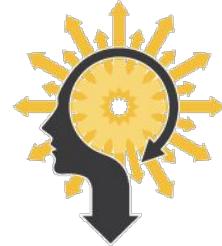
Introduction



- **Learning Outcomes (CS 3042)**
- After completing this module, students should be able to
 - explain the role of database systems in information management.
 - describe the basic fundamental concepts of data modelling and relational databases.
 - apply entity-relationship modelling and normalization for simple database requirements.
 - demonstrate the use of a query language to create, update and query a simple database.
 - construct simple applications that requires manipulating data in a DBMS.
- Recommended Text: Silberschatz, Korth and Sudarshan *Database System Concepts, 7th Edition*, McGraw-Hill 2 McGraw Hill.

Slides are based on the slides provided with this book

Introduction



Learning Outcomes (CS 3272)



- After completing this module, students should be able to
- Explain the role of embedded databases in information management
 - Describe the use of Embedded Databases for mobile, context-aware very small databases, resident on smart cards, PADs, cellular phone SIM cards or Wireless Sensors.
 - Explain the importance of data consistency, concurrency control, query processing, optimization, recovery management in Embedded Databases.
 - Demonstrate the use of a query language to create, update and query a simple database.
 - Construct simple applications that require manipulating data in an Embedded Database.

Grading Scheme

- **Method of Assessment:**
 - 2 hour closed book examination - 60%
 - Continuous assessments - 50%
- **Breakdown of the Continuous Assessments**
- **Marks:**
 - 2-hour closed book end of semester examination - 60%
 - Mini project (Group) - 20%
 - Mid Term Quiz (Moodle) - 10 %
 - In-class/lab pop quizzes/labs - 10 %



Database Management System (DBMS)

- DBMS contains information about a particular enterprise
 - Collection of interrelated data
 - Set of programs to access the data
 - An environment that is both *convenient* and *efficient* to use
- Database Applications:
 - Banking: transactions
 - Airlines: reservations, schedules
 - Universities: registration, grades
 - Sales: customers, products, purchases
 - Online retailers: order tracking, customized recommendations
 - Manufacturing: production, inventory, orders, supply chain
 - Human resources: employee records, salaries, tax deductions
- Databases can be very large.
- Databases touch all aspects of our lives



University Database Example

- Application program examples
 - Add new students, instructors, and courses
 - Register students for courses, and generate class rosters
 - Assign grades to students, compute grade point averages (GPA) and generate transcripts
- In the early days, database applications were built directly on top of file systems

Drawbacks of using file systems to store data

- Data redundancy and inconsistency
 - Multiple file formats, duplication of information in different files
- Difficulty in accessing data
 - Need to write a new program to carry out each new task
- Data isolation – multiple files and formats
- Integrity problems
 - Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
 - Hard to add new constraints or change existing ones

Drawbacks of using file systems to store data (Cont.)

- ◆ Atomicity of updates
 - Failures may leave database in an inconsistent state with partial updates carried out
 - Example: Transfer of funds from one account to another should either complete or not happen at all
- ◆ Concurrent access by multiple users
 - Concurrent access needed for performance
 - Uncontrolled concurrent accesses can lead to inconsistencies
 - Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- ◆ Security problems
 - Hard to provide user access to some, but not all, data

Database systems offer solutions to all the above problems

Levels of Abstraction

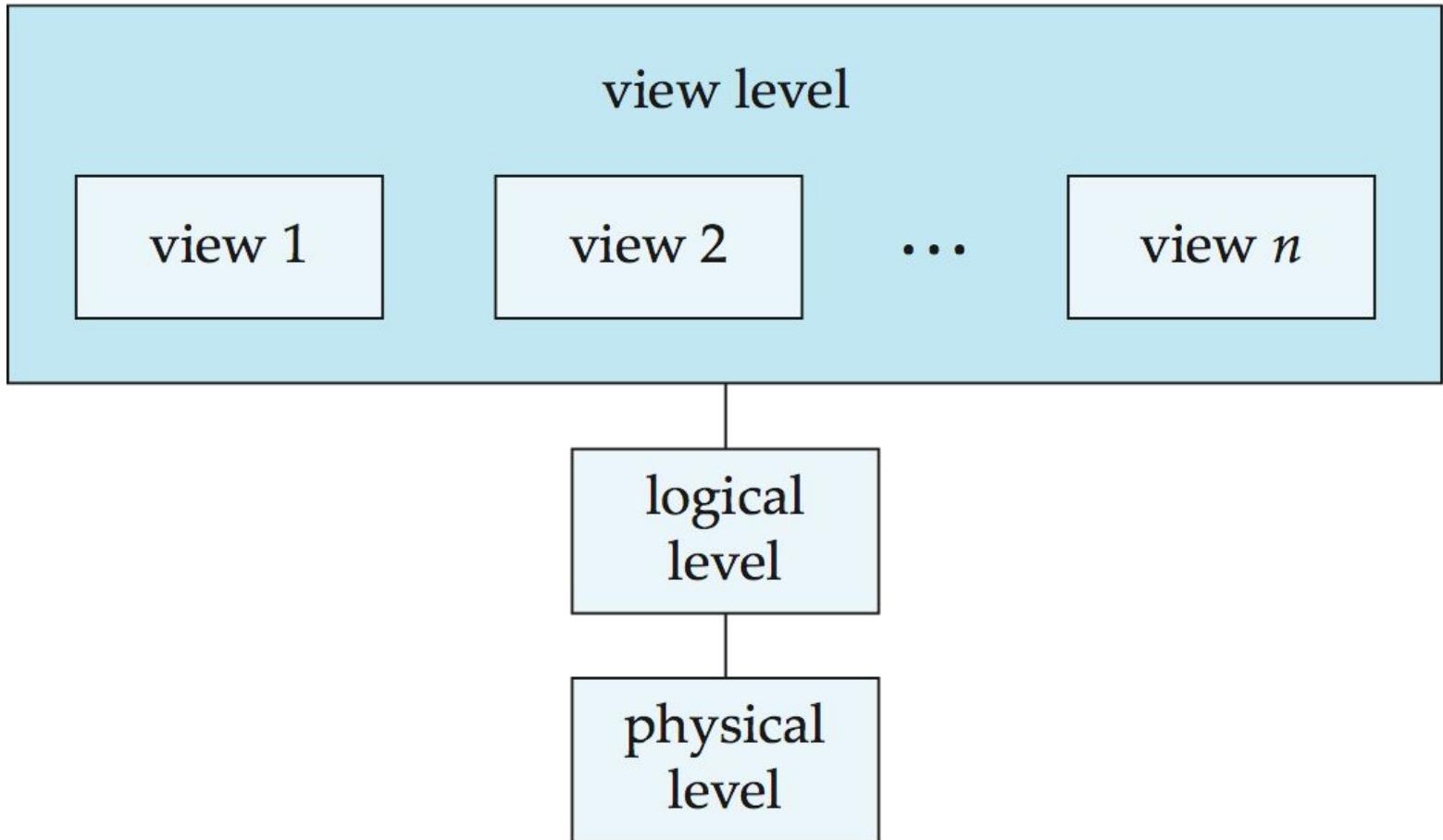
- **Physical level:** describes how a record (e.g., customer) is stored.
- **Logical level:** describes data stored in database, and the relationships among the data.

```
type instructor = record  
    ID : string;  
    name : string;  
    dept_name : string;  
    salary : integer;  
end;
```

- **View level:** application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

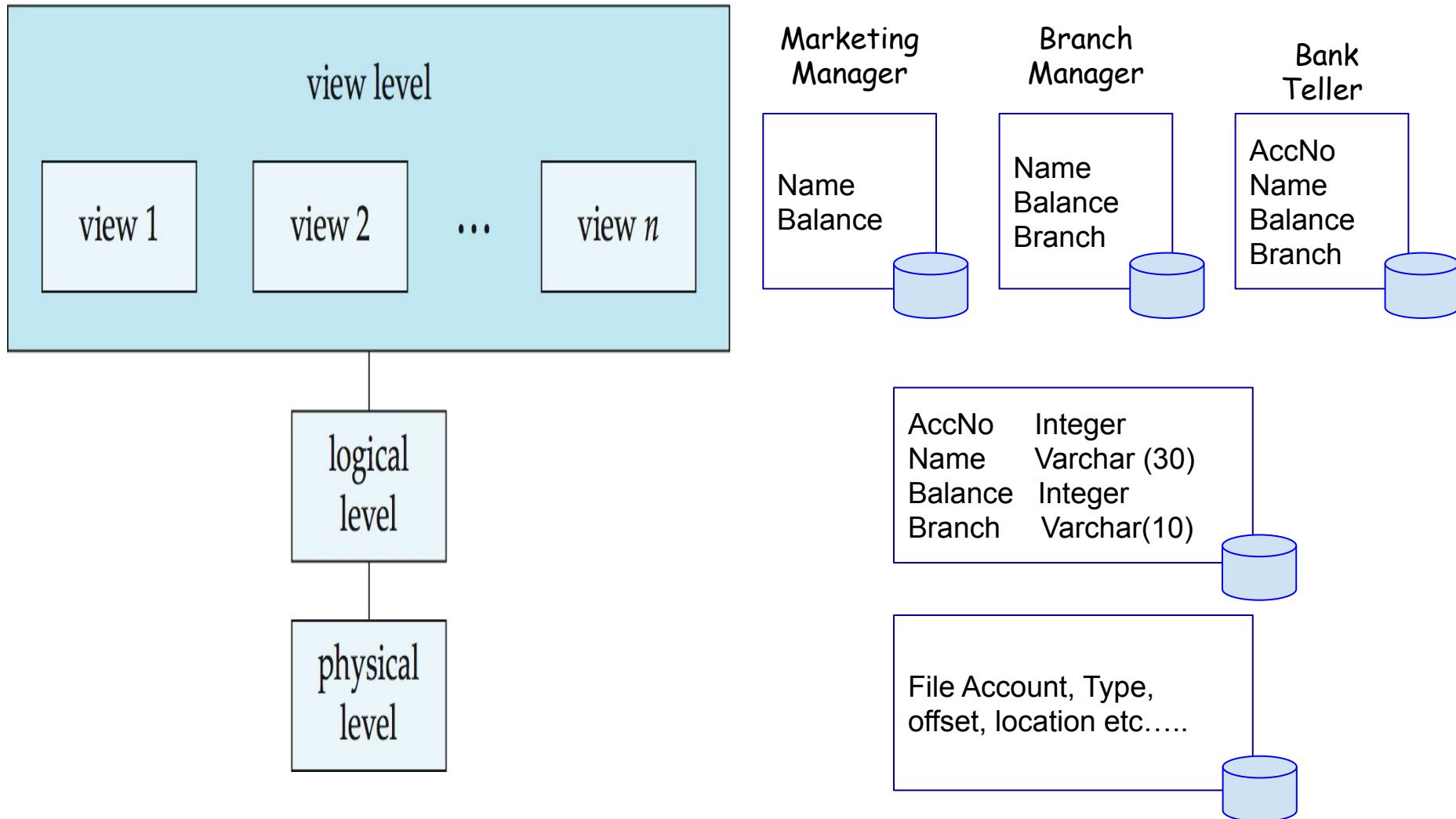
View of Data

An architecture for a database system



View of Data

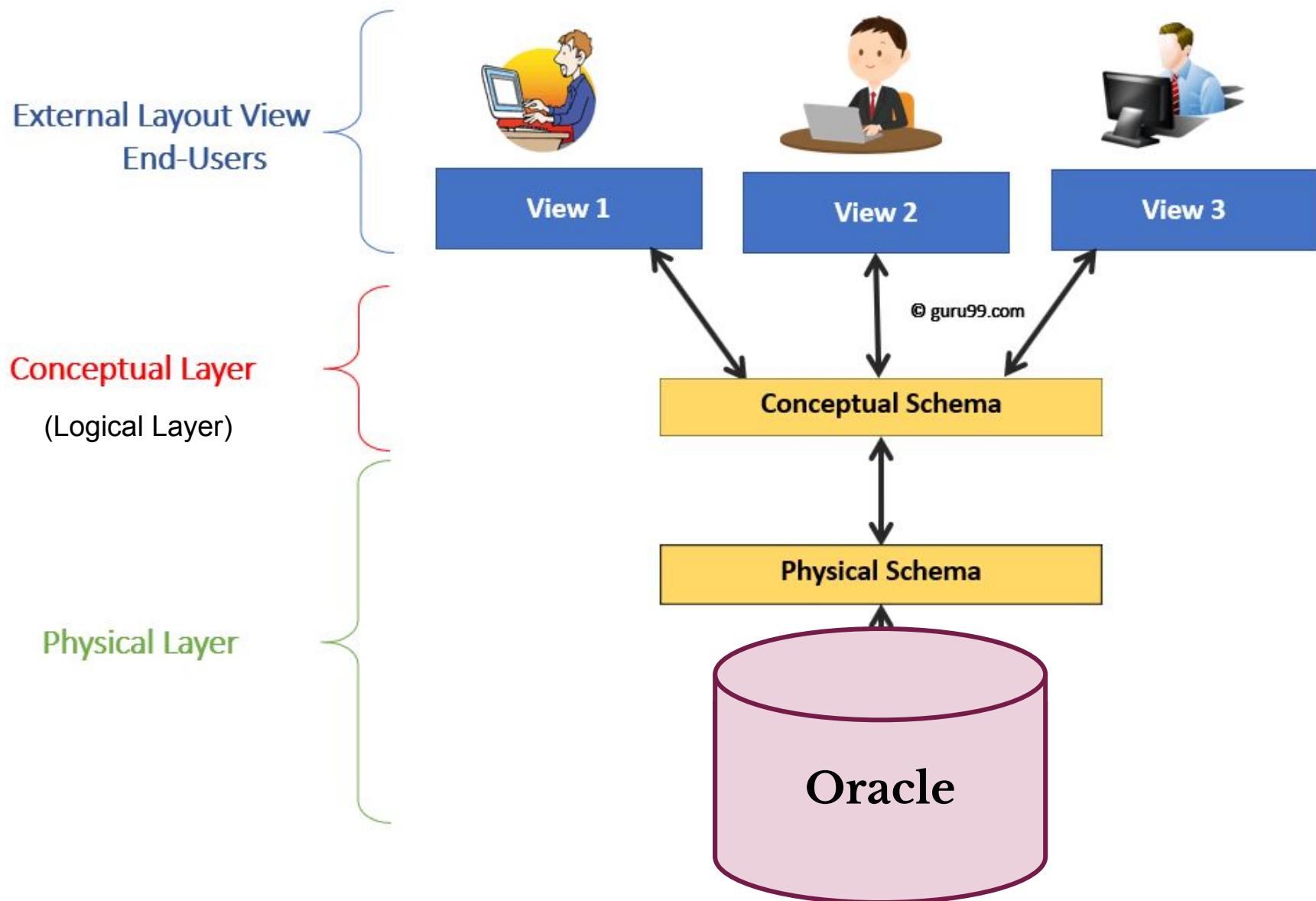
An architecture for a database system



Instances and Schemas

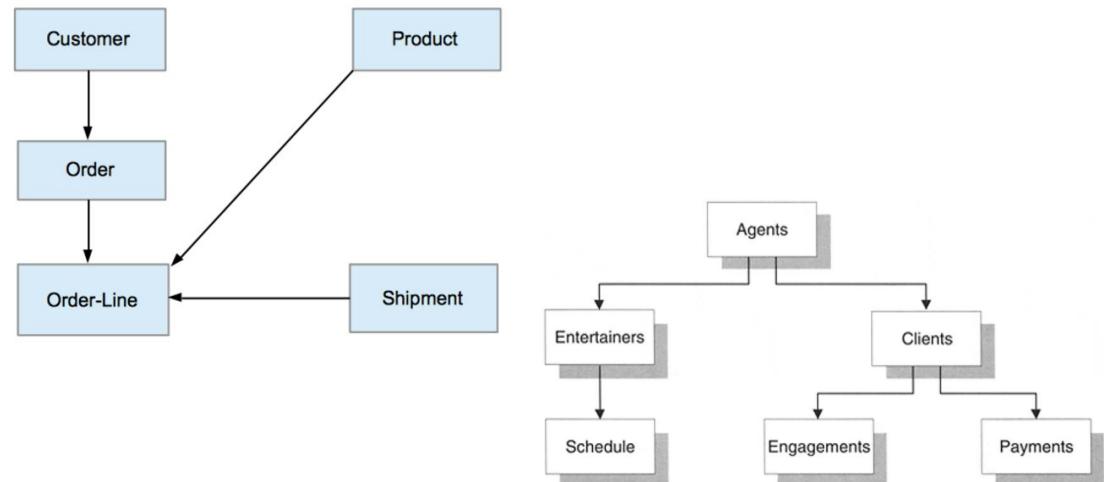
- Similar to types and variables in programming languages
- **Schema** – the logical structure of the database
 - Example: The database consists of information about a set of customers and accounts and the relationship between them
 - Analogous to type information of a variable in a program
 - **Physical schema:** database design at the physical level
 - **Logical schema:** database design at the logical level
- **Instance** – the actual content of the database at a particular point in time
 - Analogous to the value of a variable
- **Physical Data Independence** – the ability to modify the physical schema without changing the logical schema
 - Applications depend on the logical schema
 - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

Physical Data Independence



Data Models

- A collection of tools for describing
 - Data
 - Data relationships
 - Data semantics
 - Data constraints
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semistructured data model (XML)
- Other older models:
 - Network model
 - Hierarchical model



Relational Model



Ted Codd

Turing Award 1981

- Relational model (Chapter 2)
- Example of tabular data in the relational model

Columns

Rows

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

(a) The *instructor* table

A Sample Relational Database

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

(a) The *instructor* table

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table

Data Manipulation Language (DML)

- Language for accessing and manipulating the data organized by the appropriate data model
 - DML also known as query language
- Two classes of languages
 - **Procedural** – user specifies what data is required and how to get those data
 - **Declarative (nonprocedural)** – user specifies what data is required without specifying how to get those data
- SQL is the most widely used query language

Data Definition Language (DDL)

- Specification notation for defining the database schema

Example: **create table** *instructor* (
 ID **char**(5),
 name **varchar**(20),
 dept_name **varchar**(20),
 salary **numeric**(8,2))

- DDL compiler generates a set of table templates stored in a **data dictionary**
- Data dictionary contains metadata (i.e., data about data)
 - Database schema
 - Integrity constraints
 - Primary key (*ID* uniquely identifies instructors)
 - Referential integrity (**references** constraint in SQL)
 - e.g. *dept_name* value in any *instructor* tuple must appear in *department* relation
 - Authorization

SQL

- **SQL:** widely used non-procedural language
 - Example: Find the name of the instructor with ID 22222
`select name
from instructor
where instructor.ID = '22222'`
 - Example: Find the ID and building of instructors in the Physics dept.
`select instructor.ID, department.building
from instructor, department
where instructor.dept_name = department.dept_name and
department.dept_name = 'Physics'`
- Application programs generally access databases through one of
 - Language extensions to allow embedded SQL
 - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database
- Chapters 3, 4 and 5

Database Design

The process of designing the general structure of the database:

- Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas.
 - Business decision – What attributes should we record in the database?
 - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database

Database Design?

- Is there any problem with this design?

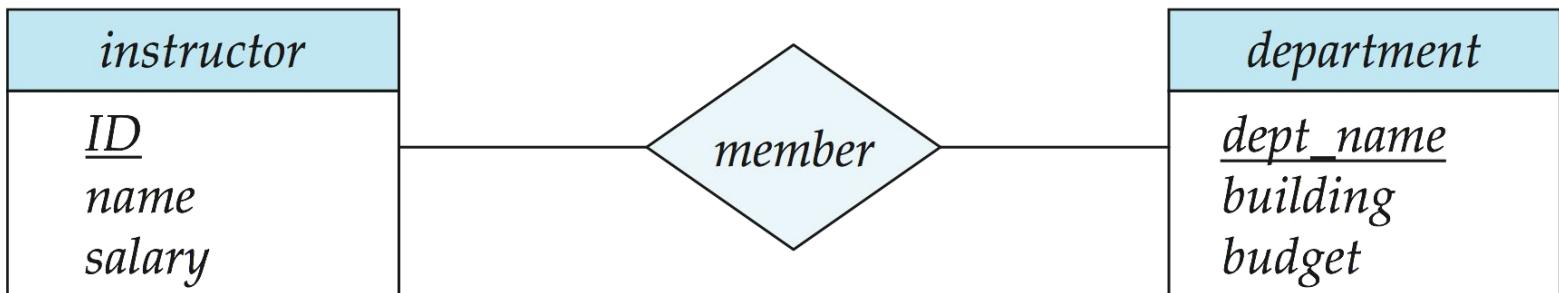
<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Design Approaches

- Normalization Theory (Chapter 7)
 - ◆ Formalize what designs are bad, and test for them
- Entity Relationship Model (Chapter 6)
 - ◆ Models an enterprise as a collection of *entities* and *relationships*
 - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
 - Described by a set of *attributes*
 - Relationship: an association among several entities
 - ◆ Represented diagrammatically by an *entity-relationship diagram*:

The Entity-Relationship Model

- Models an enterprise as a collection of *entities* and *relationships*
 - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
 - Described by a set of *attributes*
 - Relationship: an association among several entities
- Represented diagrammatically by an *entity-relationship diagram*:



What happened to dept_name of instructor and student?

Object-Relational Data Models

- Relational model: flat, “atomic” values
- Object Relational Data Models
 - Extend the relational data model by including object orientation and constructs to deal with added data types.
 - Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
 - Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
 - Provide upward compatibility with existing relational languages.

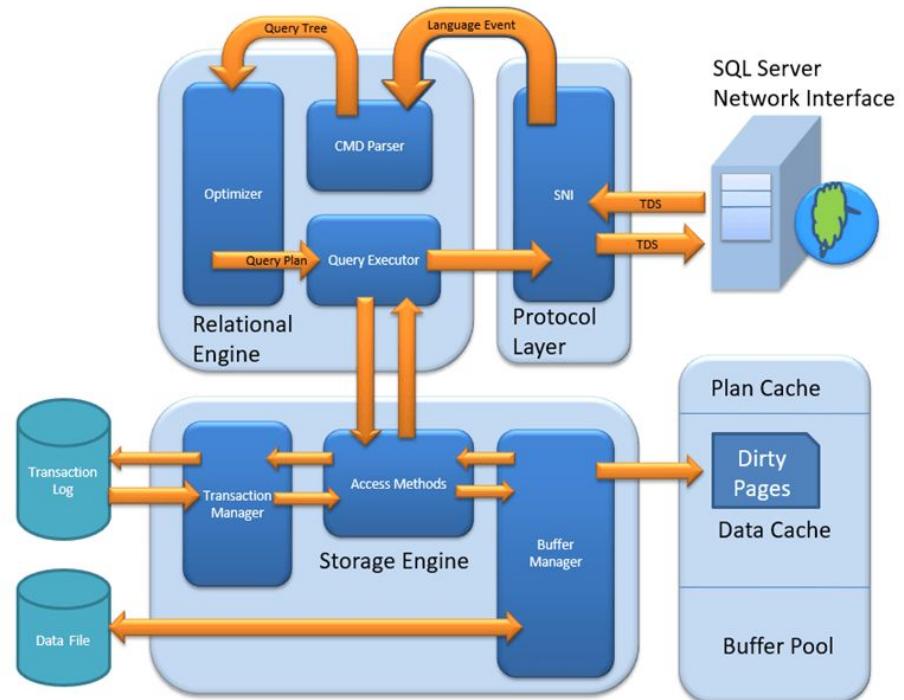
XML: Extensible Markup Language

- Defined by the WWW Consortium (W3C)
- Originally intended as a document markup language not a database language
- The ability to specify new tags, and to create nested tag structures made XML a great way to exchange **data**, not just documents
- XML has become the basis for all new generation data interchange formats.
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

Database Engine



1. Storage manager
2. Query processing
3. Transaction manager

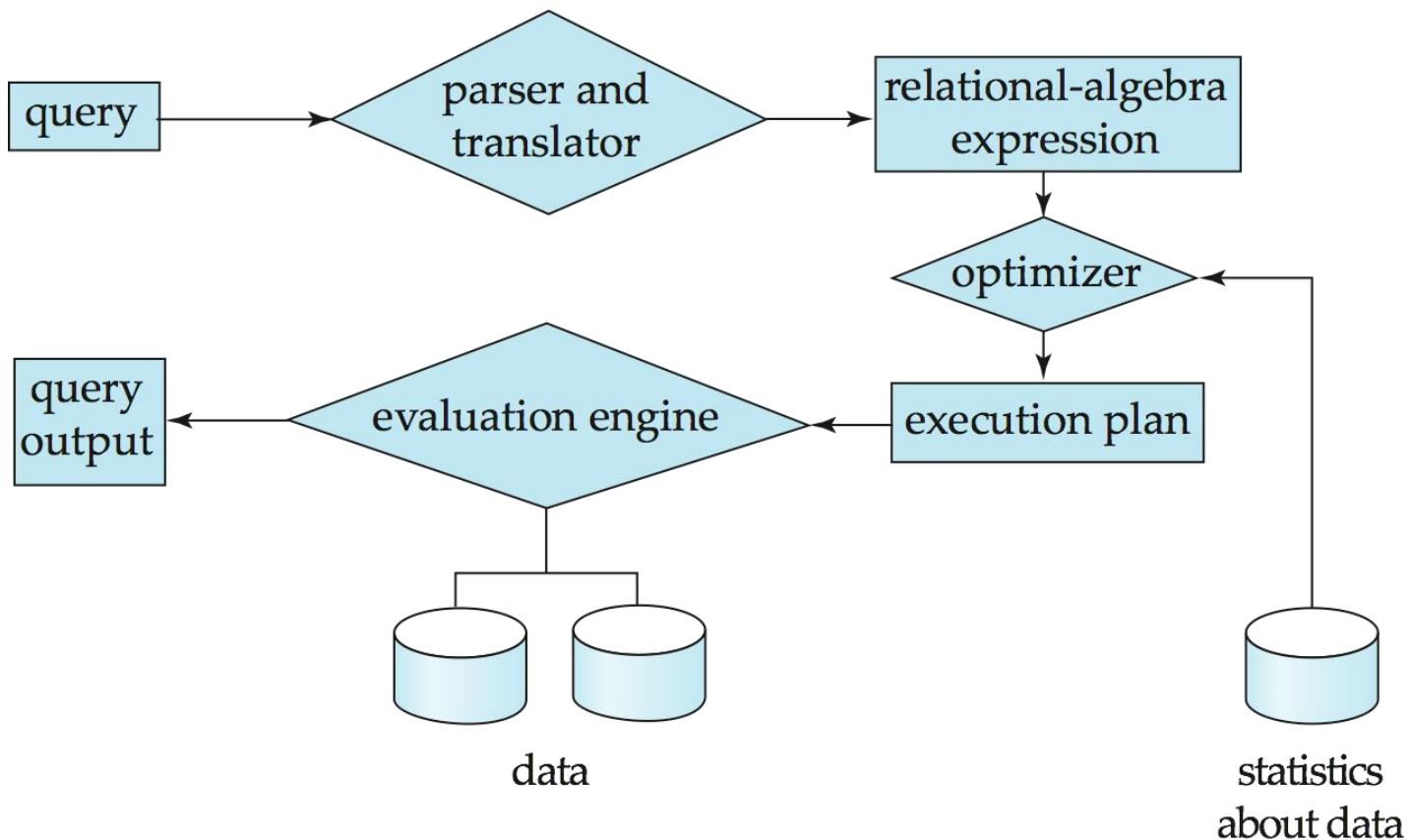


Storage Management

- **Storage manager** is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
 - Interaction with the file manager
 - Efficient storing, retrieving and updating of data
- Issues:
 - Storage access
 - File organization
 - Indexing and hashing

Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



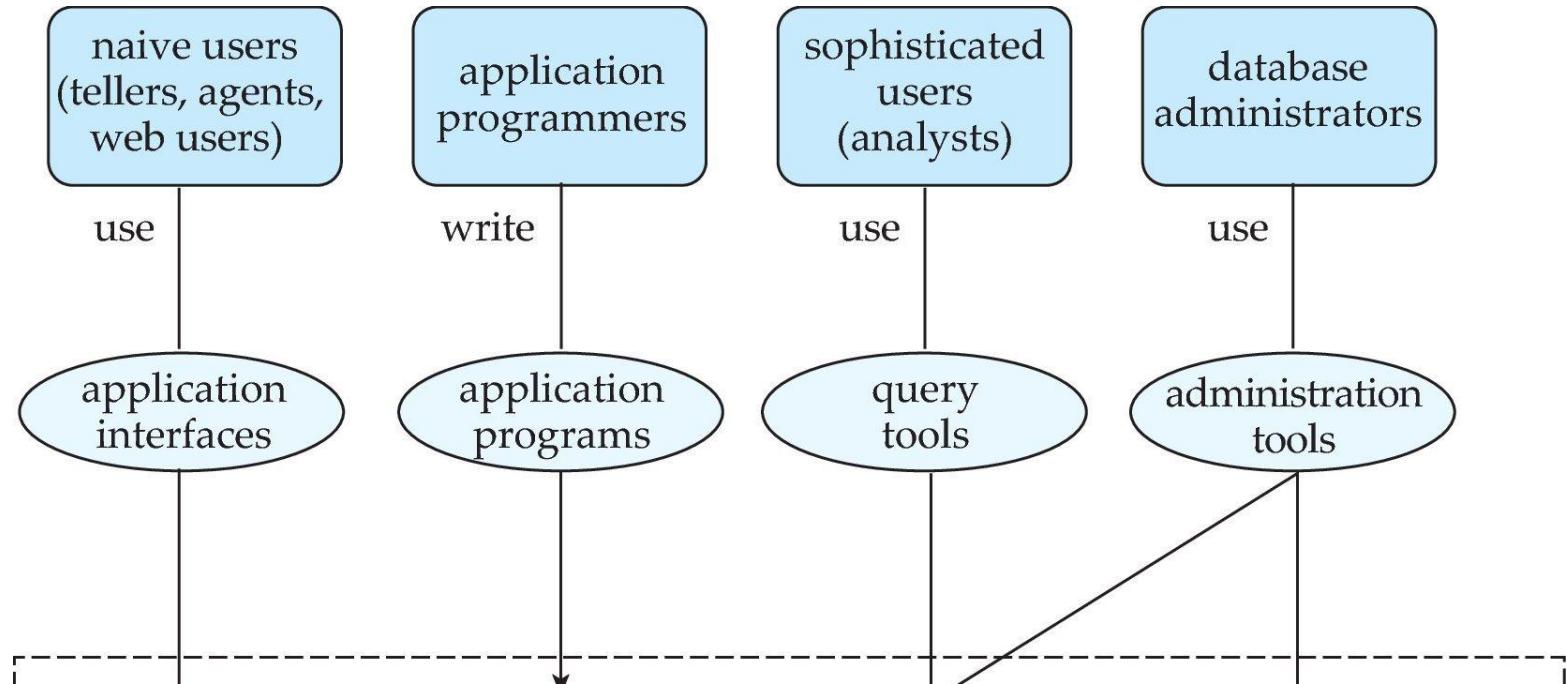
Query Processing (Cont.)

- Alternative ways of evaluating a given query
 - Equivalent expressions
 - Different algorithms for each operation
- Cost difference between a good and a bad way of evaluating a query can be enormous
- Need to estimate the cost of operations
 - Depends critically on statistical information about relations which the database must maintain
 - Need to estimate statistics for intermediate results to compute cost of complex expressions

Transaction Management

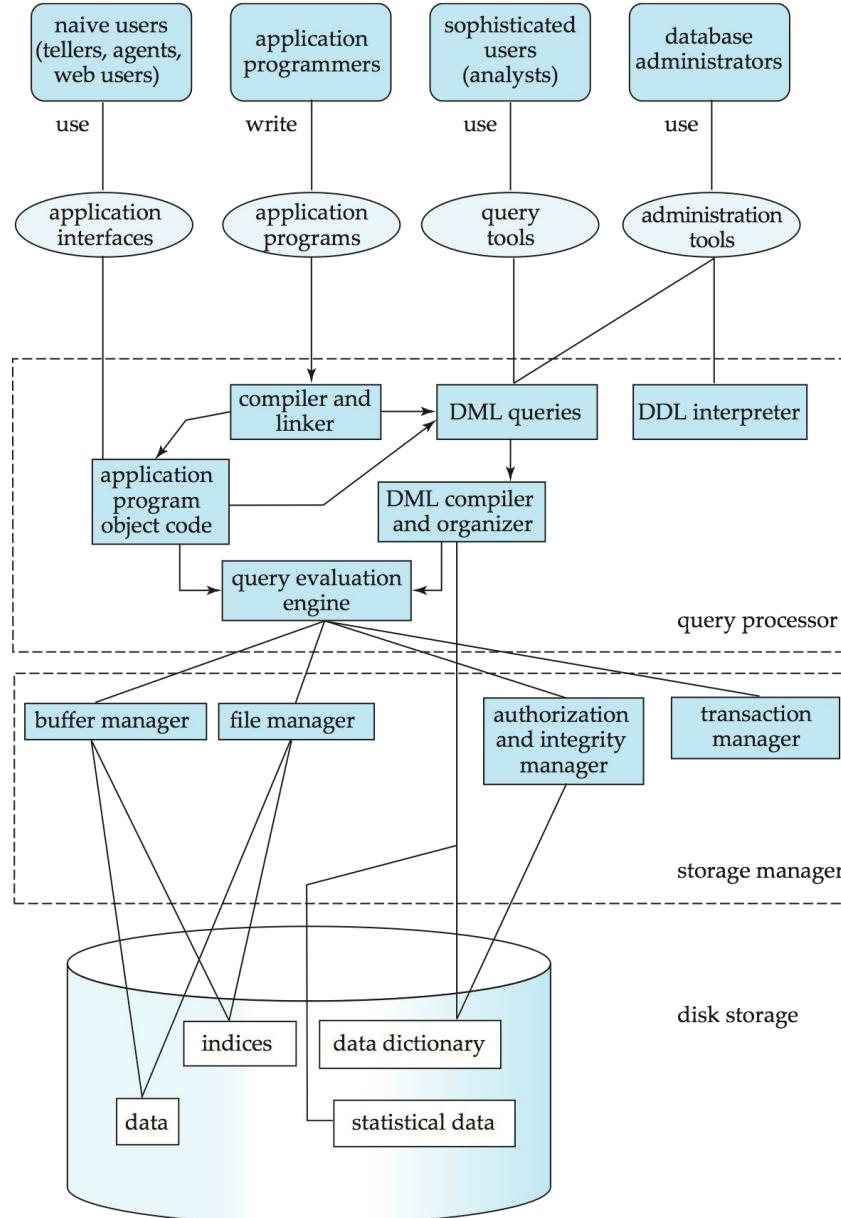
- What if the system fails?
- What if more than one user is concurrently updating the same data?
- A **transaction** is a collection of operations that performs a single logical function in a database application
- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Database Users and Administrators



Database

Database System Internals



Database Architecture

The architecture of a database systems is greatly influenced by

the underlying computer system on which the database is running:

- Centralized
- Client-server
- Parallel (multi-processor)
- Distributed

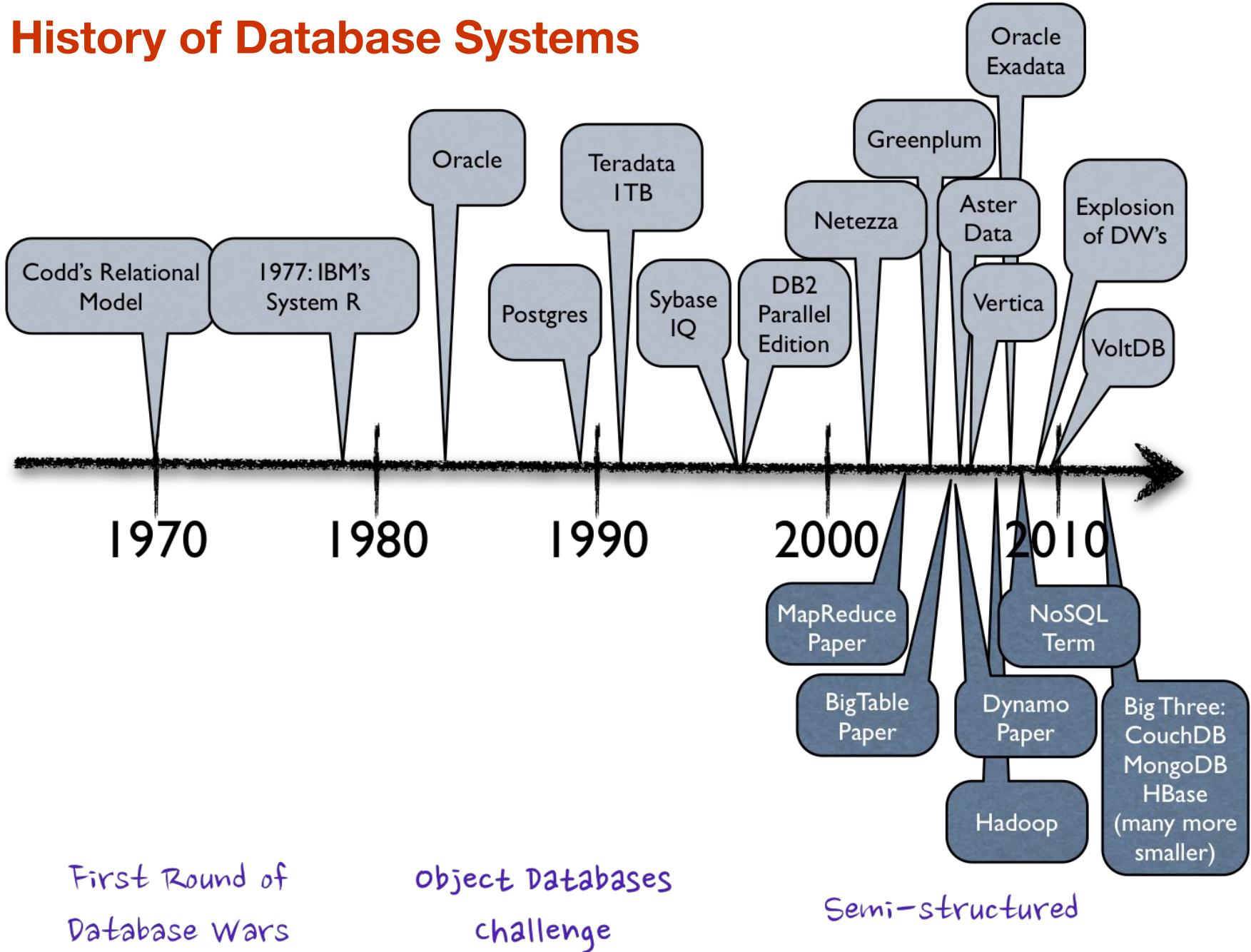
History of Database Systems

- ★ 1950s and early 1960s:
 - Data processing using magnetic tapes for storage
 - Tapes provided only sequential access
 - Punched cards for input
- ★ Late 1960s and 1970s:
 - Hard disks allowed direct access to data
 - Network and hierarchical data models in widespread use
 - Ted Codd defines the relational data model
 - Would win the ACM Turing Award for this work
 - IBM Research begins System R prototype
 - UC Berkeley begins Ingres prototype
 - High-performance (for the era) transaction processing

History (cont.)

- ★ 1980s:
 - Research relational prototypes evolve into commercial systems
 - SQL becomes industrial standard
 - Parallel and distributed database systems
 - Object-oriented database systems
- ★ 1990s:
 - Large decision support and data-mining applications
 - Large multi-terabyte data warehouses
 - Emergence of Web commerce
- ★ Early 2000s:
 - XML and XQuery standards
 - Automated database administration
- ★ Later 2000s:
 - Giant data storage systems
 - Google BigTable, Yahoo PNuts, Amazon, ..

History of Database Systems



End of Chapter 1

Chapter 2: Intro to Relational Model

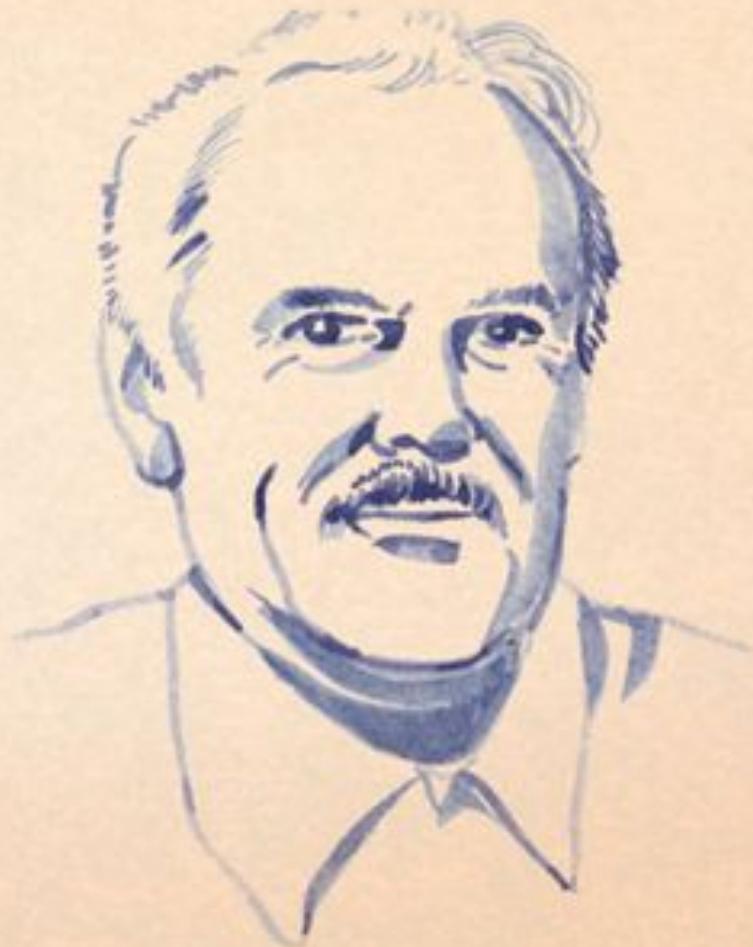
Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use

Father of the Relational Database: Edgar F. Codd

A British computer scientist, Codd made important contributions to the theory of relational databases. While working for IBM, he created the relational model for database management.



Exercise 1.1

Briefly explain 5 (five) reasons as to why you would use a DBMS for a university student registration system rather than a simple flat file.

Example of a Relation

The diagram illustrates a relation table with four columns: *ID*, *name*, *dept_name*, and *salary*. The table contains 12 tuples (rows). Annotations with arrows point to specific parts of the table:

- Three arrows point from the text "attributes (or columns)" to the header cells of the first three columns.
- Two arrows point from the text "tuples (or rows)" to the second and third rows of the table.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

Attribute Types

- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
- The special value ***null*** is a member of every domain
- The null value causes complications in the definition of many operations

Relation Schema and Instance

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

Example:

instructor = (*ID*, *name*, *dept_name*, *salary*)

- Formally, given sets D_1, D_2, \dots, D_n a **relation** r is a subset of $D_1 \times D_2 \times \dots \times D_n$
Thus, a relation is a set of n -tuples (a_1, a_2, \dots, a_n) where each $a_i \in D_i$

- The current values (**relation instance**) of a relation are specified by a table
- An element t of r is a *tuple*, represented by a *row* in a table

Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- Example: *instructor* relation with unordered tuples

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

Database

- A database consists of multiple relations
- Information about an enterprise is broken up into parts

instructor

student

advisor

- Bad design:

univ (instructor -ID, name, dept_name, salary, student_Id, ..)

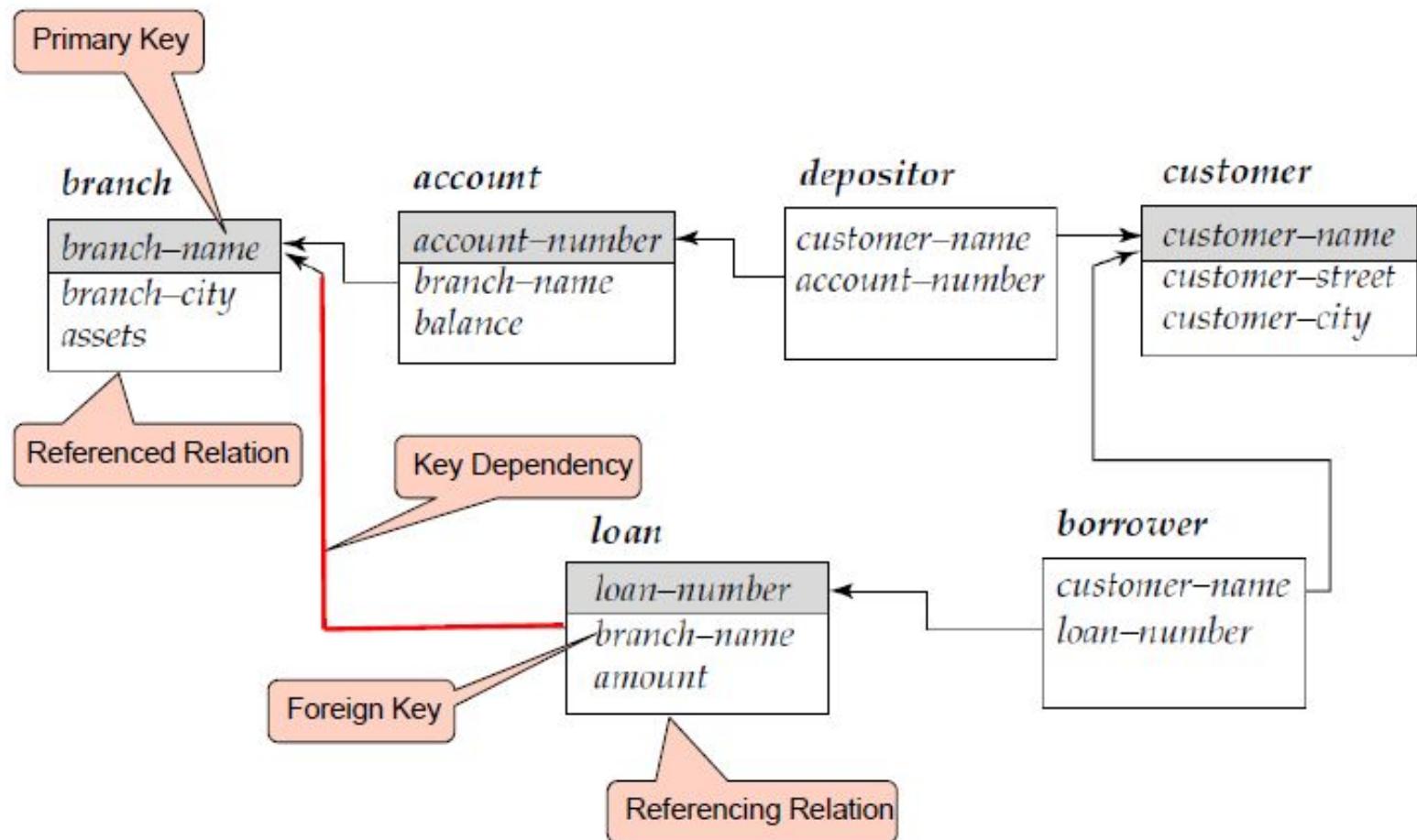
results in

- repetition of information (e.g., two students have the same instructor)
 - the need for null values (e.g., represent an student with no advisor)
- Normalization theory (Chapter 7) deals with how to design “good” relational schemas

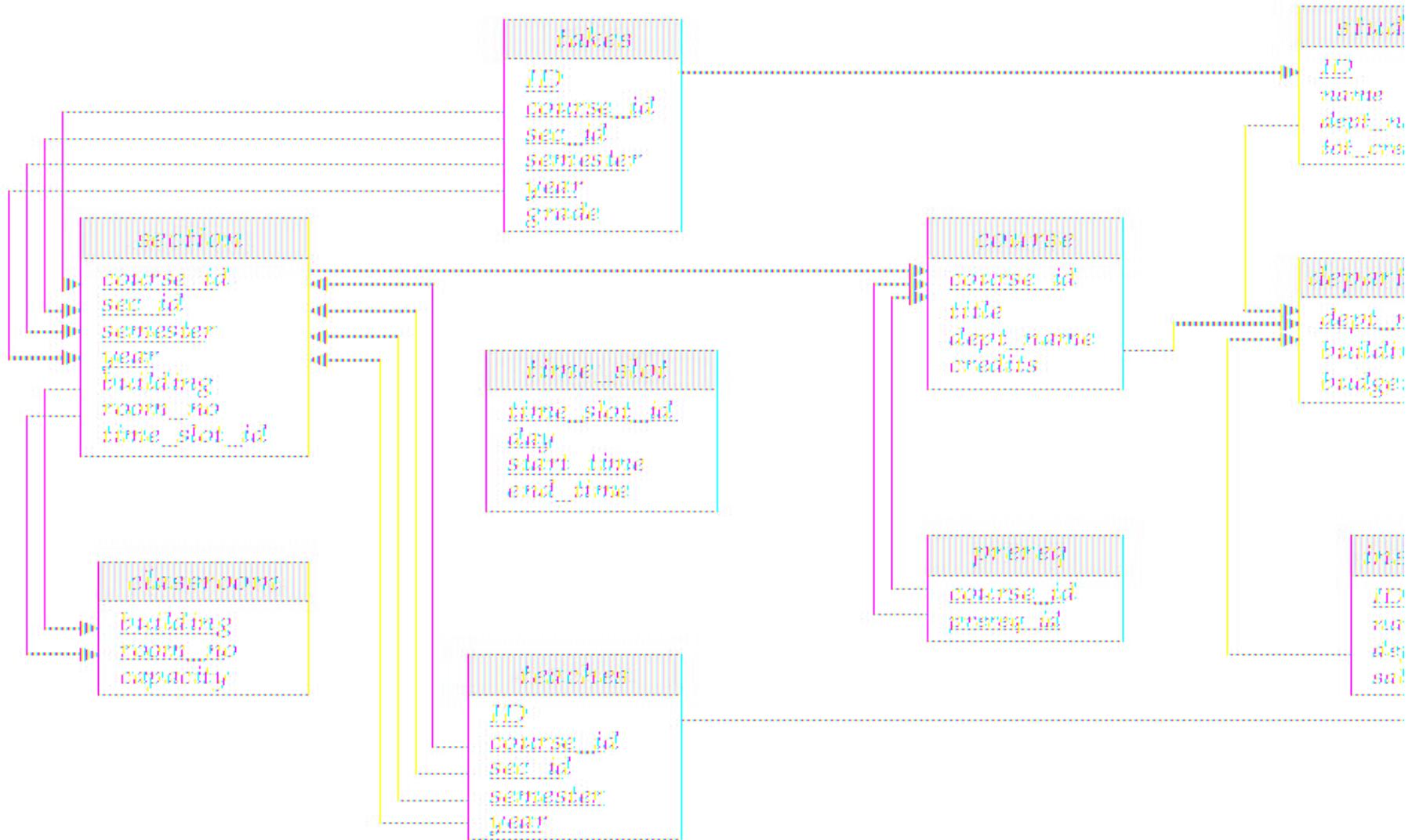
Keys

- Let $K \subseteq R$
- K is a **superkey** of R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$
 - Example: $\{ID\}$ and $\{ID, name\}$ are both superkeys of *instructor*.
- Superkey K is a **candidate key** if K is minimal
Example: $\{ID\}$ is a candidate key for *Instructor*
- One of the candidate keys is selected to be the **primary key**.
 - which one?
- **Foreign key** constraint: Value in one relation must appear in another
 - **Referencing** relation
 - **Referenced** relation

Example: Bank DB Schema



Schema Diagram for University Database



Exercise 1.2 What are appropriate Primary Keys ?

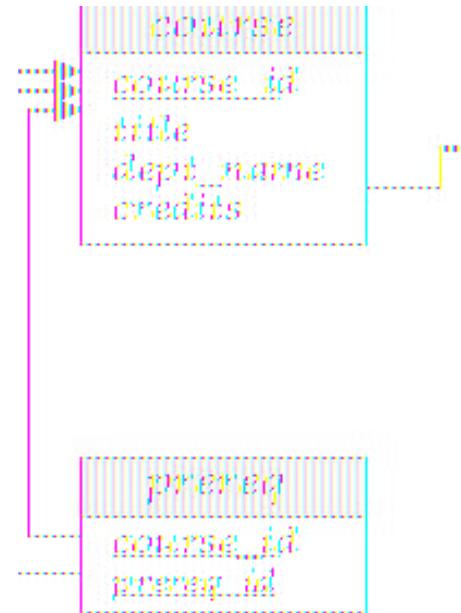
employee (person_name, street, city)

works (person_name, company_name, salary)

company (company_name, city)

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

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000



Exercise 1.3 Give an example insert that would violate the foreign key constraint ?

Exercise 1.4 Give an example Delete that would violate the foreign key constraint ?

Exercise 1.5 From the Database Instance of Instructor shown no two instructors have the same name. From this observation can we conclude that name can be used as a superkey (primary key) of instructor ?

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

Relational Query Languages

- Procedural vs.non-procedural, or declarative
- “Pure” languages:
 - Relational algebra
 - Tuple relational calculus
 - Domain relational calculus
- Relational operators
- Relational Algebra operators

Six basic operators

Select: σ

Project: Π

Union: U

set difference: $-$

Cartesian product: \times

Rename:

The operators take one or two relations as inputs and produce a new relation as a result.

Selection of tuples

- Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

Select tuples with A=B and D > 5

$\sigma_{A=B \text{ and } D > 5} (r)$

A	B	C	D
α	α	1	7
β	β	23	10

Selection of Columns (Attributes)

Relation r :

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

Select A and C Projection

$$\Pi_{A, C}(r)$$

A	C
α	1
α	1
β	1
β	2

$$=$$

A	C
α	1
β	1
β	2

Joining two relations – Cartesian Product

Relations r, s :

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

$r \times s$:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Union of two relations

Relations r , s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

$r \cup s$:

A	B
α	1
α	2
β	1
β	3

Set difference of two relations

- Relations r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r - s$:

A	B
α	1
β	1

Set Intersection of two relations

- Relation r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cap s$

A	B
α	2

Joining two relations – Natural Join

- Let r and s be relations on schemas R and S respectively.
Then, the “natural join” of relations R and S is a relation on schema $R \cup S$ obtained as follows:
 - Consider each pair of tuples t_r from r and t_s from s .
 - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
 - t has the same value as t_r on r
 - t has the same value as t_s on s

Natural Join Example

- Relations r, s:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

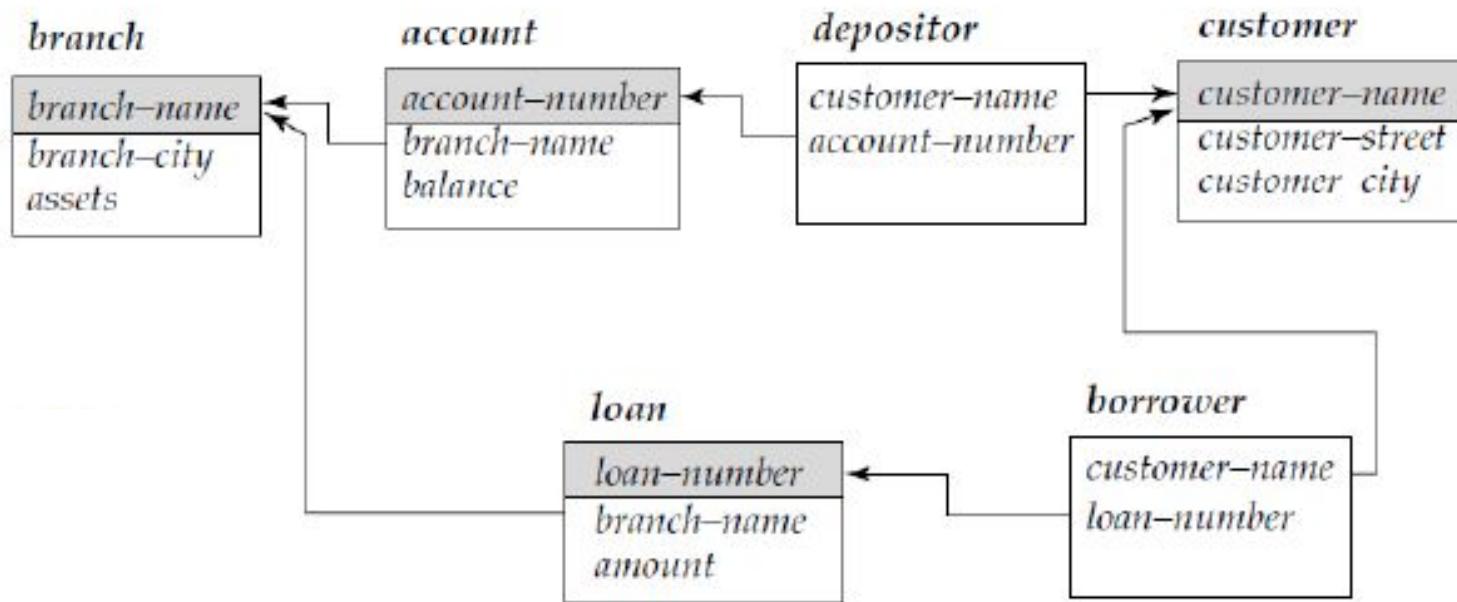
- Natural Join
 - $r \bowtie s$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Figure in-2.1

Symbol (Name)	Example of Use
σ (Selection)	$\sigma_{\text{salary} \geq 85000}(\text{instructor})$ Return rows of the input relation that satisfy the predicate.
Π (Projection)	$\Pi_{ID, \text{salary}}(\text{instructor})$ Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
\bowtie (Natural Join)	$\text{instructor} \bowtie \text{department}$ Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.
\times (Cartesian Product)	$\text{instructor} \times \text{department}$ Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)
\cup (Union)	$\Pi_{name}(\text{instructor}) \cup \Pi_{name}(\text{student})$ Output the union of tuples from the two input relations.

Exercise 1.6



Consider the bank database of Figure ???. Give an expression in the relational algebra for each of the following queries.

- Find the names of all branches located in “Chicago”.
- Find the names of all borrowers who have a loan in branch “Down-town”.

Exercise 1.7

Employee(emp_no, emp_name, emp_city,)

Assignment(proj_no, emp_no, hours,.....)

Project(proj_name, budget, proj_no, proj_start_date, proj_end_date, proj_location,.....)

Express each of the following queries in Relational Algebra:

1. List the name(s) and budget(s) of projects started before 1st May 2008.
2. List the name(s) of projects with a budget value above Rs. 1,000,000.
3. Find the name(s) of employees who are from city “Moratuwa”.
4. Find the name(s) of employees who are from city “Moratuwa” and work on projects located in “Moratuwa”.
5. Find the name(s) of employees who work on projects valued above Rs. 1,000,000.

End of Chapter 2

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use

Figure 2.01

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

Figure 2.02

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure 2.03

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure 2.04

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

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 2.05

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 2.06

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 2.07

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure 2.10

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
83821	Brandt	Comp. Sci.	92000

Figure 2.11

<i>ID</i>	<i>salary</i>
10101	65000
12121	90000
15151	40000
22222	95000
32343	60000
33456	87000
45565	75000
58583	62000
76543	80000
76766	72000
83821	92000
98345	80000

Figure 2.12

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
12121	Wu	90000	Finance	Painter	120000
15151	Mozart	40000	Music	Packard	80000
22222	Einstein	95000	Physics	Watson	70000
32343	El Said	60000	History	Painter	50000
33456	Gold	87000	Physics	Watson	70000
45565	Katz	75000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
76543	Singh	80000	Finance	Painter	120000
76766	Crick	72000	Biology	Watson	90000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000

Figure 2.13

<i>ID</i>	<i>salary</i>
12121	90000
22222	95000
33456	87000
83821	92000

Figure 1.02

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

(a) The *instructor* table

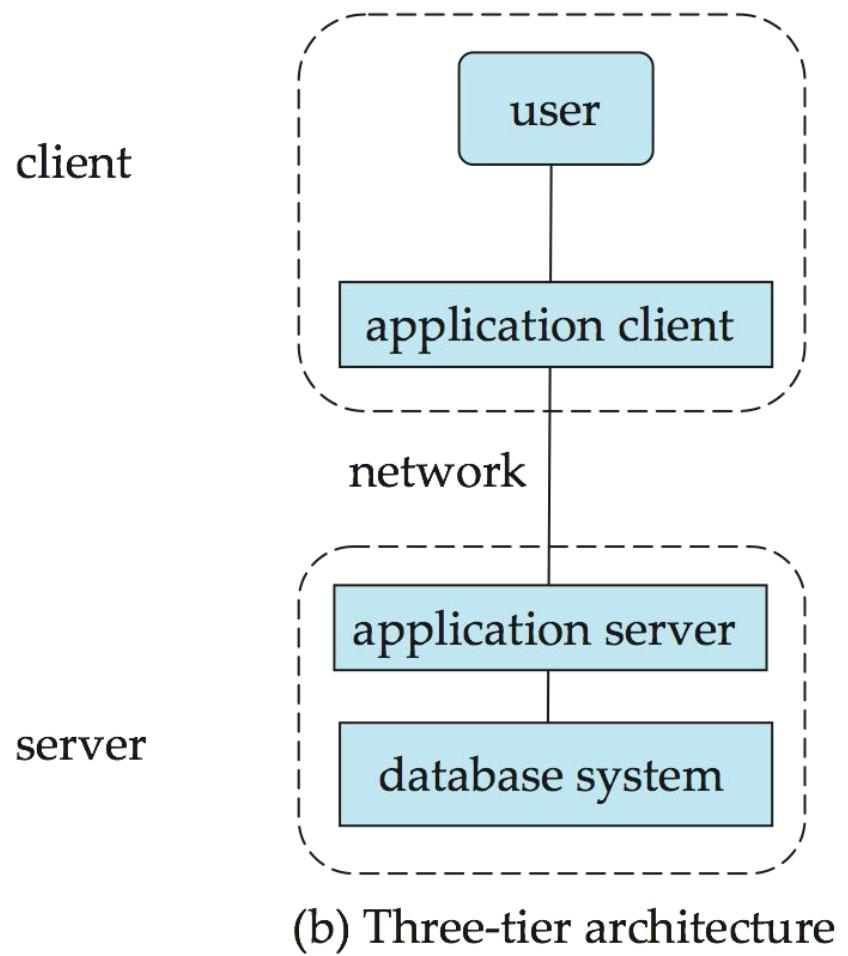
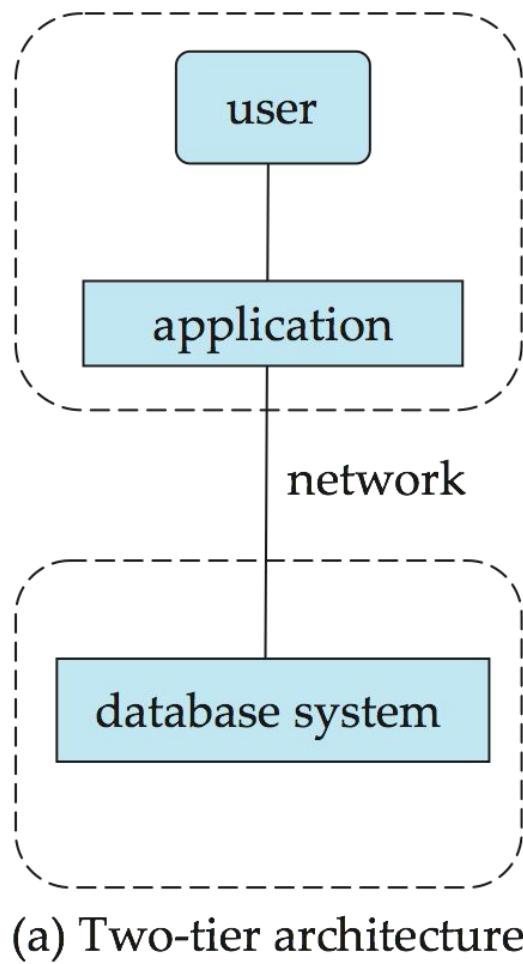
<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table

Figure 1.04

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 1.06



Chapter 6: Database Design Using the E-R Model

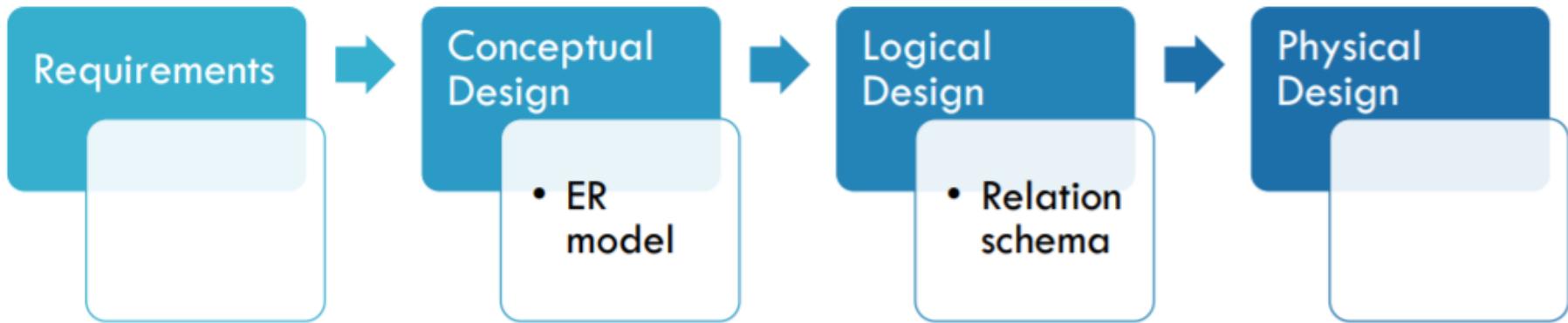
Slides based on: **Database System Concepts, 7th Ed.**

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Outline

- Overview of the Design Process
- The Entity-Relationship Model
- Complex Attributes
- Mapping Cardinalities
- Primary Key
- Removing Redundant Attributes in Entity Sets
- Reducing ER Diagrams to Relational Schemas
- Extended E-R Features
- Entity-Relationship Design Issues
- Alternative Notations for Modeling Data
- Other Aspects of Database Design

Design Phases



Design Objective:

Avoid Redundancies, Inconsistencies and incompleteness

Design Phases

- Initial phase -- characterize fully the data needs of the prospective database users.
- Second phase -- choosing a data model
 - Applying the concepts of the chosen data model
 - Translating these requirements into a conceptual schema of the database.
 - A fully developed conceptual schema indicates the functional requirements of the enterprise.
 - Describe the kinds of operations (or transactions) that will be performed on the data.

Design Phases (Cont.)

- Final Phase -- Moving from an abstract data model to the implementation of the database
 - Logical Design – Deciding on the database schema.
 - Database design requires that we find a “good” collection of relation schemas.
 - Business decision – What attributes should we record in the database?
 - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
 - Physical Design – Deciding on the physical layout of the database

Design Alternatives

- In designing a database schema, we must ensure that we avoid two major pitfalls:
 - **Redundancy:** a bad design may result in repeat information.
 - Redundant representation of information may lead to data inconsistency among the various copies of information
 - **Incompleteness:** a bad design may make certain aspects of the enterprise difficult or impossible to model.
- Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose.

Design Approaches

- Entity Relationship Model (covered in this chapter)
 - Models an enterprise as a collection of *entities* and *relationships*
 - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
 - Described by a set of *attributes*
 - Relationship: an association among several entities
 - Represented diagrammatically by an *entity-relationship diagram*:
- Normalization Theory (Chapter 7)
 - Formalize what designs are bad, and test for them

Outline of the ER Model

ER model -- Database Modeling

- The ER data model was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.
- The ER data model employs three basic concepts:
 - entity sets,
 - relationship sets,
 - attributes.
- The ER model also has an associated diagrammatic representation, the **ER diagram**, which can express the overall logical structure of a database graphically.

Entity Sets

- An **entity** is an object that exists and is distinguishable from other objects.
 - Example: specific person, company, event, plant
- An **entity set** is a set of entities of the same type that share the same properties.
 - Example: set of all persons, companies, trees, holidays
- An entity is represented by a set of **attributes**; i.e., descriptive properties possessed by all members of an entity set.
 - Example:
instructor= (ID, name, salary)
course=(course_id, title, credits)
- A subset of the attributes form a **primary key** of the entity set; i.e., uniquely identifying each member of the set.

Entity Sets -- *instructor* and *student*

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

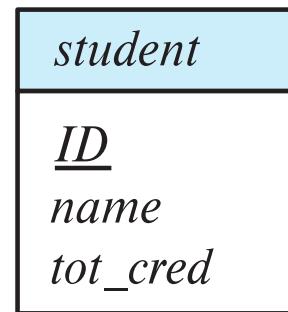
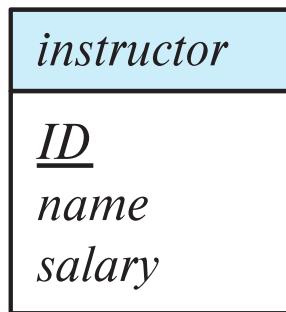
instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Representing Entity sets in ER Diagram

- Entity sets can be represented graphically as follows:
 - Rectangles represent entity sets.
 - Attributes listed inside entity rectangle
 - Underline indicates primary key attributes



Relationship Sets

- A **relationship** is an association among several entities

Example:

44553 (Peltier) advisor 22222 (Einstein)
student entity relationship set instructor entity

- A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

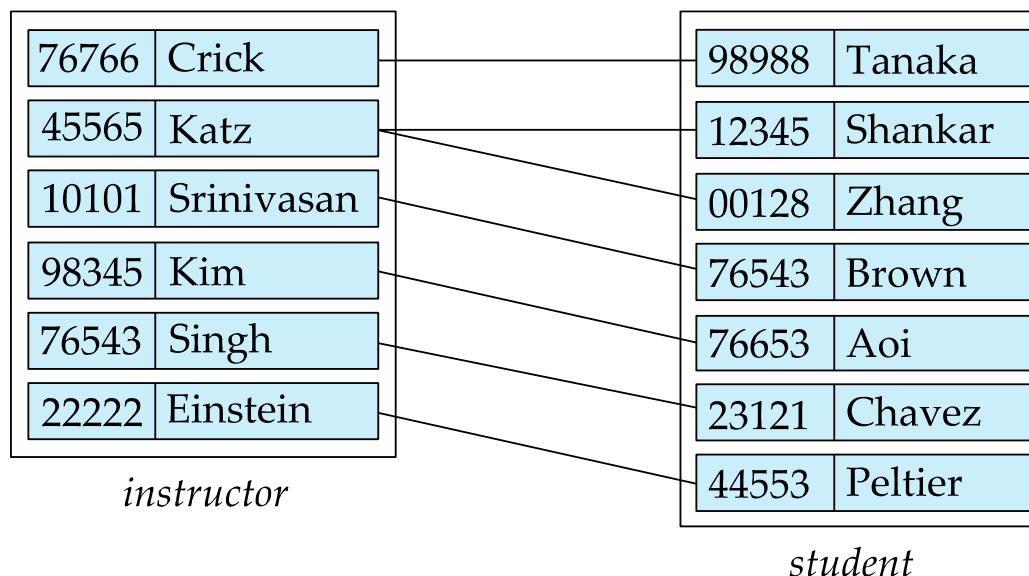
where (e_1, e_2, \dots, e_n) is a relationship

- Example:

$(44553, 22222) \in \text{advisor}$

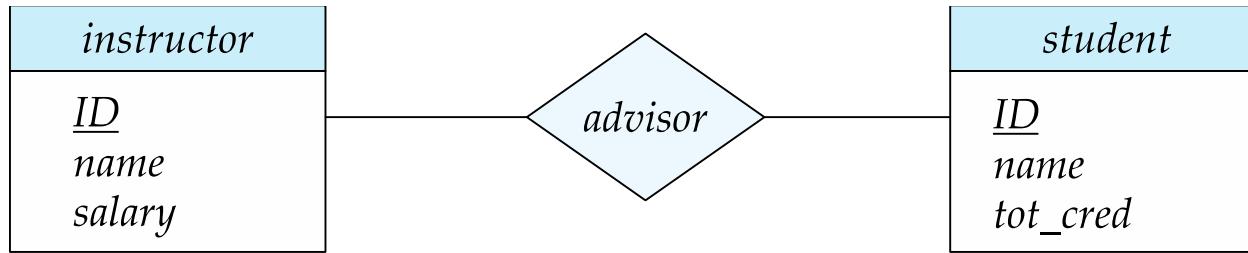
Relationship Sets (Cont.)

- Example: we define the relationship set `advisors` to denote the associations between students and the instructors who act as their advisors.
- Pictorially, we draw a line between related entities.



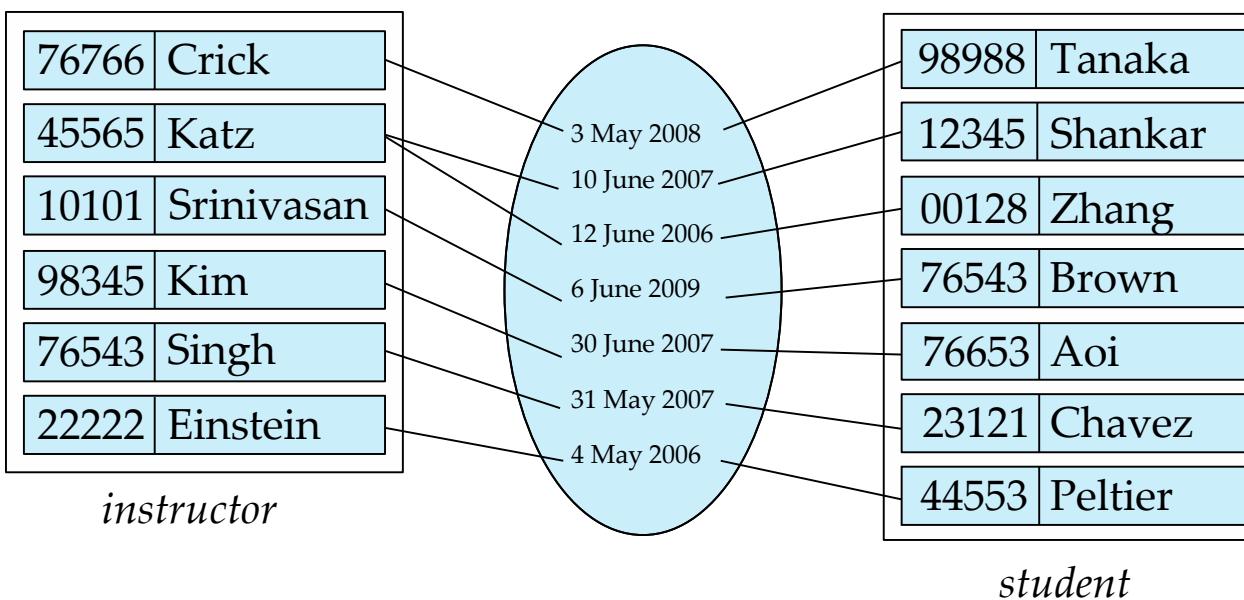
Representing Relationship Sets via ER Diagrams

- Diamonds represent relationship sets.

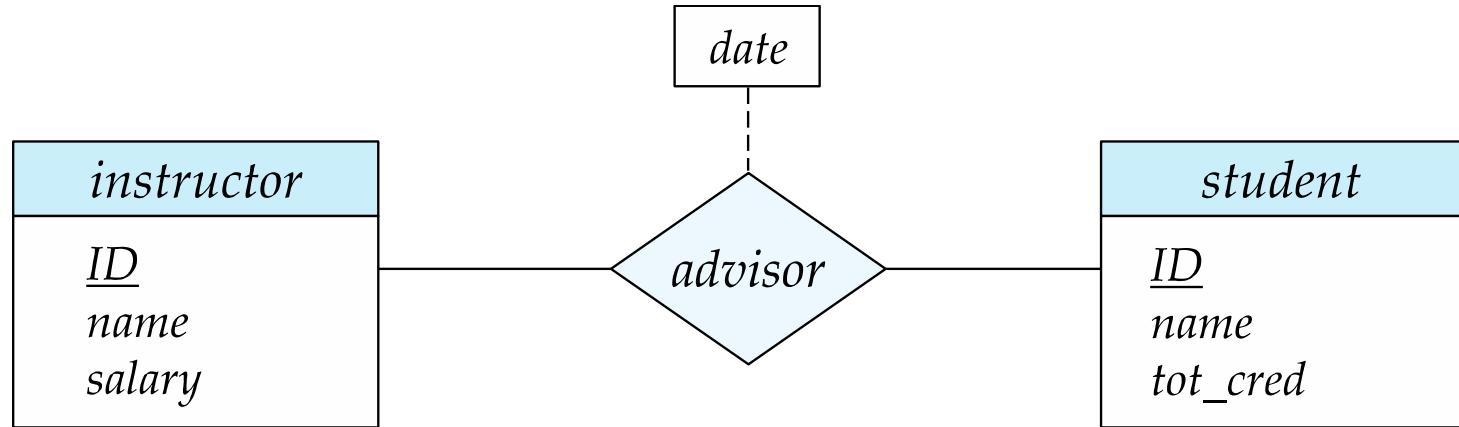


Relationship Sets (Cont.)

- An attribute can also be associated with a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor

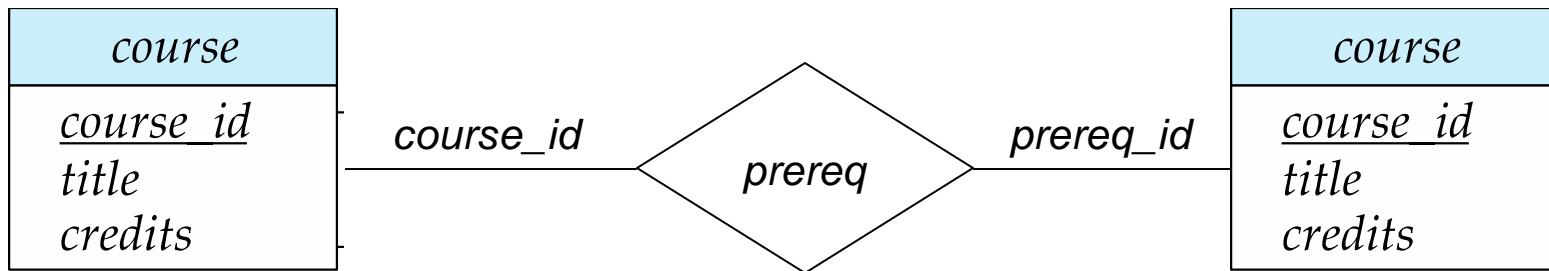


Relationship Sets with Attributes

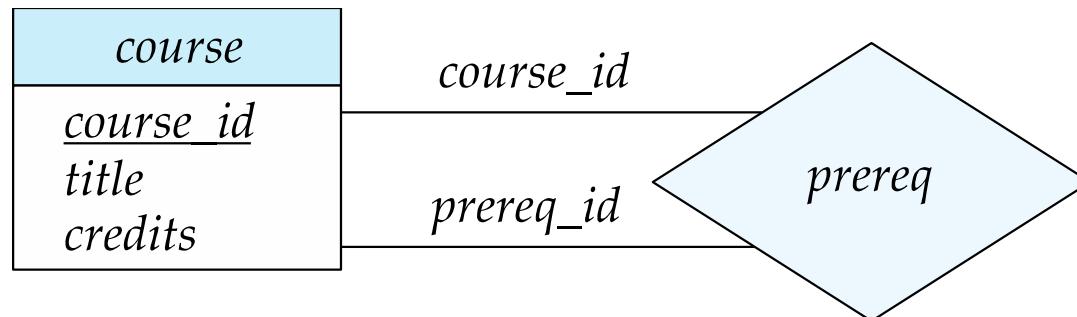


Roles

- Entity sets of a relationship need not be distinct
 - Each occurrence of an entity set plays a “role” in the relationship
- The labels “*course_id*” and “*prereq_id*” are called **roles**.



Database Systems in S4 is a prerequisite for Database Internals in S8

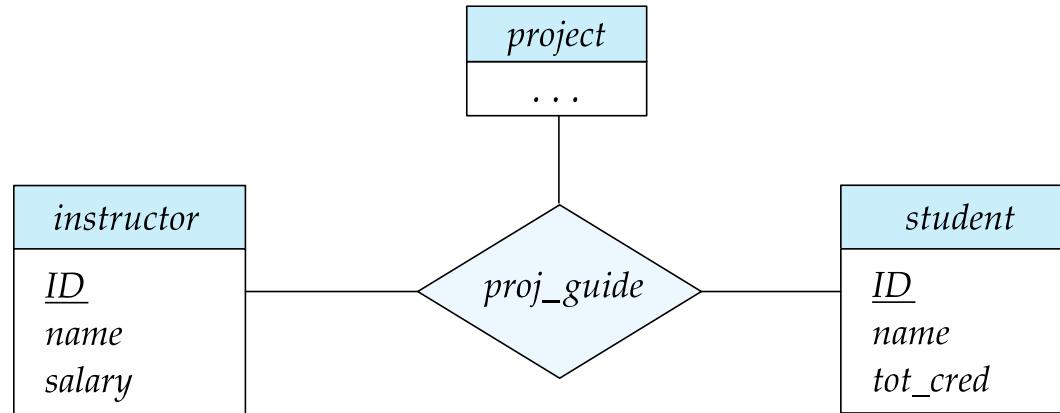


Degree of a Relationship Set

- Binary relationship
 - involve two entity sets (or degree two).
 - most relationship sets in a database system are binary.
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
 - Example: *students* work on research *projects* under the guidance of an *instructor*.
 - relationship *proj_guide* is a ternary relationship between *instructor*, *student*, and *project*

Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship

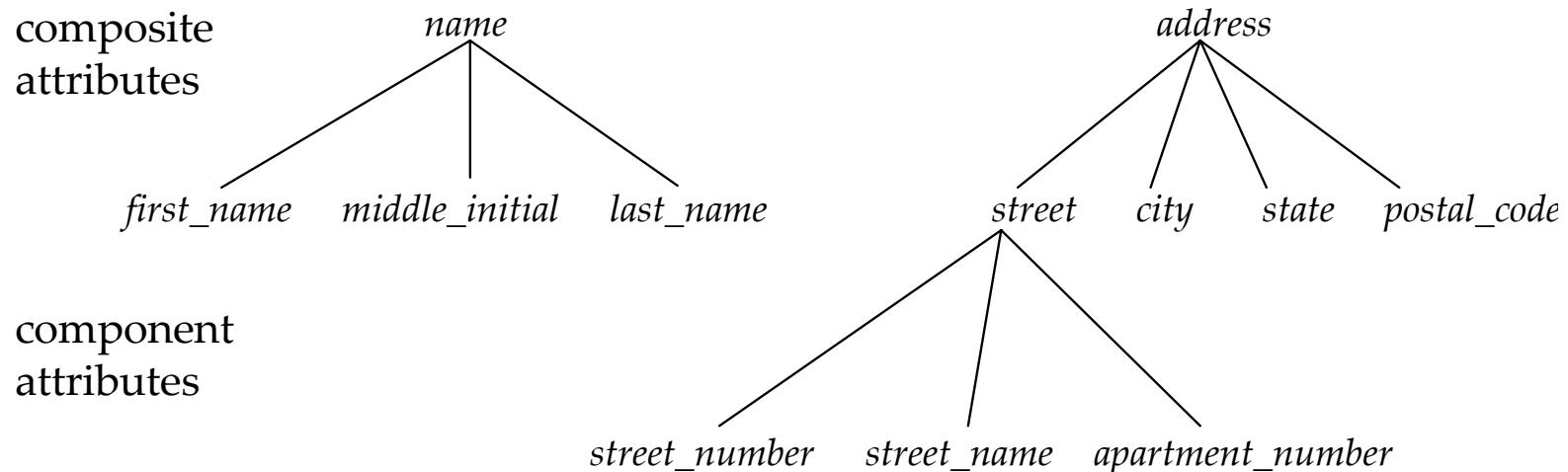


Complex Attributes

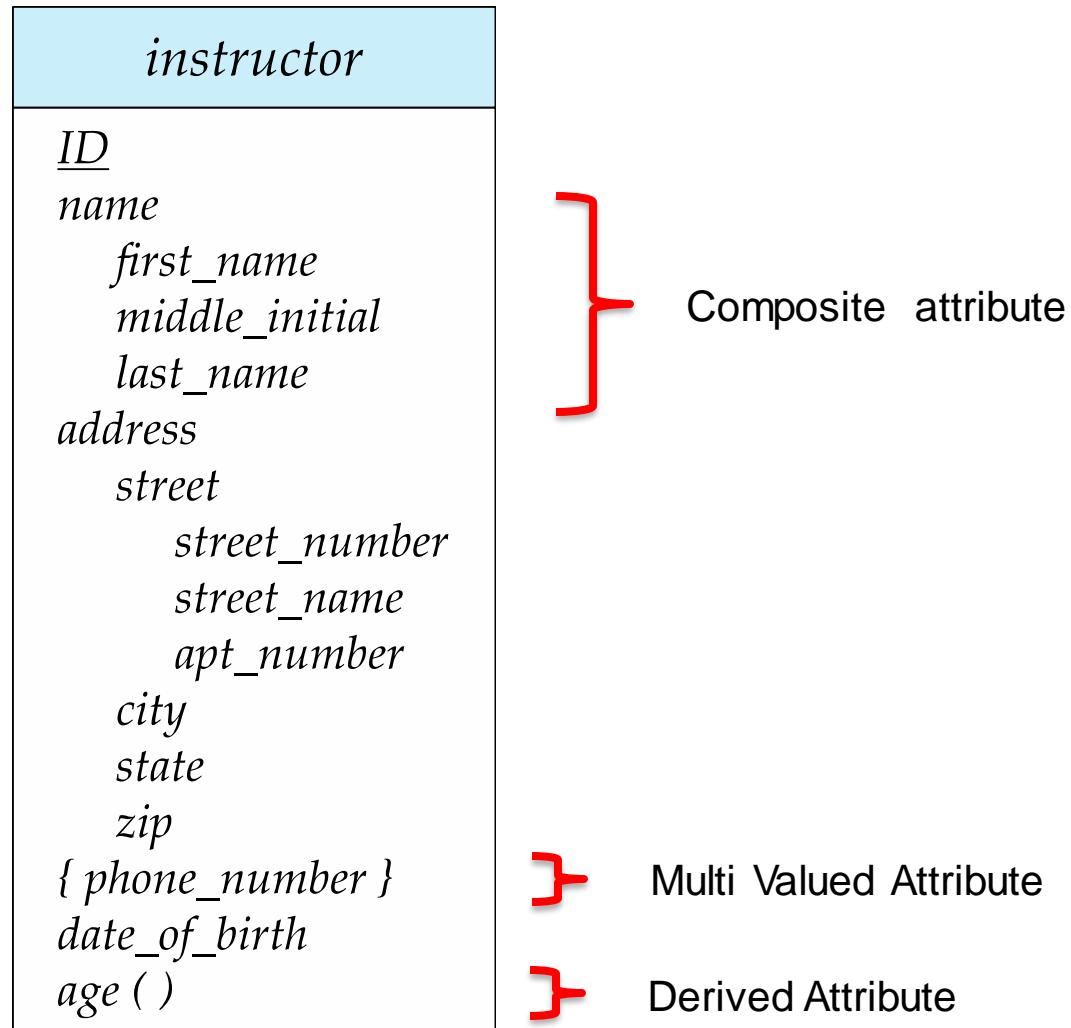
- Attribute types:
 - **Simple** and **composite** attributes.
 - **Single-valued** and **multivalued** attributes
 - Example: multivalued attribute: *phone_numbers*
 - **Derived** attributes
 - Can be computed from other attributes
 - Example: age, given date_of_birth
- **Domain** – the set of permitted values for each attribute

Composite Attributes

- Composite attributes allow us to divide attributes into subparts (other attributes).



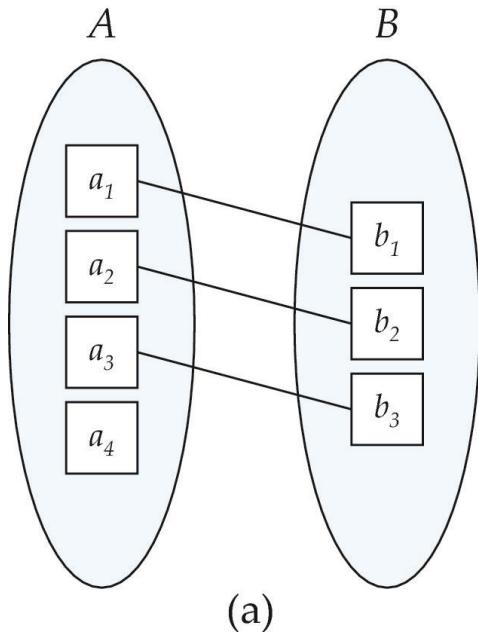
Representing Complex Attributes in ER Diagram



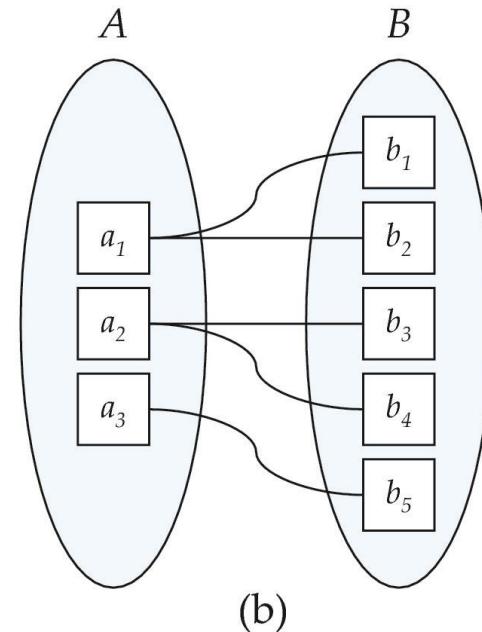
Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
 - One to one
 - One to many
 - Many to one
 - Many to many

Mapping Cardinalities



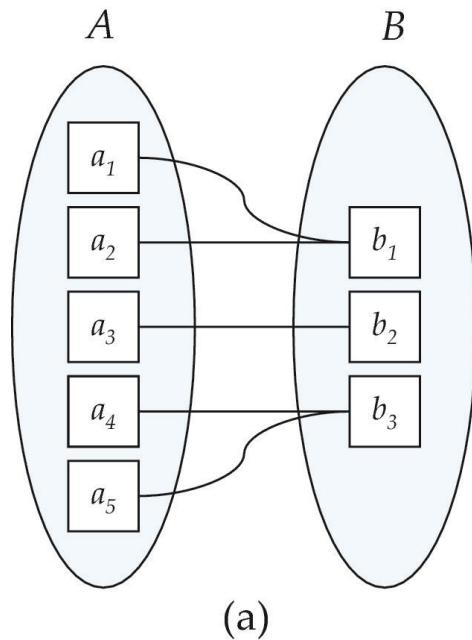
One to one



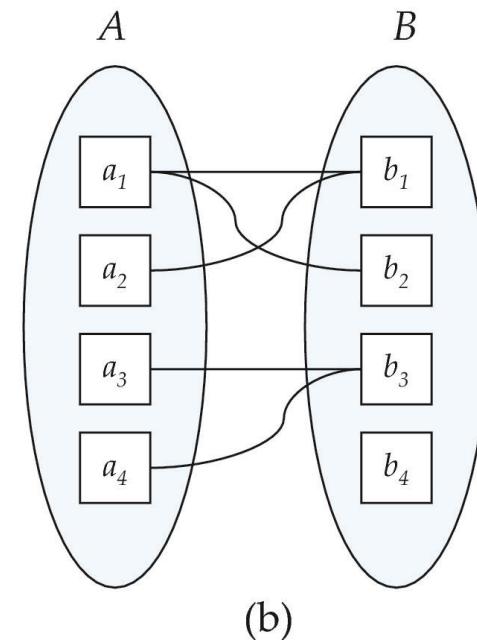
One to many

Note: Some elements in A and B may not be mapped to any elements in the other set

Mapping Cardinalities



Many to one

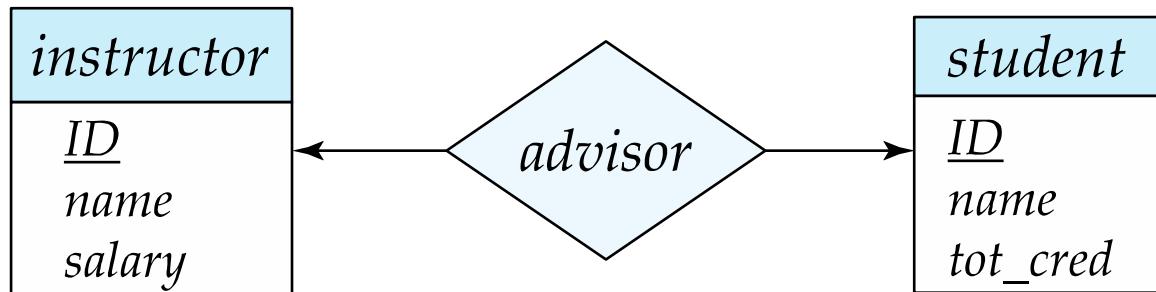


Many to many

Note: Some elements in A and B may not be mapped to any elements in the other set

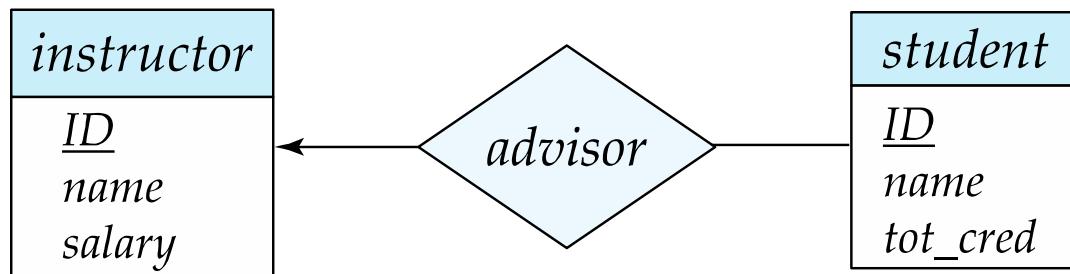
Representing Cardinality Constraints in ER Diagram

- We express cardinality constraints by drawing either a directed line (\rightarrow), signifying “one,” or an undirected line ($-$), signifying “many,” between the relationship set and the entity set.
- One-to-one relationship between an *instructor* and a *student*:
 - A student is associated with at most one *instructor* via the relationship *advisor*
 - A *student* is associated with at most one *instructor* as advisor.



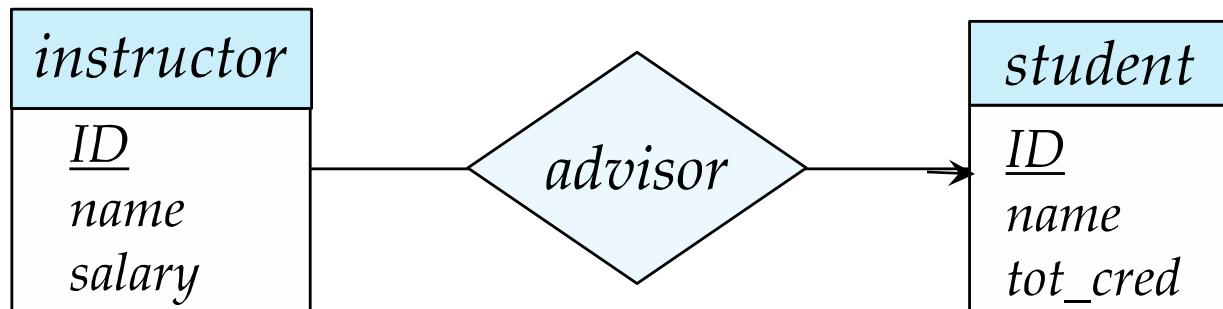
One-to-Many Relationship

- one-to-many relationship between an *instructor* and a *student*
 - an instructor is associated with several (including 0) students via *advisor*
 - a student is associated with at most one instructor via *advisor*,



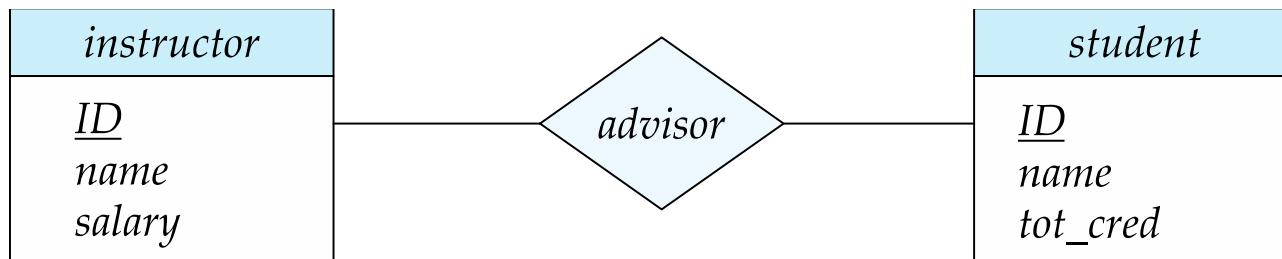
Many-to-One Relationships

- In a many-to-one relationship between an *instructor* and a *student*,
 - an *instructor* is associated with at most one *student* via *advisor*,
 - and a *student* is associated with several (including 0) *instructors* via *advisor*



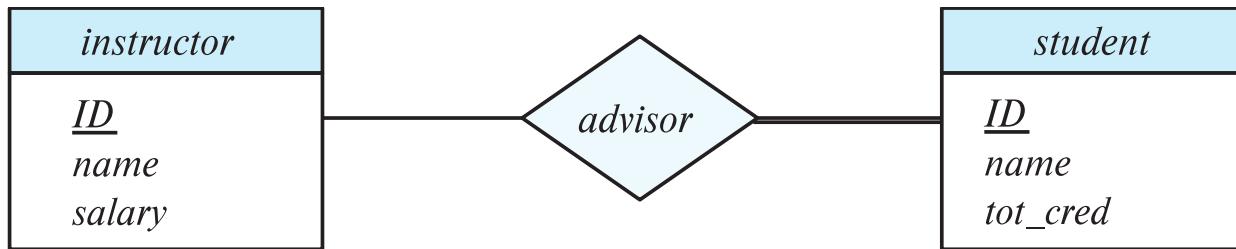
Many-to-Many Relationship

- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*



Total and Partial Participation

- **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set

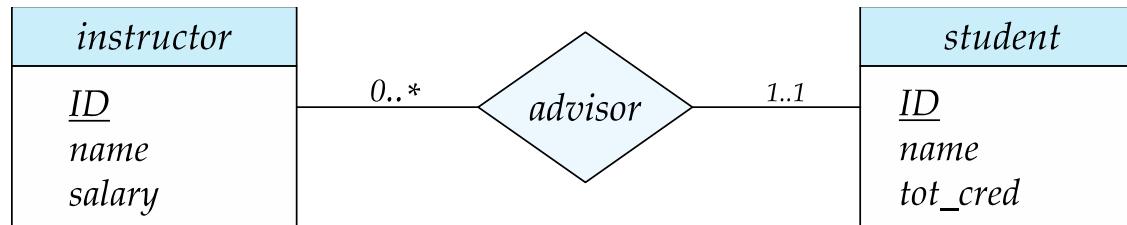


participation of *student* in *advisor* relation is total

- every *student* must have an associated instructor
- **Partial participation:** some entities may not participate in any relationship in the relationship set
 - Example: participation of *instructor* in *advisor* is partial

Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form $l..h$, where l is the minimum and h the maximum cardinality
 - A minimum value of 1 indicates total participation.
 - A maximum value of 1 indicates that the entity participates in at most one relationship
 - A maximum value of * indicates no limit.
- Example



- Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors

Cardinality Constraints on Ternary Relationships

- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- For example, an arrow from *proj_guide* to *instructor* indicates each student has at most one guide for a project
- If there is more than one arrow, there are two ways of defining the meaning.
 - For example, a ternary relationship R between A , B and C with arrows to B and C could mean
 1. Each A entity is associated with a unique entity from B and C or
 2. Each pair of entities from (A, B) is associated with a unique C entity, and each pair (A, C) is associated with a unique B
 - Each alternative has been used in different formalisms
 - To avoid confusion we outlaw more than one arrow

Primary Key

- Primary keys provide a way to specify how entities and relations are distinguished. We will consider:
 - Entity sets
 - Relationship sets.
 - Weak entity sets

Primary key for Entity Sets

- By definition, individual entities are distinct.
- From database perspective, the differences among them must be expressed in terms of their attributes.
- The values of the attribute values of an entity must be such that they can uniquely identify the entity.
 - No two entities in an entity set are allowed to have exactly the same value for all attributes.
- A key for an entity is a set of attributes that suffice to distinguish entities from each other

Primary Key for Relationship Sets

- To distinguish among the various relationships of a relationship set we use the individual primary keys of the entities in the relationship set.
 - Let R be a relationship set involving entity sets E_1, E_2, \dots, E_n
 - The primary key for R consists of the union of the primary keys of entity sets E_1, E_2, \dots, E_n
 - If the relationship set R has attributes a_1, a_2, \dots, a_m associated with it, then the primary key of R also includes the attributes a_1, a_2, \dots, a_m
- Example: relationship set “advisor”.
 - The primary key consists of $instructor.ID$ and $student.ID$
- The choice of the primary key for a relationship set depends on the mapping cardinality of the relationship set.

Choice of Primary key for Binary Relationship

- Many-to-Many relationships. The preceding union of the primary keys is a minimal superkey and is chosen as the primary key.
- One-to-Many relationships . The primary key of the “Many” side is a minimal superkey and is used as the primary key.
- Many-to-one relationships. The primary key of the “Many” side is a minimal superkey and is used as the primary key.
- One-to-one relationships. The primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key.

Weak Entity Sets

- Consider the following entities.
 - $\text{course} = \{\underline{\text{course_id}}, \text{title}, \text{credits}\}$
 - $\text{section} = \{\underline{\text{course_id}}, \underline{\text{sec_id}}, \text{semester}, \text{year}, \text{building}\}$
- *Section* entity, which is uniquely identified by a course_id , semester , year , and sec_id .
- Clearly, section entities are related to course entities. Suppose we create a relationship set sec_course between entity sets *section* and *course*.
- Note that the information in sec_course is redundant, since *section* already has an attribute course_id , which identifies the course with which the section is related.
- One option to deal with this redundancy is to get rid of the relationship sec_course ; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.

Weak Entity Sets (Cont.)

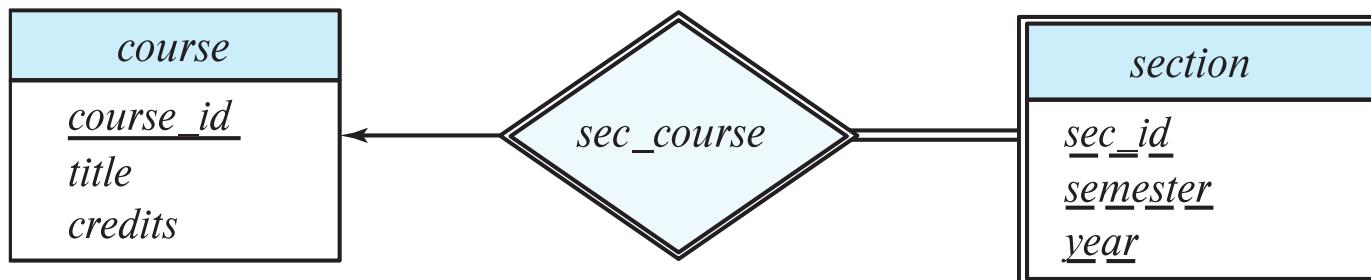
- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester*.
 - However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely
- To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case, the *course_id*, required to identify *section* entities uniquely.
- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**
- Instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity.

Weak Entity Sets (Cont.)

- An entity set that is not a weak entity set is termed a **strong entity set**.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
- The identifying entity set is said to **own** the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- The set of attributes that allows distinguishing among weak entities is called the **discriminator** or **partial key**.
- Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped the attribute *course_id* from the entity set *section*.

Expressing Weak Entity Sets (Ex 1)

- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course_id*, *sec_id*, *semester*, *year*)



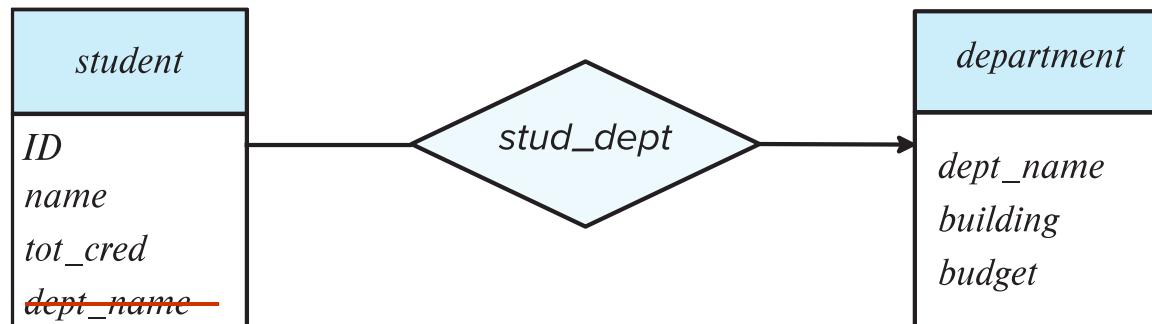
Expressing Weak Entity Sets (Ex 2)

- Consider how banks store details about children (for saving accounts). Since a child cannot be uniquely identified (no NIC) we associate a child with an adult who is his guardian.
- Assume that one guardian would not name all his children with the same name. Then name becomes a partial key for the weak entity set child and adult is the associated identifying entity set.
- Primary key for child is (NIC, name)



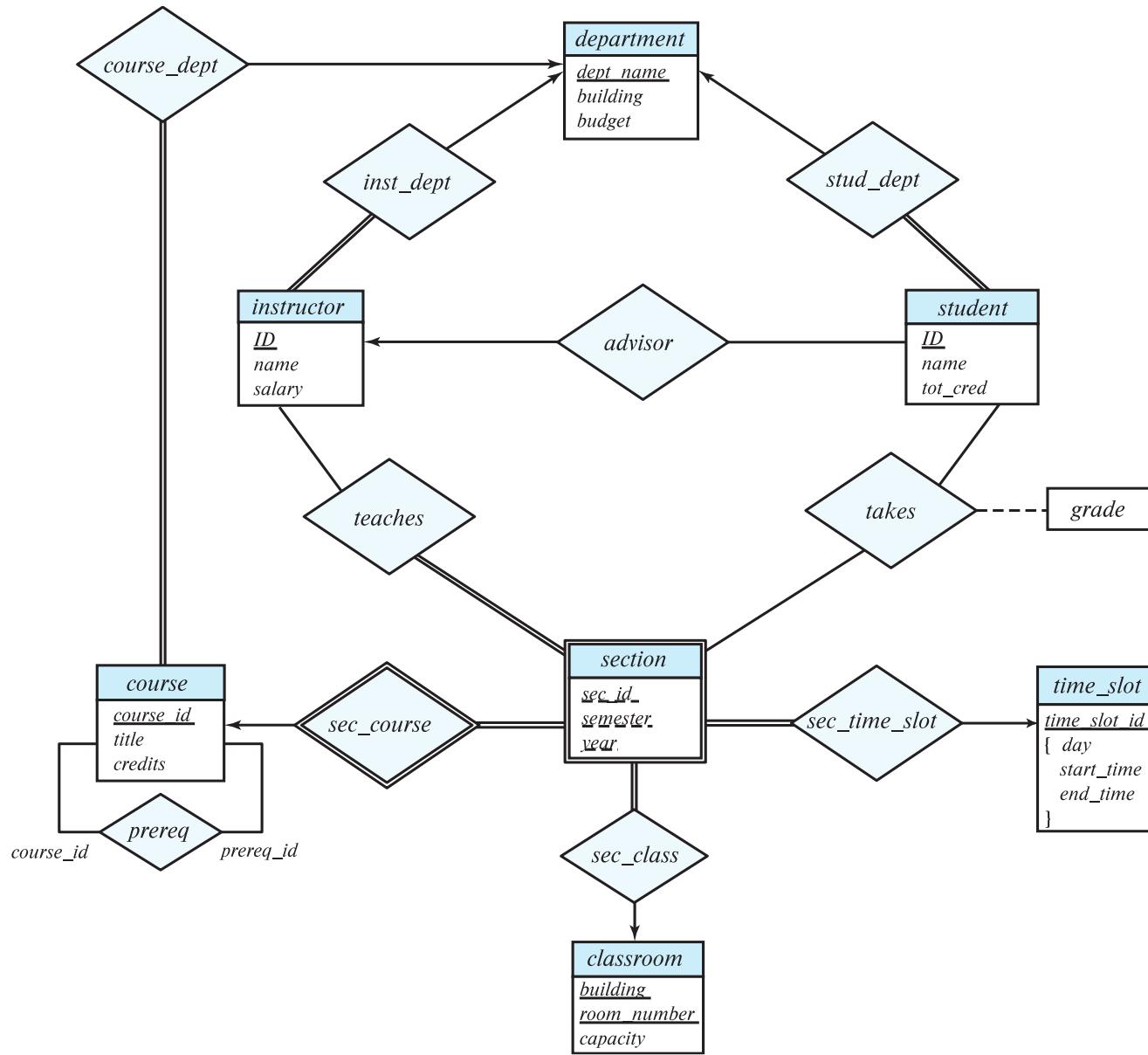
Redundant Attributes

- Suppose we have entity sets:
 - *student*, with attributes: *ID*, *name*, *tot_cred*, *dept_name*
 - *department*, with attributes: *dept_name*, *building*, *budget*
- We model the fact that each student has an associated department using a relationship set *stud_dept*
- The attribute *dept_name* in *student* below replicates information present in the relationship and is therefore redundant
 - and needs to be removed.
- BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see later.



(a) Incorrect use of attribute

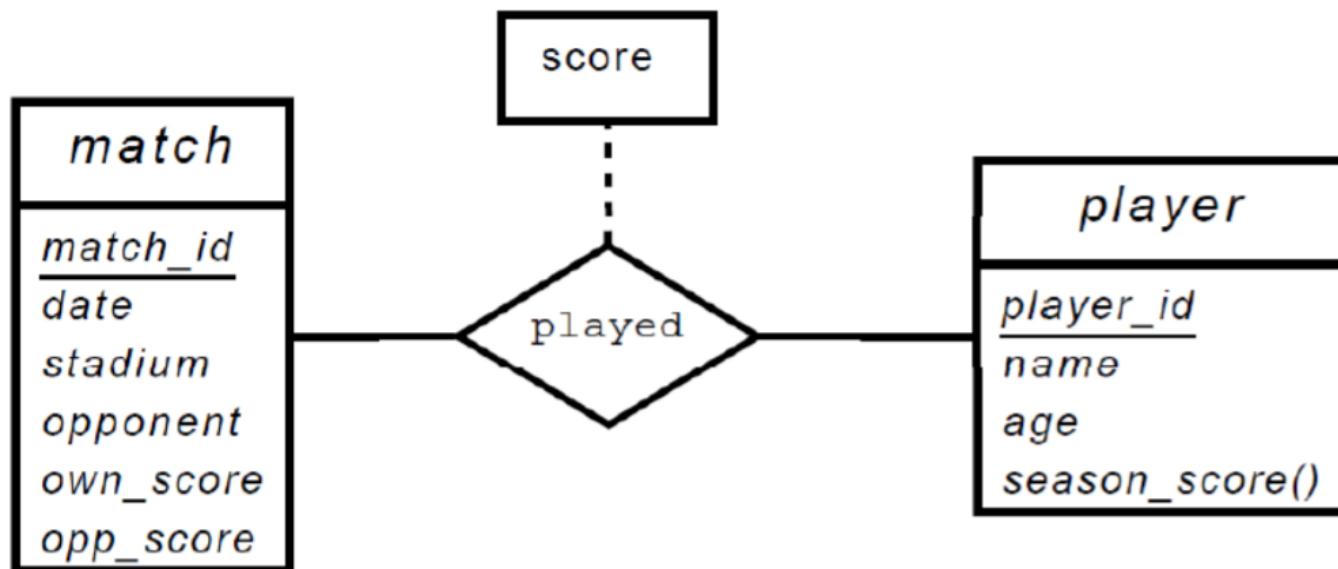
E-R Diagram for a University Enterprise



Exercise

Design an ER diagram for keeping track of the exploits of your favourite SLPL team. You should store the matches played, the scores in each match, the players in each match, individual player statistics (batting only) for each match. Summary statistics should be modeled derived attributes

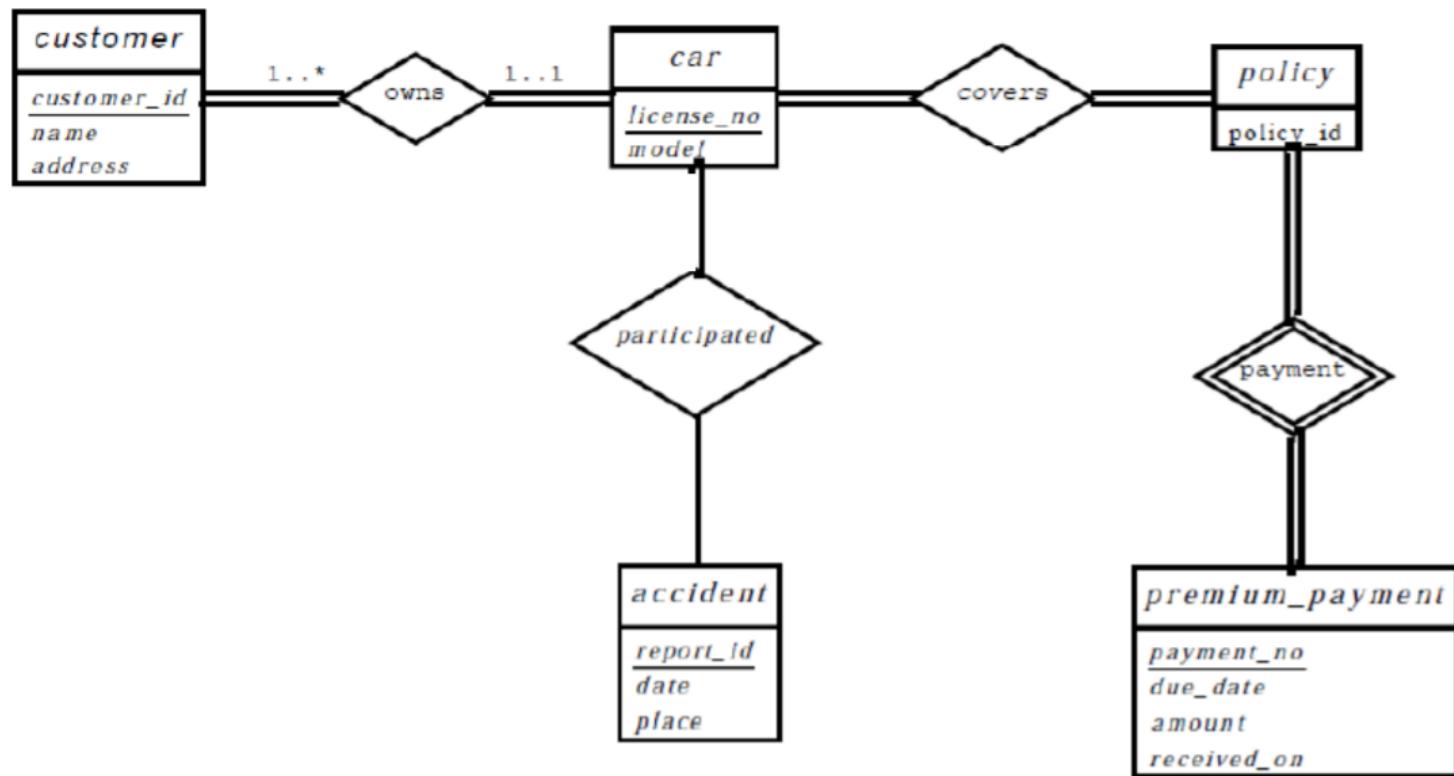
Sample answer



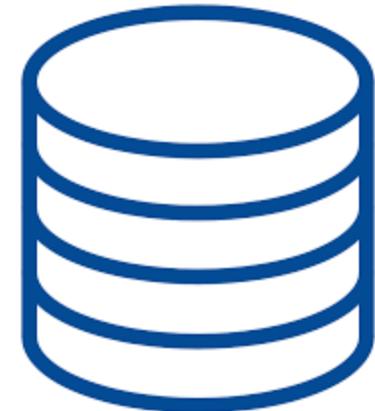
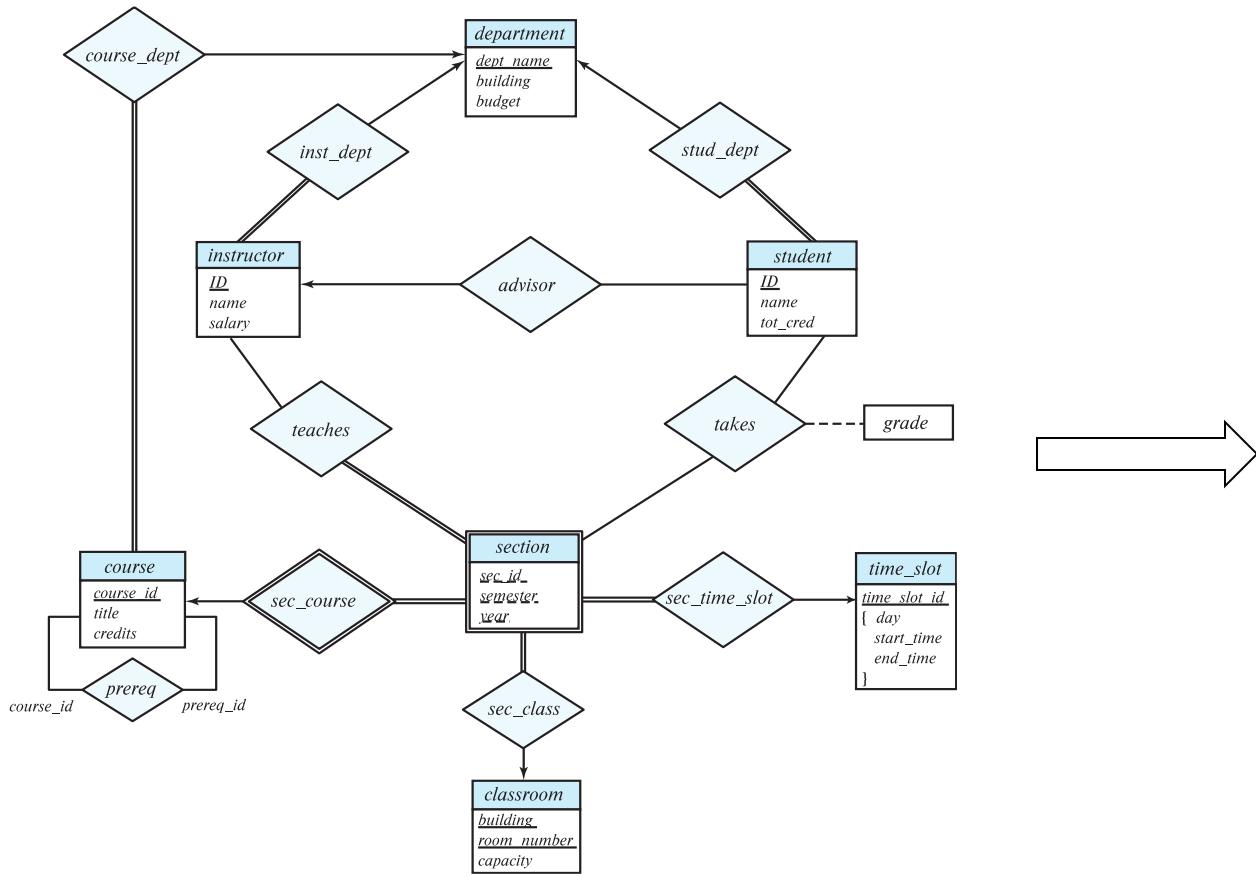
Exercise

Construct and ER diagram for a car Insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars, and has one or more premium payment associated with it. Each payment is for a particular period of time and has an associated due date, and the date when the payment was received.

Sample answer



Reduction to Relation Schemas



Reduction to Relation Schemas

- Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of schemas.
- For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set.
- Each schema has a number of columns (generally corresponding to attributes), which have unique names.

Representation of Entity Sets with Composite Attributes

<i>instructor</i>
<i>ID</i>
<i>name</i>
<i>first_name</i>
<i>middle_initial</i>
<i>last_name</i>
<i>address</i>
<i>street</i>
<i>street_number</i>
<i>street_name</i>
<i>apt_number</i>
<i>city</i>
<i>state</i>
<i>zip</i>
{ <i>phone_number</i> }
<i>date_of_birth</i>
<i>age()</i>

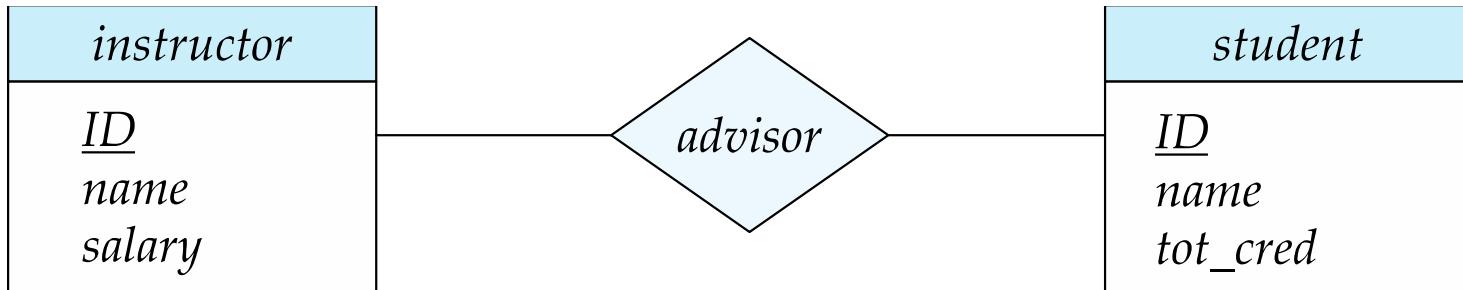
- Composite attributes are flattened out by creating a separate attribute for each component attribute
 - Example: given entity set *instructor* with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes *name_first_name* and *name_last_name*
 - Prefix omitted if there is no ambiguity (*name_first_name* could be *first_name*)
- Ignoring multivalued attributes, extended instructor schema is
 - *instructor*(*ID*,
 first_name, *middle_initial*, *last_name*,
 street_number, *street_name*,
 apt_number, *city*, *state*, *zip_code*,
 date_of_birth)

Representation of Entity Sets with Multivalued Attributes

- A multivalued attribute M of an entity E is represented by a separate schema EM
- Schema EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M
- Example: Multivalued attribute $phone_number$ of $instructor$ is represented by a schema:
 $inst_phone=(\underline{ID}, \underline{phone_number})$
- Each value of the multivalued attribute maps to a separate tuple of the relation on schema EM
 - For example, an $instructor$ entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples:
 - (22222, 456-7890) and
 - (22222, 123-4567)

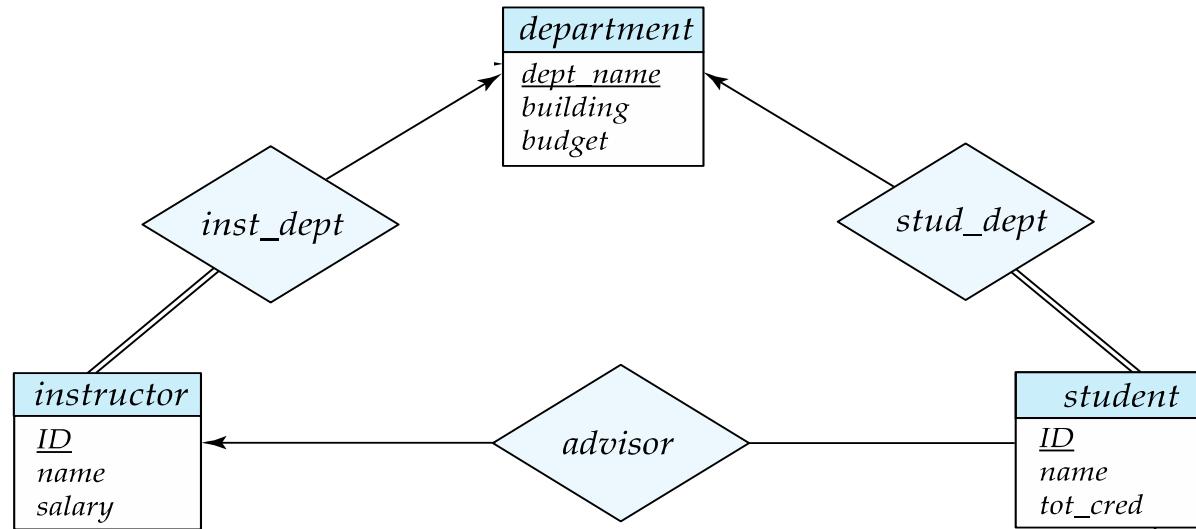
Representing Relationship Sets

- A many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- Example: schema for relationship set *advisor*
 - ***advisor* = (s_id, i_id)**
 - student = (s_id, name, tot_cred)
 - instructor = (i_id, name, salary)



Redundancy of Schemas

- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the “many” side, containing the primary key of the “one” side
- Example: Instead of creating a schema for relationship set *inst_dept*, add an attribute *dept_name* to the schema arising from entity set *instructor*
- Example

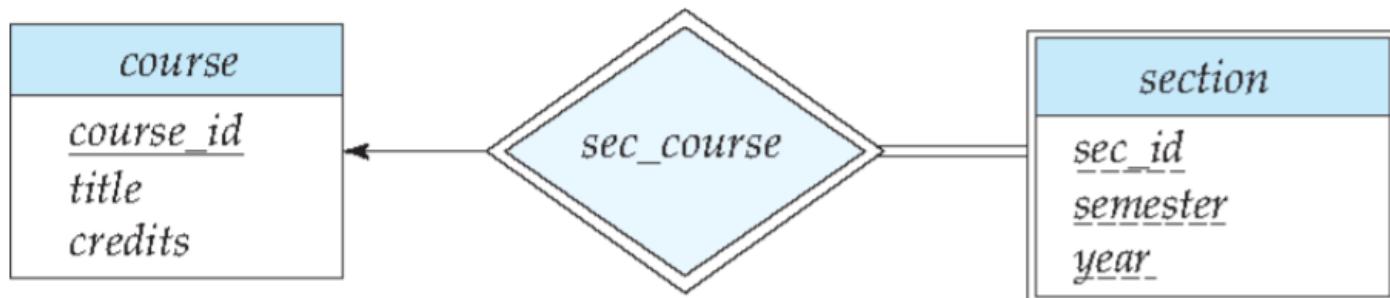


Redundancy of Schemas (Cont.)

- For one-to-one relationship sets, either side can be chosen to act as the “many” side
 - That is, an extra attribute can be added to either of the tables corresponding to the two entity sets
- If participation is *partial* on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values

Redundancy of Schemas (Cont.)

- The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.
- Example: The *section* schema already contains the attributes that would appear in the *sec_course* schema



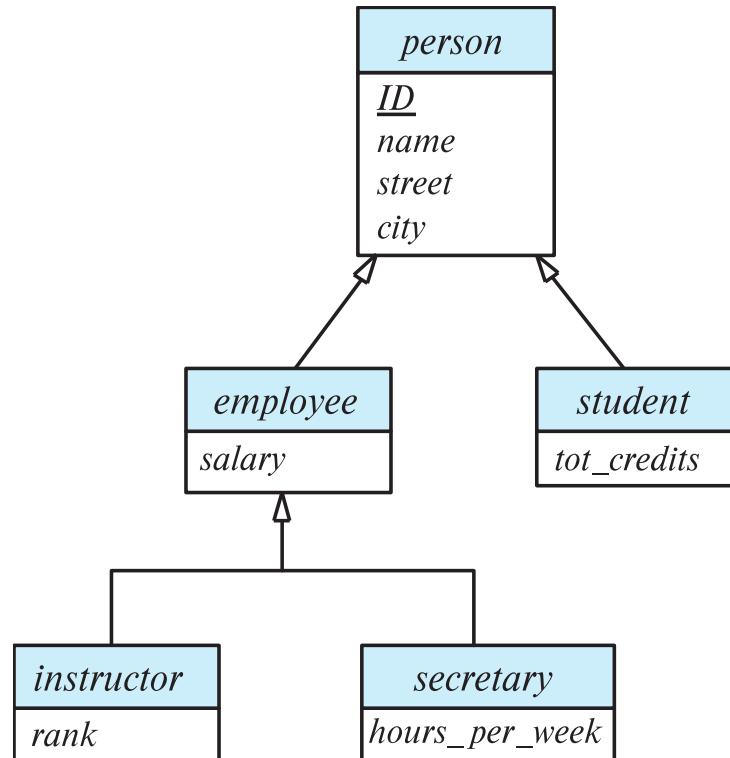
Extended E-R Features

Specialization

- Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set.
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* “is a” *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Specialization Example

- **Overlapping** – *employee* and *student*
- **Disjoint** – *instructor* and *secretary*
- Total and partial



Representing Specialization via Schemas

- Method 1:
 - Form a schema for the higher-level entity
 - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

- Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema

Representing Specialization as Schemas (Cont.)

- Method 2:

- Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary

- Drawback: *name*, *street* and *city* may be stored redundantly for people who are both students and employees

Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.

Completeness constraint

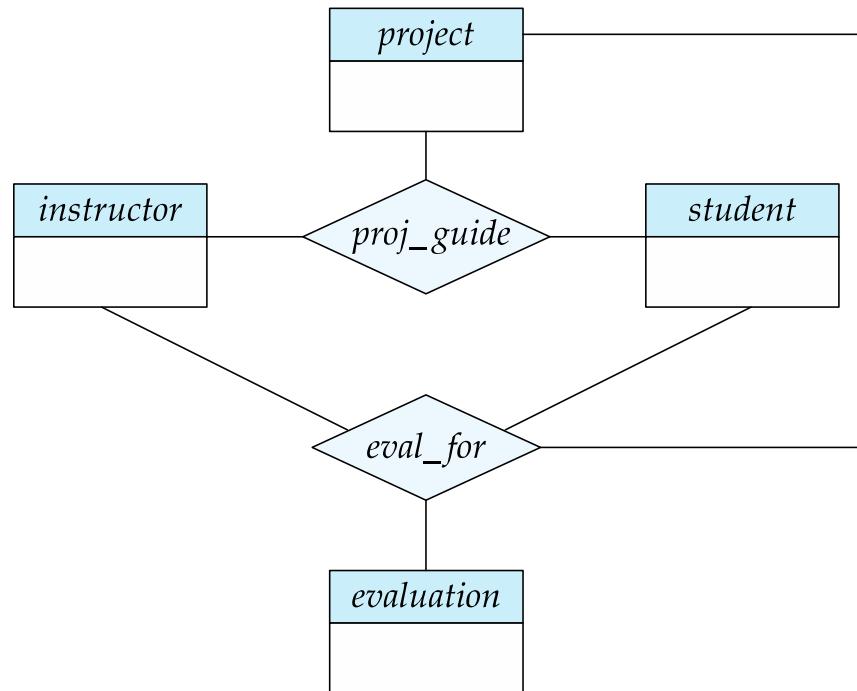
- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
 - **total**: an entity must belong to one of the lower-level entity sets
 - **partial**: an entity need not belong to one of the lower-level entity sets

Completeness constraint (Cont.)

- Partial generalization is the default.
- We can specify total generalization in an ER diagram by adding the keyword **total** in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).
- The *student* generalization is total: All student entities must be either graduate or undergraduate. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets, the completeness constraint for a generalized higher-level entity set is usually total

Aggregation

- Consider the ternary relationship *proj_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project

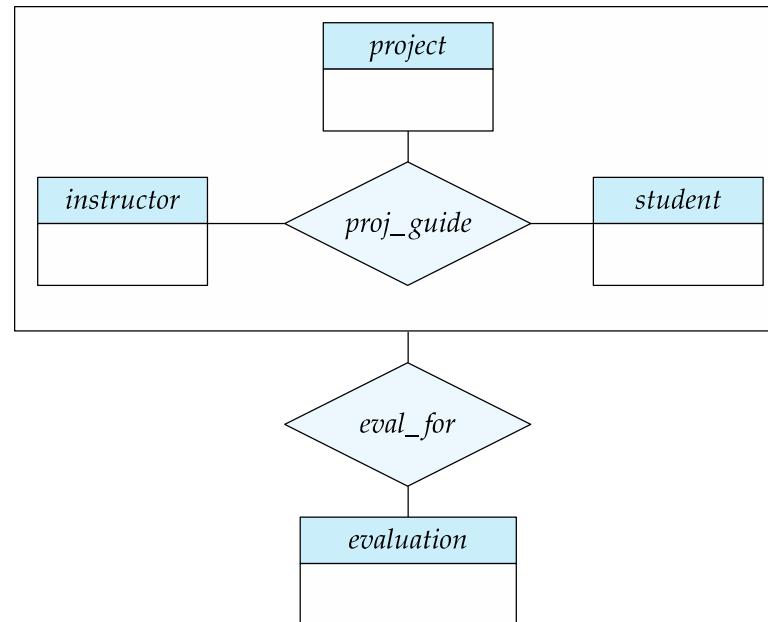


Aggregation (Cont.)

- Relationship sets *eval_for* and *proj_guide* represent overlapping information
 - Every *eval_for* relationship corresponds to a *proj_guide* relationship
 - However, some *proj_guide* relationships may not correspond to any *eval_for* relationships
 - So we can't discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
 - Treat relationship as an abstract entity
 - Allows relationships between relationships
 - Abstraction of relationship into new entity

Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
 - A student is guided by a particular instructor on a particular project
 - A student, instructor, project combination may have an associated evaluation



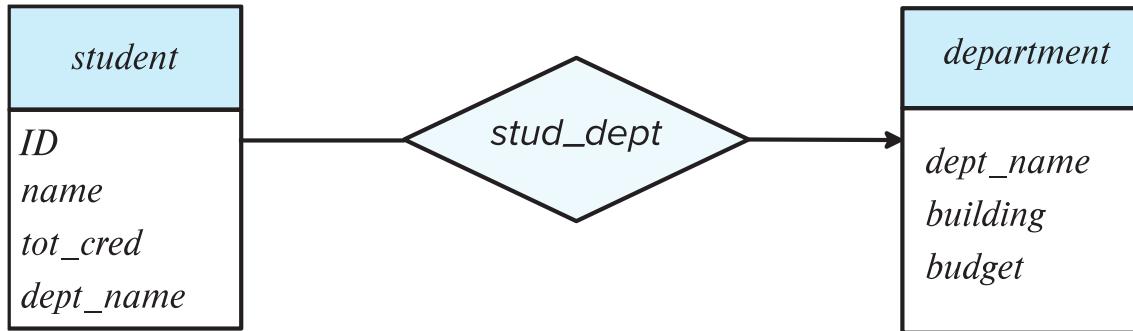
Reduction to Relational Schemas

- To represent aggregation, create a schema containing
 - Primary key of the aggregated relationship,
 - The primary key of the associated entity set
 - Any descriptive attributes
- In our example:
 - The schema *eval_for* is:
 $\text{eval_for}(s_ID, project_id, i_ID, evaluation_id)$
 - The schema *proj_guide* is redundant.

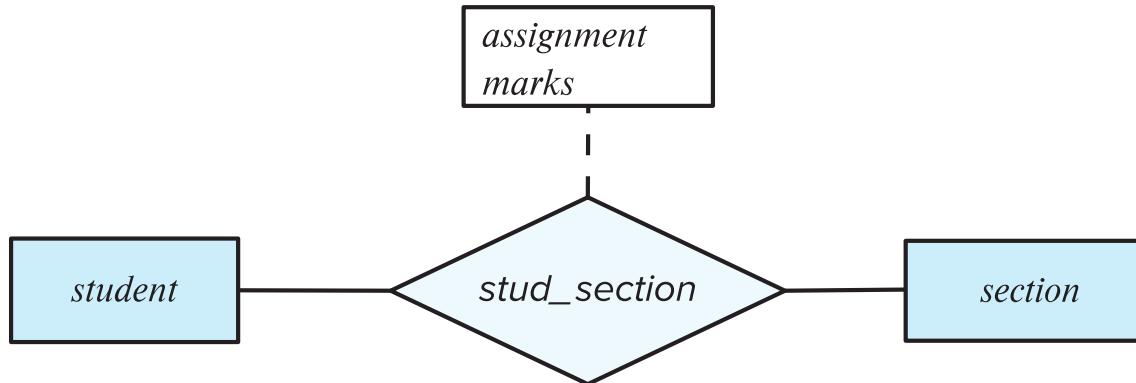
Design Issues

Common Mistakes in E-R Diagrams

- Example of erroneous E-R diagrams

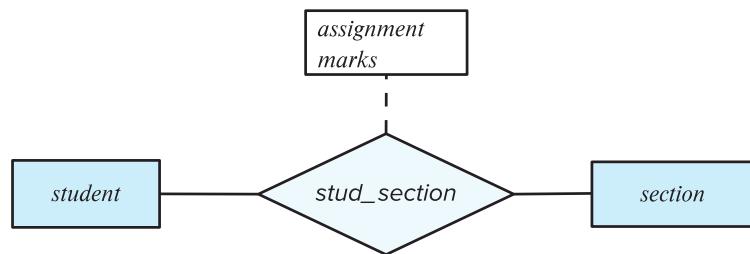


(a) Incorrect use of attribute

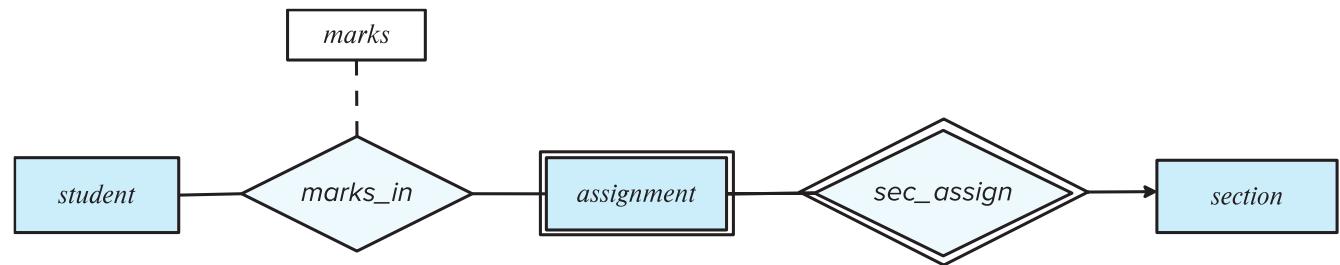


(b) Erroneous use of relationship attributes

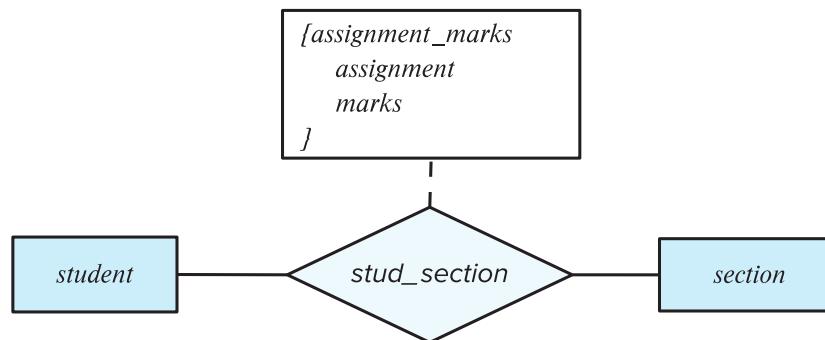
Common Mistakes in E-R Diagrams (Cont.)



(b) Erroneous use of relationship attributes



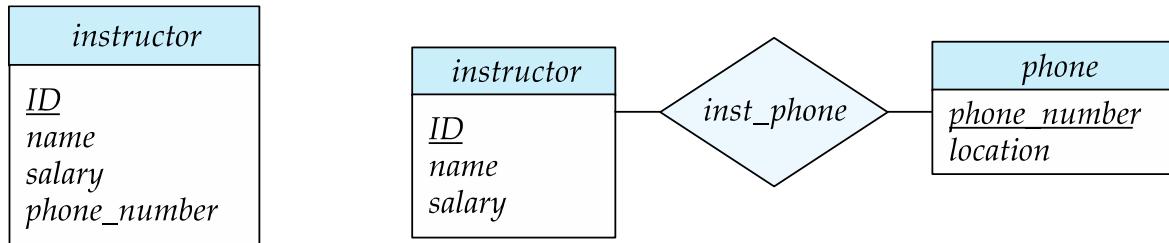
(c) Correct alternative to erroneous E-R diagram (b)



(d) Correct alternative to erroneous E-R diagram (b)

Entities vs. Attributes

- Use of entity sets vs. attributes

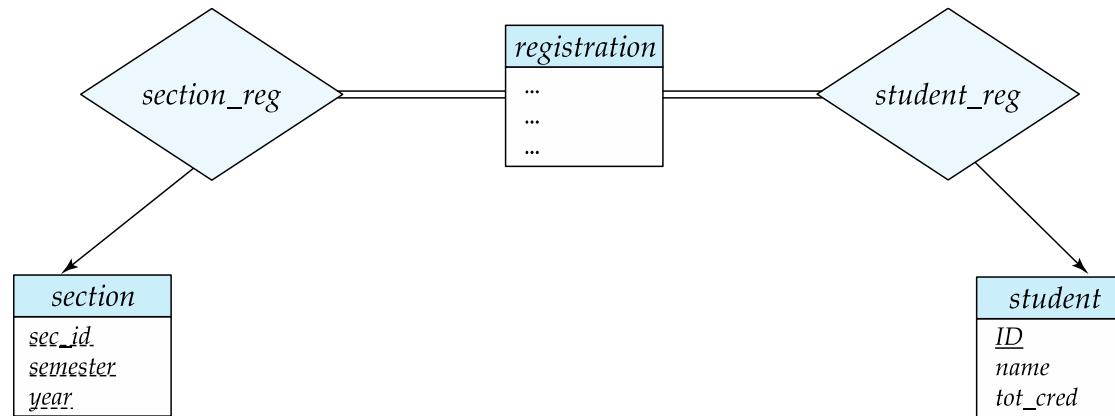


- Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)

Entities vs. Relationship sets

- **Use of entity sets vs. relationship sets**

Possible guideline is to designate a relationship set to describe an action that occurs between entities



- **Placement of relationship attributes**

For example, attribute date as attribute of advisor or as attribute of student

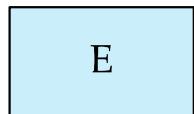
Binary Vs. Non-Binary Relationships

- Although it is possible to replace any non-binary (n -ary, for $n > 2$) relationship set by a number of distinct binary relationship sets, a n -ary relationship set shows more clearly that several entities participate in a single relationship.
- Some relationships that appear to be non-binary may be better represented using binary relationships
 - For example, a ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
 - Using two binary relationships allows partial information (e.g., only mother being known)
 - But there are some relationships that are naturally non-binary
 - Example: *proj_guide*

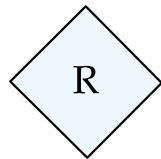
E-R Design Decisions

- The use of an attribute or entity set to represent an object.
- Whether a real-world concept is best expressed by an entity set or a relationship set.
- The use of a ternary relationship versus a pair of binary relationships.
- The use of a strong or weak entity set.
- The use of specialization/generalization – contributes to modularity in the design.
- The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure.

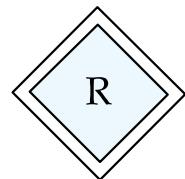
Summary of Symbols Used in E-R Notation



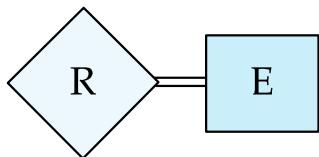
entity set



relationship set



identifying
relationship set
for weak entity set



total participation
of entity set in
relationship

E
A1
A2
A2.1
A2.2
{A3}
A4()

attributes:
simple (A1),
composite (A2) and
multivalued (A3)
derived (A4)

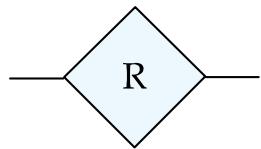
E
<u>A1</u>

primary key

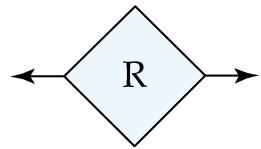
E
----- A1

discriminating
attribute of
weak entity set

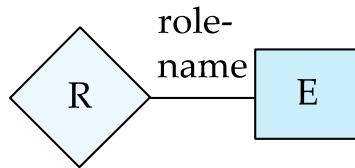
Symbols Used in E-R Notation (Cont.)



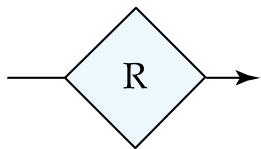
many-to-many
relationship



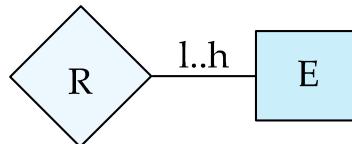
one-to-one
relationship



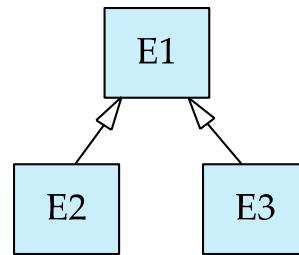
role indicator



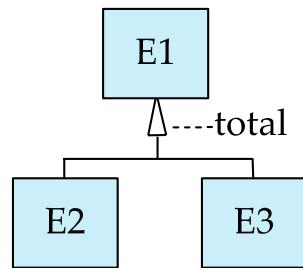
many-to-one
relationship



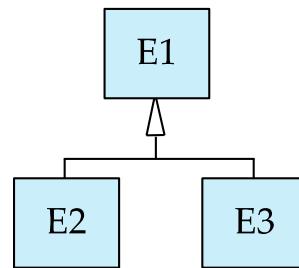
cardinality
limits



ISA: generalization
or specialization



total (disjoint)
generalization

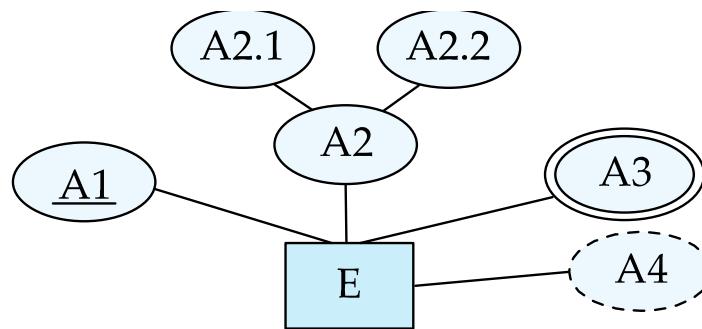


disjoint
generalization

Alternative ER Notations

- Chen, IDE1FX, ...

entity set E with
simple attribute A1,
composite attribute A2,
multivalued attribute A3,
derived attribute A4,
and primary key A1



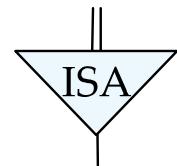
weak entity set



generalization



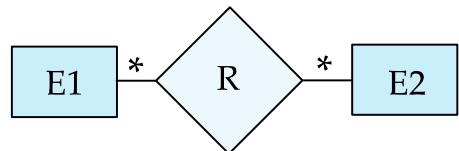
total
generalization



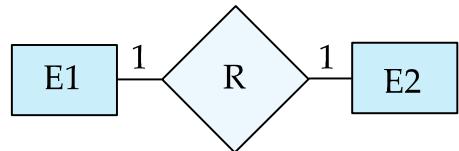
Alternative ER Notations

Chen

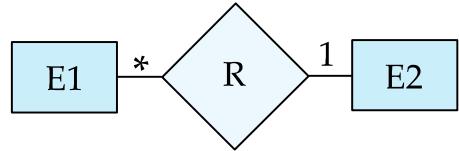
many-to-many
relationship



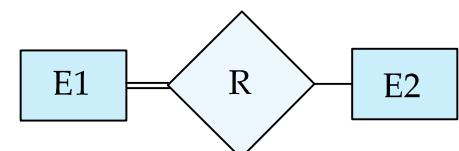
one-to-one
relationship



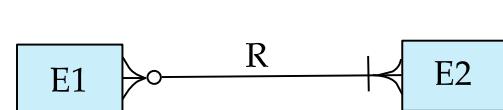
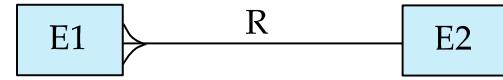
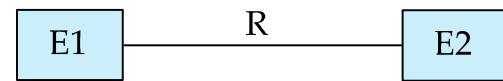
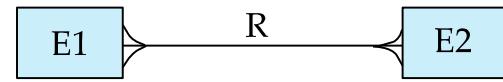
many-to-one
relationship



participation
in R: total (E1)
and partial (E2)



IDE1FX (Crows feet notation)

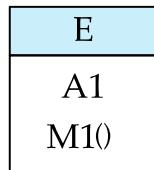


UML

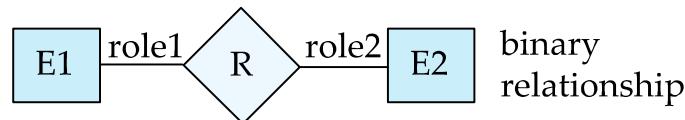
- **UML:** Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
- UML Class Diagrams correspond to E-R Diagram, but several differences.

ER vs. UML Class Diagrams

ER Diagram Notation

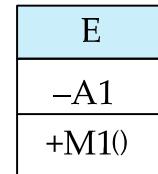


entity with attributes (simple, composite, multivalued, derived)

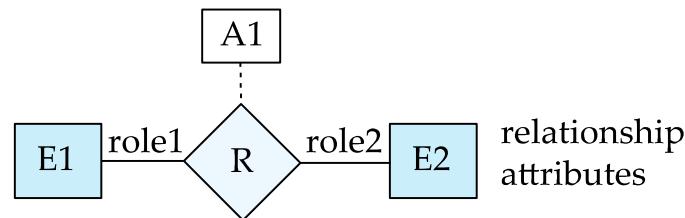


binary relationship

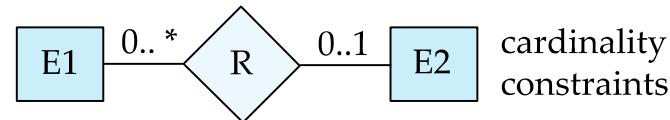
Equivalent in UML



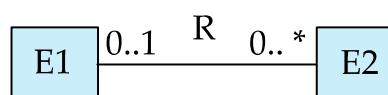
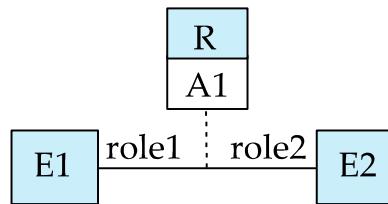
class with simple attributes and methods (attribute prefixes: + = public, - = private, # = protected)



relationship attributes



cardinality constraints



* Note reversal of position in cardinality constraint depiction

Other Aspects of Database Design

- Functional Requirements
- Data Flow, Workflow
- Schema Evolution

End of Chapter 6

Introduction to SQL

CS3042 - Database Systems

Dr Gayashan Amarasinghe

Department of Computer Science and Engineering
University of Moratuwa

Overview

- Brief History
- SQL as a language
- Data Types in SQL
- CREATE TABLE construct and Integrity Constraints
- DROP and ALTER TABLE constructs
- SELECT clause
- WHERE clause
- FROM clause
- Joins
- Natural join
- Rename operation
- Ordering the results
- String operations
- Aggregate functions
- Nested Subqueries

Database System Concepts 6th Edition
by Abraham Silberschatz, Henry F. Korth,
and S. Sudarshan

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
 - SQL-86, SQL-89, SQL-92
 - SQL:1999, SQL:2003, SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- Not all examples here may work on your particular system.

Structured Query Language

SQL is a...

- **Very high level** language
 - Works well because it is optimized well.
- **Data Definition Language**

CREATE TABLE

DROP TABLE

- **Data Manipulation Language**

SELECT

INSERT

DELETE

UPDATE

Data types in SQL

- **char(n).** Fixed length character string, with user-specified length n .
- **varchar(n).** Variable length character strings, with user-specified maximum length n .
- **int.** Integer (a finite subset of the integers that is machine dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least n digits.

CREATE TABLE construct

- An sql relation is defined using **CREATE TABLE** construct

CREATE TABLE r(A₁ D₁, A₂ D₂, ..., A_n D_n,
(integrity-constraint₁),
...,
(integrity-constraint_k))

- r - name of the relation
- A_i - Attribute name in the schema of the relation
- D_i - Data type of values in the domain of attribute A_i

Eg:

```
CREATE TABLE instructor (  
    ID char(5),  
    name varchar(20) not null,  
    dept_name varchar(20),  
    salary numeric(8,2))
```

Integrity constraints in CREATE TABLE

- **not null**
- **primary key** (A_1, A_2, \dots, A_n)
- **foreign key** (A_m, \dots, A_n) **references** r

Eg: Declare ID as the primary key for the instructor table.

```
create table instructor (
    ID char(5),
    name varchar(20) not null,
    dept_name varchar(20),
    salary numeric(8,2),
    primary key (ID),
    foreign key (dept_name)
        references department(dept_name))
```

- **primary key** declaration on an attribute automatically ensures **not null**

DROP and ALTER TABLE constructs

- **DROP TABLE** student
 - Deletes the table and its content
- **DELETE FROM** student
 - Deletes all the content of the table, but retains the table
- **ALTER TABLE**
 - **ALTER TABLE r ADD A D**
 - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain/data type of *A*.
 - All tuples in the relation are assigned *null* as the value for the new attribute.
 - **ALTER TABLE r DROP A**
 - where *A* is the name of an attribute in relation *r*.
 - Many databases do not support this functionality.

Basic Query Structure

- SQL is also a Data Manipulation language
- A typical SQL query has the form

```
SELECT A1,A2,...,An  
FROM r1,r2,...,rm  
WHERE P
```

- A_i represents an attribute
 - r_j represents a relation
 - P is a predicate
-
- The result of a query is another relation

SELECT clause

- The **SELECT** clause lists the attributes desired in the result of a query
 - corresponds to the projection operation of the relational algebra
- Eg: Find the names of all the departments with instructors

```
SELECT dept_name  
FROM instructor
```

SQL names are case insensitive.
name \equiv Name \equiv NAME

- To force the elimination of duplicates, insert the keyword **DISTINCT** after **SELECT**.
- Eg:

```
SELECT DISTINCT dept_name  
FROM instructor
```

SELECT clause

- An asterisk in the SELECT clause denotes “all attributes”

Eg:

```
SELECT * FROM instructor
```

- The **SELECT** clause can contain arithmetic expressions involving the operation, +, −, , and /, and operating on constants or attributes of tuples.

Eg:

```
SELECT ID, name, salary/12  
FROM instructor
```

would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

WHERE clause

- **WHERE** clause specifies conditions that the result must satisfy
 - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept with salary > 80000

SELECT *name*

FROM *instructor*

WHERE *dept_name* = 'Comp. Sci.' **AND** *salary* > 80000

- Comparison results can be combined using the logical connectives **and**, **or**, and **not**.
- Comparisons can be applied to results of arithmetic expressions.

FROM clause

- **FROM** clause lists the relations involved in the query
 - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

SELECT *

FROM *instructor, teaches*

- generates every possible instructor – teaches pair, with all attributes from both relations
- Cartesian product is not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra)

Joins

- For all instructors who have taught some course, find their names and the course ID of the courses they taught.

```
SELECT name, course_id  
FROM instructor, teaches  
WHERE instructor.ID = teaches.ID
```

Cartesian product of instructor relation and teaches relation

instructor x teaches

- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department

```
SELECT section.course_id, semester, year, title  
FROM section, course  
WHERE section.course_id = course.course_id  
AND dept_name = 'Comp. Sci.'
```

Writing some queries

Instructor(ID, name, dept_name, salary)

<u>ID</u>	<u>name</u>	<u>dept_name</u>	<u>salary</u>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
.....

Teaches(ID, course_id, sec_id, semester, year)

<u>ID</u>	<u>course_id</u>	<u>sec_id</u>	<u>semester</u>	<u>year</u>
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

- Write the SQL query and a sample output.

1. Find all the names of instructors in the Music Dept.
2. Find all the names of instructors who make more than 50000
3. Find all the names of instructors who are in the Music dept. and make more than 60000
4. Find all the course id's of courses taught in the Fall semester in year 2009.
5. Find all the names of instructors and the corresponding course id that they teach.
6. Find all the course id's taught by "Mozart" in Fall 2011.

```
select name  
from instructor  
where dept_name = 'Music'  
> Mozart
```

0

Natural Join

- **NATURAL JOIN** matches tuples with the same values for all common attributes, and retains only one copy of each common column.

SELECT *

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010

Natural Join example

- List the names of instructors along with the course ID of the courses that they taught.
 - Without NATURAL JOIN

```
SELECT name, course_id  
FROM instructor, teaches  
WHERE instructor.id = teaches.id
```

- With NATURAL JOIN

```
SELECT name, course_id  
FROM instructor NATURAL JOIN teaches
```

Be cautious with Natural Join

- Dangers in NATURAL JOIN
 - beware of unrelated attributes with same name which get equated incorrectly
- List the names of instructors along with the titles of courses that they teach
 - Incorrect version (makes course.dept_name = instructor.dept_name)

```
SELECT name, title  
FROM instructor NATURAL JOIN teaches NATURAL JOIN course
```

- Correct version

```
SELECT name, title  
FROM instructor NATURAL JOIN teaches, course  
WHERE teaches.course_id = course.course_id
```

- Another correct version

```
SELECT name, title  
FROM (instructor NATURAL JOIN teaches) JOIN course USING(course_id)
```

Rename operation

- SQL allows renaming relations and attributes using the **AS** clause

```
SELECT ID, name, salary/12 AS monthly_salary  
FROM instructor
```

- Find the names of all instructors who have a higher salary than some instructor in ‘Comp. Sci’.

```
SELECT DISTINCT T.name  
FROM instructor AS T, instructor AS S  
WHERE T.salary > S.salary AND S.dept_name = ‘Comp. Sci.’
```

- Keyword **AS** is optional and can be omitted.

```
SELECT ID, name, salary/12 monthly_salary  
FROM instructor
```

Ordering the results

- List in alphabetic order the names of all instructors

```
SELECT DISTINCT name  
FROM instructor  
ORDER BY name
```

Ascending order is the default.

- We may specify **DESC** for descending order or **ASC** for ascending order, for each attribute.

Example: **ORDER BY** *name DESC*

- Can sort on multiple attributes

Example: **ORDER BY** *dept_name, name*

String operations

- **LIKE** operator can be used for string matching

```
SELECT name  
FROM instructor  
WHERE name LIKE "%dar%"
```

%	matches any substring
_	matches any character
\	escape character for % or _

- Patterns are case sensitive
- SQL supports a variety of string operations such as
 - concatenation (using “||”)
 - converting from upper to lowercase (and vice versa)
 - finding string length, extracting substrings, etc.

WHERE clause predicates

- SQL includes a **BETWEEN** comparison operator

Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, \$90,000 and \$100,000)

```
SELECT name  
FROM instructor  
WHERE salary BETWEEN 90000 AND 100000
```

- Tuple comparison

```
SELECT name, course_id  
FROM instructor, teaches  
WHERE (instructor.ID, dept_name) = (teaches.ID, 'Biology')
```

Aggregate functions

- These functions operate on the multiset of values of a column of a relation, and return a value
 - **AVG**: average value
 - **MIN**: minimum value
 - **MAX**: maximum value
 - **SUM**: sum of values
 - **COUNT**: number of values
- Find the average salary of instructors in the Computer Science department

```
SELECT AVG(salary)  
FROM instructor  
WHERE dept_name = 'Comp. Sci.');
```

Aggregate functions

- Find the total number of instructors who teach a course in the Spring 2010 semester

SELECT COUNT (DISTINCT *ID*)

FROM teaches

WHERE semester = 'Spring' **AND** year = 2010

- Find the number of tuples in the course relation

SELECT COUNT (*)

FROM course;

Aggregate functions - GROUP BY clause

- Find the average salary of instructors in each department

```
SELECT dept_name, AVG(salary)  
FROM instructor  
GROUP BY dept_name
```

- Attributes in **SELECT** clause outside of aggregate functions must appear in **GROUP BY** list

```
/* erroneous query */  
SELECT dept_name, ID, AVG(salary)  
FROM instructor  
GROUP BY dept_name
```

Aggregate functions - HAVING clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

```
SELECT dept_name, AVG(salary)  
FROM instructor  
GROUP BY dept_name  
HAVING AVG(salary) > 42000
```

Predicates in the **HAVING** clause are applied after the formation of groups whereas predicates in the **WHERE** clause are applied before forming groups.

Aggregate functions - NULL values

- All aggregate operations **except COUNT(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
 - count returns 0
 - all other aggregates return null

Eg: Total salary of all the instructors

```
SELECT SUM(salary)
```

```
FROM instructor
```

- this statement ignores null values
- result becomes null, if there are only null values in salary column.

Nested subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Eg: Find courses offered in Fall 2009 and in Spring 2010

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND year= 2009 AND
course_id IN (SELECT course_id
FROM section
WHERE semester = 'Spring'
AND year= 2010)
```

Nested subqueries

Find courses offered in Fall 2009 but not in Spring 2010

```
SELECT DISTINCT course_id  
FROM section  
WHERE semester = 'Fall' AND year= 2009 AND  
course_id NOT IN (SELECT course_id  
FROM section  
WHERE semester = 'Spring'  
AND year= 2010)
```

Thank you!

Practice task

Employee(**emp_no**, **emp_name**, **emp_city**,)

Assignment(**proj_no**, **emp_no**, **hours**,.....)

Project(**proj_name**, **budget**, **proj_no**, **proj_start_date**, **proj_end_date**,
proj_location,.....)

Express each of the following queries in SQL statements:

1. List the name(s) and budget(s) of projects started before 1st May 2008.
2. List the name(s) of projects with a budget value above Rs. 1,000,000.
3. Find the name(s) of employees who are from city “Moratuwa”.
4. Find the name(s) of employees who are from city “Moratuwa” and work on projects located in “Moratuwa”.
5. Find the name(s) of employees who work on projects valued above Rs. 1,000,000.

Chapter 7: Relational Database Design P1

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use

Outline

- Features of Good Relational Design
- Functional Dependencies
- Decomposition Using Functional Dependencies
- Normal Forms
- Functional Dependency Theory
- Algorithms for Decomposition using Functional Dependencies
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Atomic Domains and First Normal Form
- Database-Design Process
- Modeling Temporal Data

Overview of Normalization

Features of Good Relational Designs

- Suppose we combine *instructor* and *department* into *in_dep*, which represents the natural join on the relations *instructor* and *department*

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

- There is repetition of information
- Need to use null values (if we add a new department with no instructors)

A Combined Schema Without Repetition

Not all combined schemas result in repetition of information

- Consider combining relations
 - $\text{sec_class(sec_id, building, room_number)}$ and
 - $\text{section(course_id, sec_id, semester, year)}$
- into one relation
 - $\text{section(course_id, sec_id, semester, year, building, room_number)}$
- No repetition in this case

Decomposition

- The only way to avoid the repetition-of-information problem in the `in_dep` schema is to decompose it into two schemas – `instructor` and `department` schemas.
- Not all decompositions are good. Suppose we decompose

employee(ID, name, street, city, salary)

into

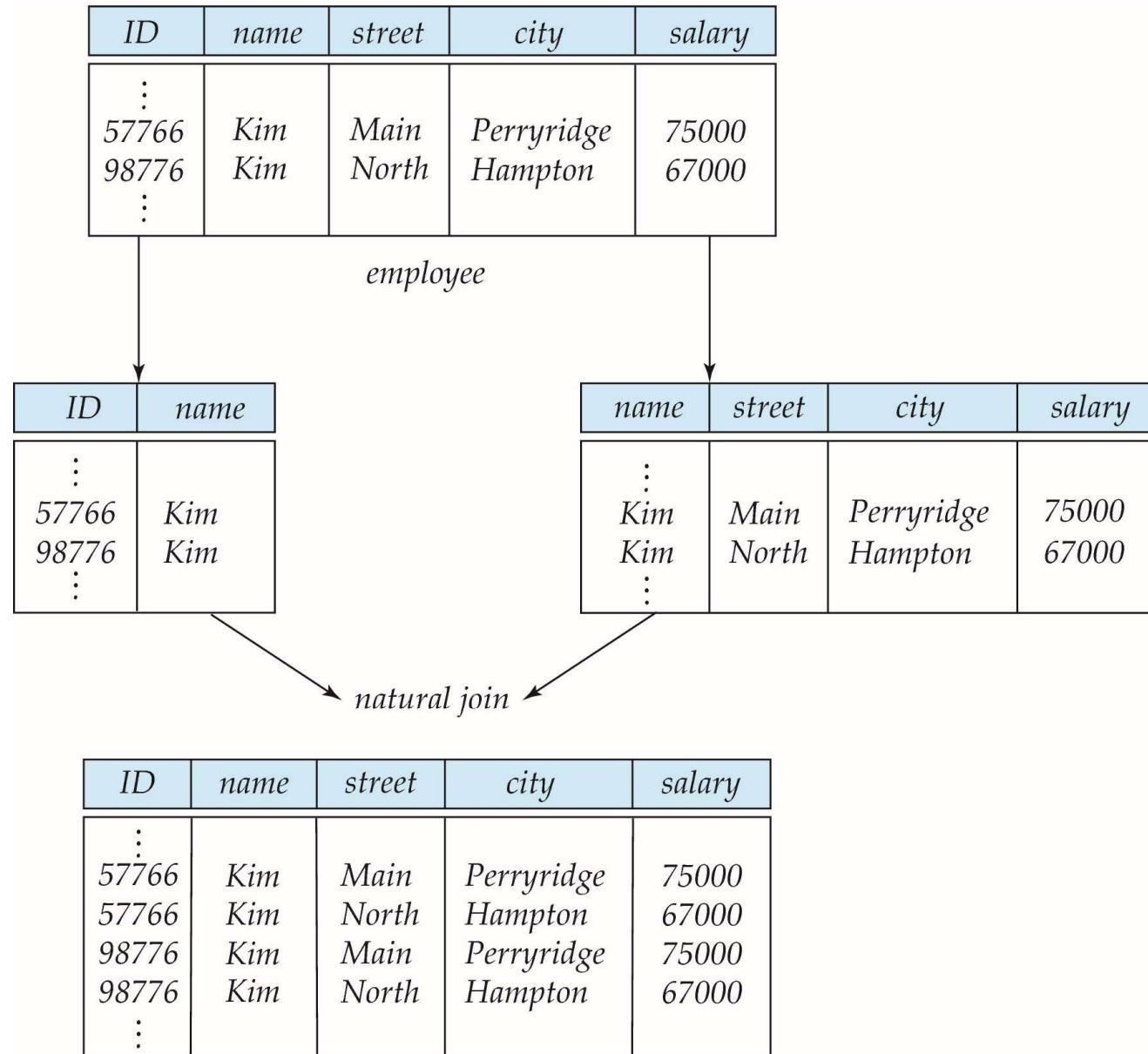
employee1 (ID, name)

employee2 (name, street, city, salary)

The problem arises when we have two employees with the same name

- The next slide shows how we lose information -- we cannot reconstruct the original `employee` relation -- and so, this is a **lossy decomposition**.

A Lossy Decomposition



Lossless Decomposition

- Let R be a relation schema and let R_1 and R_2 form a decomposition of R . That is $R = R_1 \cup R_2$
- We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with the two relation schemas $R_1 \cup R_2$
- Formally,

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

- And, conversely a decomposition is lossy if

$$r \subset \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$$

Example of Lossless Decomposition

- Decomposition of $R = (A, B, C)$

$$R_1 = (A, B) \quad R_2 = (B, C)$$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B	C
α	1	A
β	2	B

Normalization Theory

- Decide whether a particular relation R is in “good” form.
- In the case that a relation R is not in “good” form, decompose it into set of relations $\{R_1, R_2, \dots, R_n\}$ such that
 - Each relation is in good form
 - The decomposition is a lossless decomposition
- Our theory is based on:
 - Functional dependencies
 - Multivalued dependencies

First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units
 - Examples of non-atomic domains:
 - ▶ Set of names, composite attributes
 - ▶ Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
 - Example: Set of accounts stored with each customer, and set of owners stored with each account
 - We assume all relations are in first normal form.

First Normal Form (Cont'd)

- Atomicity is actually a property of how the elements of the domain are used.
 - Example: Strings would normally be considered indivisible
 - Suppose that students are given roll numbers which are strings of the form CS0012 or EE1127
 - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
 - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

Functional Dependencies

- There are usually a variety of constraints (rules) on the data in the real world.
- For example, some of the constraints that are expected to hold in a university database are:
 - Students and instructors are uniquely identified by their ID.
 - Each student and instructor has only one name.
 - Each instructor and student is (primarily) associated with only one department.
 - Each department has only one value for its budget, and only one associated building.

Functional Dependencies (Cont.)

- An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation;
 - A legal instance of a database is one where all the relation instances are legal instances
-
- Constraints on the set of legal relations.
 - Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
 - A functional dependency is a generalization of the notion of a key.

Functional Dependencies Definition

- Let R be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α , they also agree on the attributes β . That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider $r(A,B)$ with the following instance of r .

1	4
1	5
3	7

- On this instance, $B \rightarrow A$ hold; $A \rightarrow B$ does **NOT** hold,

Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F .
 - If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
 - etc.
- The set of **all** functional dependencies logically implied by F is the **closure** of F .
- We denote the *closure* of F by F^+ .

Keys and Functional Dependencies

- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
 - $K \rightarrow R$, and
 - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

in_dep (ID, name, salary, dept_name, building, budget).

We expect these functional dependencies to hold:

dept_name → building

ID → building

but would not expect the following to hold:

dept_name → salary

Use of Functional Dependencies

- We use functional dependencies to:
 - To test relations to see if they are legal under a given set of functional dependencies.
 - If a relation r is legal under a set F of functional dependencies, we say that r **satisfies** F .
 - To specify constraints on the set of legal relations
 - We say that F **holds on** R if all legal relations on R satisfy the set of functional dependencies F .
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
 - For example, a specific instance of *instructor* may, by chance, satisfy $name \rightarrow ID$.

Trivial Functional Dependencies

- A functional dependency is **trivial** if it is satisfied by all instances of a relation
- Example:
 - $ID, name \rightarrow ID$
 - $name \rightarrow name$
- In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F .
 - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F .
- We denote the *closure* of F by F^+ .
- F^+ is a superset of F .

Boyce-Codd Normal Form

A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for R

Example schema *not* in BCNF:

instr_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

because $\text{dept_name} \rightarrow \text{building}, \text{budget}$
holds on *instr_dept*, but *dept_name* is not a superkey

Decomposing a Schema into BCNF

- Suppose we have a schema R and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

We decompose R into:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

- In our example,

- $\alpha = \text{dept_name}$
- $\beta = \text{building}, \text{budget}$

and inst_dept is replaced by

- $(\alpha \cup \beta) = (\text{dept_name}, \text{building}, \text{budget})$
- $(R - (\beta - \alpha)) = (\text{ID}, \text{name}, \text{salary}, \text{dept_name})$

BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving*.
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form*.

Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

$$\alpha \rightarrow \beta \text{ in } F^+$$

at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
- α is a superkey for R
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R .

(**NOTE**: each attribute may be in a different candidate key)

- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in “good” form.
- In the case that a relation scheme R is not in “good” form, decompose it into a set of relation scheme $\{R_1, R_2, \dots, R_n\}$ such that
 - each relation scheme is in good form
 - the decomposition is a lossless-join decomposition
 - Preferably, the decomposition should be dependency preserving.

Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in “good” form.
- In the case that a relation scheme R is not in “good” form, decompose it into a set of relation scheme $\{R_1, R_2, \dots, R_n\}$ such that
 - each relation scheme is in good form
 - the decomposition is a lossless-join decomposition
 - Preferably, the decomposition should be dependency preserving.

How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation

inst_info (ID, child_name, phone)

- where an instructor may have more than one phone and can have multiple children

<i>ID</i>	<i>child_name</i>	<i>phone</i>
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	Willian	512-555-4321

inst_info

How good is BCNF? (Cont.)

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

(99999, David, 981-992-3443)
(99999, William, 981-992-3443)

How good is BCNF? (Cont.)

- Therefore, it is better to decompose *inst_info* into:

inst_child

<i>ID</i>	<i>child_name</i>
99999	David
99999	David
99999	William
99999	Willian

inst_phone

<i>ID</i>	<i>phone</i>
99999	512-555-1234
99999	512-555-4321
99999	512-555-1234
99999	512-555-4321

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF).

Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F .
 - For e.g.: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F .
- We denote the *closure* of F by F^+ .

Closure of a Set of Functional Dependencies

- We can find F^+ , the closure of F , by repeatedly applying **Armstrong's Axioms**:
 - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
 - if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ **(augmentation)**
 - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
 - **sound** (generate only functional dependencies that actually hold), and
 - **complete** (generate all functional dependencies that hold).

Example

- $R = (A, B, C, G, H, I)$
 $F = \{ A \rightarrow B$
 $A \rightarrow C$
 $CG \rightarrow H$
 $CG \rightarrow I$
 $B \rightarrow H\}$
- some members of F^+
 - $A \rightarrow H$
 - ▶ by transitivity from $A \rightarrow B$ and $B \rightarrow H$
 - $AG \rightarrow I$
 - ▶ by augmenting $A \rightarrow C$ with G , to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
 - $CG \rightarrow HI$
 - ▶ by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

Procedure for Computing F^+

- To compute the closure of a set of functional dependencies F :

$F^+ = F$

repeat

for each functional dependency f in F^+

 apply reflexivity and augmentation rules on f

 add the resulting functional dependencies to F^+

for each pair of functional dependencies f_1 and f_2 in F^+

if f_1 and f_2 can be combined using transitivity

then add the resulting functional dependency to F^+

until F^+ does not change any further

NOTE: We shall see an alternative procedure for this task later

Closure of Functional Dependencies (Cont.)

■ Additional rules:

- If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (**union**)
- If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (**decomposition**)
- If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \delta$ holds, then $\alpha\beta \rightarrow \delta$ holds (**pseudotransitivity**)

The above rules can be inferred from Armstrong's axioms.

Closure of Attribute Sets

- Given a set of attributes α , define the ***closure*** of α **under** F (denoted by α^+) as the set of attributes that are functionally determined by α under F
- Algorithm to compute α^+ , the closure of α under F

```
result :=  $\alpha$ ;  
while (changes to result) do  
  for each  $\beta \rightarrow \gamma$  in  $F$  do  
    begin  
      if  $\beta \subseteq result$  then result := result  $\cup$   $\gamma$   
    end
```

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- $(AG)^+$
 1. $result = AG$
 2. $result = ABCG$ ($A \rightarrow C$ and $A \rightarrow B$)
 3. $result = ABCGH$ ($CG \rightarrow H$ and $CG \subseteq AGBC$)
 4. $result = ABCGHI$ ($CG \rightarrow I$ and $CG \subseteq AGBCH$)
- Is AG a candidate key?
 1. Is AG a super key?
 1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$
 2. Is any subset of AG a superkey?
 1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
 2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$



Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
 - To test if α is a superkey, we compute α^+ and check if α^+ contains all attributes of R .
- Testing functional dependencies
 - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.
 - That is, we compute α^+ by using attribute closure, and then check if it contains β .
 - Is a simple and cheap test, and very useful
- Computing closure of F
 - For each $\gamma \subseteq R$, we find the closure γ^+ , and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
 - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$
 - Parts of a functional dependency may be redundant
 - ▶ E.g.: on RHS: $\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$ can be simplified to
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
 - ▶ E.g.: on LHS: $\{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$ can be simplified to
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
- Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F , having no redundant dependencies or redundant parts of dependencies

Ex 1

Compute the closure of the following set F of functional dependencies for relation schema $R = (A, B, C, D, E)$.

$$\begin{aligned} A &\rightarrow BC \\ CD &\rightarrow E \\ B &\rightarrow D \\ E &\rightarrow A \end{aligned}$$

- We can find F^+ , the closure of F , by repeatedly applying **Armstrong's Axioms:**

- if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
- if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$ **(augmentation)**
- if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**

- Additional rules:

- If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds **(union)**
- If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)**
- If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds **(pseudotransitivity)**

Ex 2 (1 cont.)

Suppose that we decompose the schema $R = (A, B, C, D, E)$ into

$$\begin{aligned}(A, B, C) \\ (A, D, E).\end{aligned}$$

Show that this decomposition is a lossless-join decomposition if the following set F of functional dependencies holds:

$$\begin{aligned}A \rightarrow BC \\ CD \rightarrow E \\ B \rightarrow D \\ E \rightarrow A\end{aligned}$$

Ex 2 (1 cont.)

Suppose that we decompose the schema $R = (A, B, C, D, E)$ into

$$\begin{aligned}(A, B, C) \\ (A, D, E).\end{aligned}$$

Show that this decomposition is a lossless-join decomposition if the following set F of functional dependencies holds:

$$\begin{aligned}A \rightarrow BC \\ CD \rightarrow E \\ B \rightarrow D \\ E \rightarrow A\end{aligned}$$

A decomposition $\{R_1, R_2\}$ is a lossless-join decomposition if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$. Let $R_1 = (A, B, C)$, $R_2 = (A, D, E)$, and $R_1 \cap R_2 = A$. Since A is a candidate key in $F^+ A \rightarrow ABC$. Therefore $R_1 \cap R_2 \rightarrow R_1$.

Intermediate SQL

CS3043 - Database Systems

Overview

- Joins
 - Outer joins
 - Inner joins
- Views
- Transactions
- Integrity Constraints
- Referential Integrity
- Built-in Data Types
- User Defined Data Types
- Domains
- Large Object Data Types
- Indexing

Joined Relations

- **Join operations** take two relations and return another relation as a result.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition).
- It also specifies the attributes that are present in the result of the join.
- The join operations are typically used as subquery expressions in the **from** clause.

Examples:

```
SELECT *
FROM course LEFT OUTER JOIN prereq
ON course.course_id = prereq.prereq_id
```

```
SELECT *
FROM course INNER JOIN prereq
USING (course_id)
```

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.

Join operations - Example

- course relation

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- prereq relation

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that,
 - information of CS-315 is missing on prereq
 - information of CS-347 is missing on course

LEFT OUTER JOIN

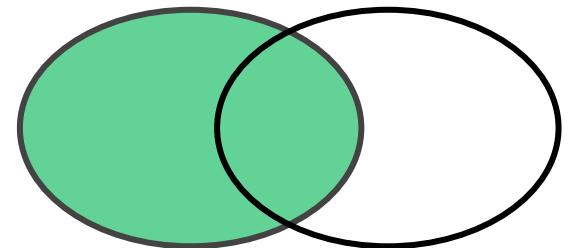
- Returns **all the rows from the left table** with matching rows from right table. If there is no match in the right table, those values will be null.

Examples:

```
SELECT *
FROM course LEFT OUTER JOIN prereq
ON course.course_id = prereq.course_id
```

```
SELECT *
FROM course LEFT OUTER JOIN prereq
USING (course_id)
```

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null

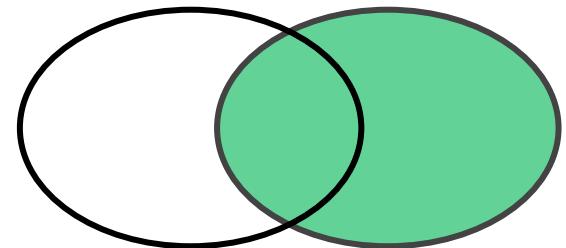


RIGHT OUTER JOIN

- Returns **all the rows from the right table** with matching rows from left table. If there is no match in the left table, those values will be null.

Examples:

```
SELECT *
FROM course RIGHT OUTER JOIN prereq
ON course.course_id = prereq.course_id
```



```
SELECT *
FROM course RIGHT OUTER JOIN prereq
USING (course_id)
```

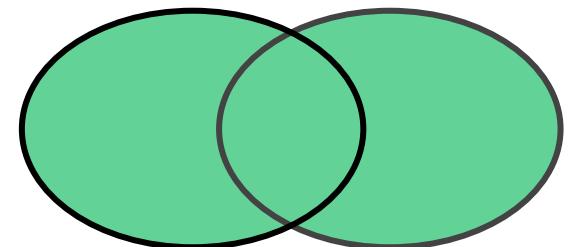
course_id	prereq_id	title	dept_name	credits
BIO-301	BIO-101	Genetics	Biology	4
CS-190	CS-101	Game Design	Comp. Sci.	4
CS-347	CS-101	null	null	null

FULL OUTER JOIN

- Returns **all the rows from the right table and all the rows from the left table.**
Missing values will be null.

Examples:

```
SELECT *
FROM course FULL OUTER JOIN prereq
ON course.course_id = prereq.course_id
```



- Not supported in MySQL

How do you emulate FULL OUTER JOIN in MySQL using LEFT and RIGHT JOIN?

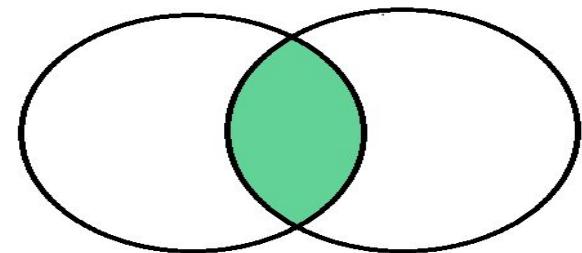
INNER JOIN

- Returns the rows when there is **a match in both tables.**

Examples:

```
SELECT *  
FROM course INNER JOIN prereq  
ON course.course_id = prereq.course_id
```

```
SELECT *  
FROM course INNER JOIN prereq  
USING (course_id)
```



Writing some queries

Instructor(ID, name, dept_name, salary)

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
.....

Teaches(ID, course_id, sec_id, semester, year)

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

Using JOINS only,

1. Find the names of instructors who do not teach any courses

```
select name  
from instructor left outer join teaches using(ID)  
where course_id is null;
```

2. Find the names of instructors who teach at least one course (no duplicates)

```
select distinct name  
from instructor inner join teaches using(ID);
```

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations) stored in the database.
- Consider a person who needs to know an instructor's name and department, but not the salary. This person should see a relation described in SQL by,

```
SELECT ID, name, dept_name  
FROM instructor
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

VIEW definition

- A view is defined using the **create view** statement which has the form

CREATE VIEW v AS <query expression>

v - name of the view

<query expression> - any legal SQL expression

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
 - Rather, a view definition causes the saving of an expression; **the expression is substituted into queries using the view.**

Example Views

- A view of instructors without their salary

```
create view faculty as
select ID, name, dept_name _name
from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
select dept_name, sum(salary)
from instructor
group by dept_name;
```

View dependencies

- One view may be used in the expression defining another view.
- A view relation v_2 is said to depend directly on a view relation v_1 , if v_1 is used in the expression defining v_2

$$v_1 \leftarrow v_2$$

- A view relation v_2 is said to depend on view relation v_1 if either v_2 depends directly on v_1 or there is a path of dependencies from v_2 to v_1

$$v_1 \leftarrow v_2 \quad \text{or} \quad v_1 \leftarrow \dots \leftarrow v_2$$

- A view relation v is said to be recursive if it depends on itself.

Views Defined Using Other Views

```
CREATE VIEW physics_fall_2009 AS
  SELECT course.course_id, sec_id, building, room_number
  FROM course, section
  WHERE course.course_id = section.course_id
        AND course.dept_name = 'Physics'
        AND section.semester = 'Fall'
        AND section.year = '2009'
```

```
CREATE VIEW physics_fall_2009_watson AS
  SELECT course_id, room_number
  FROM physics_fall_2009
  WHERE building = 'Watson'
```

View expansion

- A way to define the meaning of views defined in terms of other views.
- Let the view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

repeat

Find any view relation v_i in e_1

Replace the view relation v_i by the expression defining v_i

until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate.

Updating a View

- For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table.
- Most SQL implementations allow updates only on simple views
 - The **from** clause has only one database relation.
 - The **select** clause contains only attribute names of the relation and does not have any **expressions**, **aggregates**, or **distinct** specification.
 - Any attribute not listed in the **select** clause can be set to null
 - The query does not have a **group by** or **having** clause.

Example:

```
INSERT INTO faculty VALUES (30765, 'Green', 'Music');  
⇒ Query OK
```

```
INSERT INTO dept_total_salary VALUES ('Nuclear', 299000.00);  
⇒ ERROR : The target table dept_total_salary of the INSERT is not insertable-into
```

Materialized Views

- A logical view of the data driven by a SELECT query. But creates a physical relation/table containing all the tuples in the result of the query defining the view.
- If relations used in the query are updated, the materialized view (MV) result becomes out of date
 - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.
- **Performance is higher than Views**
 - Instead of running the expanded query against the database every time, MV acquire the results from a physical table
- Not supported in MySQL

Transactions

- Transaction is a **Unit of work** that is performed against a database.
- Have following four properties
 - **Atomicity** - the whole sequence of actions within a transaction is either fully executed or, in case of any exception, rolled back as if it never occurred.
 - **Consistency** - ensures that the database properly changes states upon a successfully committed transaction.
 - **Isolation** - enables transactions to occur independently and transparent of each other.
 - **Durability** - ensures that the result or effect of a committed transaction persists in case of a system failure.
- By default on most databases each SQL statement commits automatically
 - Can turn off auto commit for a session

START TRANSACTION;

<sequence of sql actions>

COMMIT;

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Example constraints:
 - A checking account must have a balance greater than \$10,000.00
 - A salary of a bank employee must be at least \$40 an hour
 - A customer must have a (non-null) phone number

Integrity Constraints

- **NOT NULL**

- declare certain attributes to be not null

Example:

name varchar(20) **NOT NULL**,
budget numeric(12,2) **NOT NULL**,

- **PRIMARY KEY**

- declare the primary key of the relation

Example:

PRIMARY KEY (ID)

- primary key cannot be null

- **UNIQUE**

- The unique specification states that the attributes A_1, A_2, \dots, A_m form a candidate key.
 - Candidate keys are permitted to be null (in contrast to primary keys).

Example:

UNIQUE (course_id, sec_id)

Integrity Constraints

- **CHECK (P)**
 - where P is a predicate

Example:

ensure that semester is one of fall, winter, spring or summer:

```
create table section (
    course_id varchar (8),
    sec_id_id varchar (8),
    semester varchar (6),
    year numeric (4,0),
    building varchar (15),
    room_number varchar (7),
    time_slot_id varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

Integrity Constraints - Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
 - Example: If “Biology” is a department name appearing in one of the tuples in the *course* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

```
create table course (
    course_id char(5) primary key,
    title varchar(20),
    dept_name varchar(20) references department
)
```

Cascading Actions in Referential Integrity

- A foreign key with cascade delete/update in a child table means that **if a record in the parent table is deleted/updated, then the corresponding records in the child table will automatically be deleted/updated.**

```
create table course (
    ...
    dept_name varchar(20),
    foreign key (dept_name) references department
        on delete cascade
        on update cascade,
    ...
)
```

- alternative actions to cascade: **set null, set default**

Integrity Constraint Violation during Transactions

- Example:

```
create table person (
    ID char(10),
    name char(40),
    mother char(10),
    father char(10),
    primary key ID,
    foreign key father references person,
    foreign key mother references person)
```

- How to insert a tuple without causing constraint violation ?
 - insert father and mother of a person before inserting person
 - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
 - *OR defer constraint checking and use transactions*

Built-in Data Types in SQL

- **date:** Dates, containing a year, month and date
Example: **date** ‘2005-7-27’
- **time:** Time of day in hours minutes and seconds , in hours, minutes and seconds.
Example: **time** ‘09:00:30’
time ‘09:00:30.75’
- **timestamp:** date plus time of day
Example: **timestamp** ‘2005-7-27 09:00:30.75’
- **interval:** period of time
Example: **interval** ‘1’ day
 - Subtracting a date/time/timestamp value from another gives an interval value
 - Interval values can be added to date/time/timestamp values

User Defined Types

- **create type** construct in SQL creates user-defined type

create type Dollars as numeric (12,2) final

Using the custom data type:

```
create table department(  
    dept_name varchar (20),  
    building varchar (15),  
    budget Dollars  
)
```

Specify **FINAL** if no subtypes can be created for this type. (default)

Specify **NOT FINAL** if further subtypes can be defined for this type.

Domains

- **create domain** construct in SQL-92 creates user-defined domain types
create domain *person_name* **char(20)** **not null**
- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.

```
create domain degree_level varchar(10)
constraint degree_level_test
check (value in ('Bachelors', 'Masters', 'Doctorate'));
```

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
 - **blob**: binary large object - object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
 - **clob**: character large object - object is a large collection of character data
 - When a query returns a large object, a pointer is returned rather than the large object itself.

Indexing

- Indices are data structures used to speed up access to records with specified values for index attributes. (Indices are used to find rows with specific column values quickly.)

```
CREATE INDEX student_dept_name_index ON student(dept_name);
```

Example scenario:

```
select *  
from student  
where dept_name = "Physics"
```

can be executed faster by using the index to find the required record without looking at all records of *student*

- The users cannot see the indexes, they are just used to speed up searches/queries.
- Primary key is indexed by default in many implementations (MySQL)
- More on indices in Chapter 11

Indexing

- Index is a data structure which is implemented using
 - B-Trees
 - Hash tables
 - R-Trees
- Advantages of using an index
 - Faster search time by not having to scan entire table
 - Specially when running SELECT queries or JOINS
- Disadvantages
 - Takes up space - larger the number of rows, larger the index size
 - Need to update the index, when rows are added, deleted or updated
- Best practices
 - Indices should only be used if the data in the indexed column is queried frequently
 - Index columns that are being used as foreign keys in other tables
 - Add additional indices based on performance requirements
 - Do not index every column

Sample entry in an index
("BIO-319", 0x562189)

Thank you!

Truncate and Delete

- Truncate
 - classified as a DDL statement
 - empties the table completely
 - drop and re-create the table, which is much faster than deleting rows one by one
 - does not invoke ON DELETE triggers
 - cannot be rolled back on MySQL
- Delete
 - classified as a DML statement
 - deletes row by row (can specify conditions with where clause)
 - operations are logged individually
 - invokes on delete triggers
 - can be rolled back

Chapter 8: Relational Database Design

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Chapter 8: Relational Database Design

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
- Functional Dependency Theory
- Algorithms for Functional Dependencies
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data

Combine Schemas?

- Suppose we combine *instructor* and *department* into *inst_dept*
 - (*No connection to relationship set inst_dept*)
- Result is possible repetition of information

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

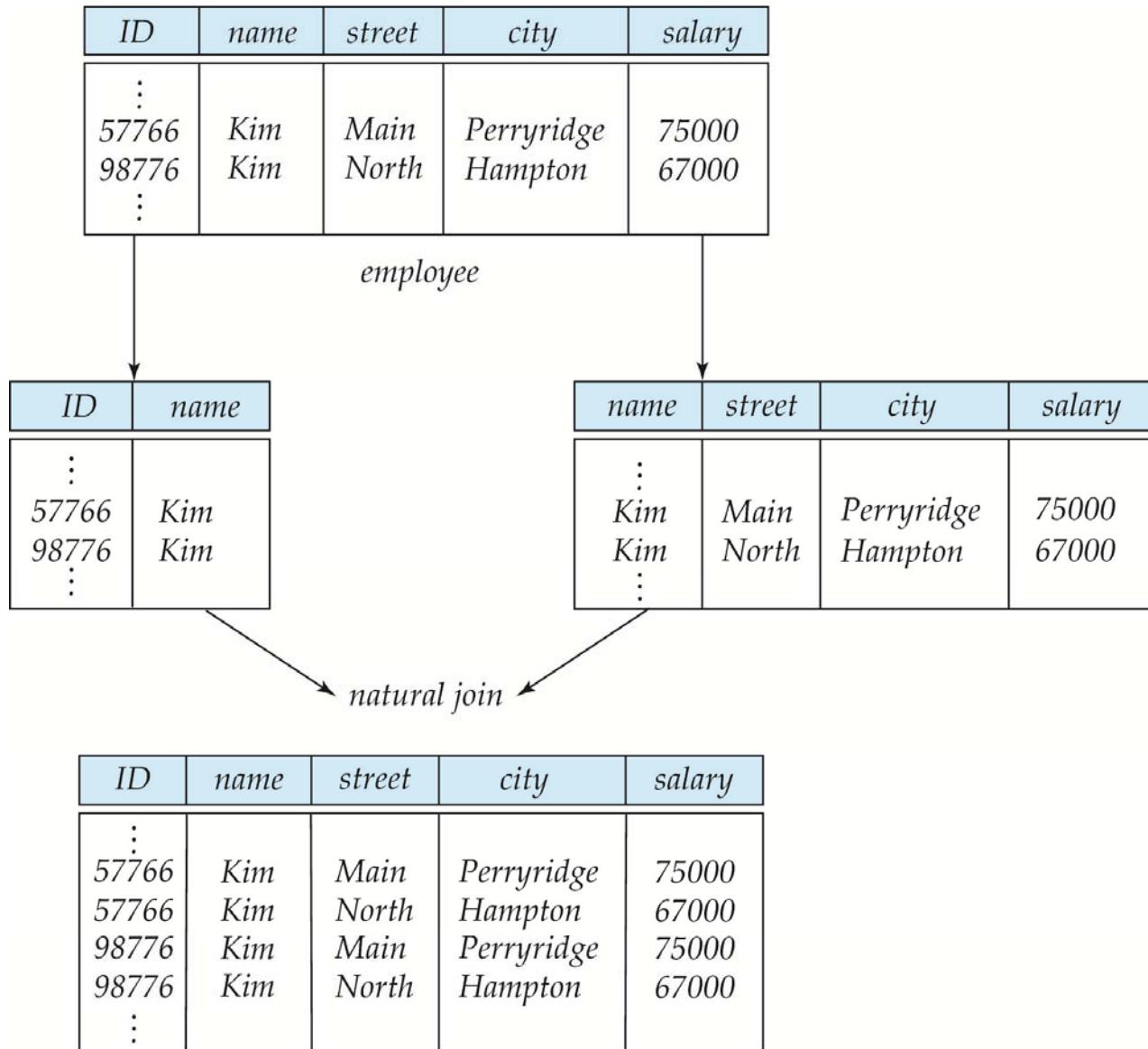
A Combined Schema Without Repetition

- Consider combining relations
 - $\text{sec_class}(\text{sec_id}, \text{building}, \text{room_number})$ and
 - $\text{section}(\text{course_id}, \text{sec_id}, \text{semester}, \text{year})$
- into one relation
 - $\text{section}(\text{course_id}, \text{sec_id}, \text{semester}, \text{year}, \text{building}, \text{room_number})$
- No repetition in this case

What About Smaller Schemas?

- Suppose we had started with *inst_dept*. How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule “if there were a schema (*dept_name*, *building*, *budget*), then *dept_name* would be a candidate key”
- Denote as a **functional dependency**:
$$\text{dept_name} \rightarrow \text{building}, \text{budget}$$
- In *inst_dept*, because *dept_name* is not a candidate key, the building and budget of a department may have to be repeated.
 - This indicates the need to decompose *inst_dept*
- Not all decompositions are good. Suppose we decompose *employee*(*ID*, *name*, *street*, *city*, *salary*) into
 - employee1* (*ID*, *name*)
 - employee2* (*name*, *street*, *city*, *salary*)
- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

A Lossy Decomposition



Example of Lossless-Join Decomposition

- **Lossless join decomposition**

- Decomposition of $R = (A, B, C)$

$$R_1 = (A, B) \quad R_2 = (B, C)$$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B	C
α	1	A
β	2	B

First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units
 - Examples of non-atomic domains:
 - ▶ Set of names, composite attributes
 - ▶ Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
 - Example: Set of accounts stored with each customer, and set of owners stored with each account
 - We assume all relations are in first normal form.

First Normal Form (Cont'd)

- Atomicity is actually a property of how the elements of the domain are used.
 - Example: Strings would normally be considered indivisible
 - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
 - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
 - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

Goal — Devise a Theory for the Following

- Decide whether a particular relation R is in “good” form.
- In the case that a relation R is not in “good” form, decompose it into a set of relations $\{R_1, R_2, \dots, R_n\}$ such that
 - each relation is in good form
 - the decomposition is a lossless-join decomposition
- Our theory is based on:
 - functional dependencies
 - multivalued dependencies

Functional Dependencies

- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key*.

Functional Dependencies (Cont.)

- Let R be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α , they also agree on the attributes β . That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider $r(A,B)$ with the following instance of r .

1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

Functional Dependencies (Cont.)

- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
 - $K \rightarrow R$, and
 - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

inst_dept (ID, name, salary, dept_name, building, budget).

We expect these functional dependencies to hold:

$$\textit{dept_name} \rightarrow \textit{building}$$

and $\textit{ID} \rightarrow \textit{building}$

but would not expect the following to hold:

$$\textit{dept_name} \rightarrow \textit{salary}$$

Use of Functional Dependencies

- We use functional dependencies to:
 - test relations to see if they are legal under a given set of functional dependencies.
 - ▶ If a relation r is legal under a set F of functional dependencies, we say that r **satisfies** F .
 - specify constraints on the set of legal relations
 - ▶ We say that F **holds on** R if all legal relations on R satisfy the set of functional dependencies F .
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
 - For example, a specific instance of *instructor* may, by chance, satisfy $name \rightarrow ID$.

Functional Dependencies (Cont.)

- A functional dependency is **trivial** if it is satisfied by all instances of a relation
 - Example:
 - ▶ $ID, name \rightarrow ID$
 - ▶ $name \rightarrow name$
 - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F .
 - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F .
- We denote the *closure* of F by F^+ .
- F^+ is a superset of F .

Boyce-Codd Normal Form

A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for R

Example schema *not* in BCNF:

instr_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

because $\text{dept_name} \rightarrow \text{building}, \text{budget}$
holds on *instr_dept*, but *dept_name* is not a superkey

Decomposing a Schema into BCNF

- Suppose we have a schema R and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

We decompose R into:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

- In our example,

- $\alpha = \text{dept_name}$
- $\beta = \text{building, budget}$

and inst_dept is replaced by

- $(\alpha \cup \beta) = (\text{dept_name, building, budget})$
- $(R - (\beta - \alpha)) = (\text{ID, name, salary, dept_name})$

BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving*.
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form*.

Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

$$\alpha \rightarrow \beta \text{ in } F^+$$

at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
- α is a superkey for R
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R .

(**NOTE**: each attribute may be in a different candidate key)

- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in “good” form.
- In the case that a relation scheme R is not in “good” form, decompose it into a set of relation scheme $\{R_1, R_2, \dots, R_n\}$ such that
 - each relation scheme is in good form
 - the decomposition is a lossless-join decomposition
 - Preferably, the decomposition should be dependency preserving.

How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation

inst_info (ID, child_name, phone)

- where an instructor may have more than one phone and can have multiple children

<i>ID</i>	<i>child_name</i>	<i>phone</i>
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	Willian	512-555-4321

inst_info

How good is BCNF? (Cont.)

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

(99999, David, 981-992-3443)
(99999, William, 981-992-3443)

How good is BCNF? (Cont.)

- Therefore, it is better to decompose *inst_info* into:

inst_child

<i>ID</i>	<i>child_name</i>
99999	David
99999	David
99999	William
99999	Willian

inst_phone

<i>ID</i>	<i>phone</i>
99999	512-555-1234
99999	512-555-4321
99999	512-555-1234
99999	512-555-4321

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF).

Functional-Dependency Theory

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.
- We then develop algorithms to generate lossless decompositions into BCNF and 3NF
- We then develop algorithms to test if a decomposition is dependency-preserving

Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F .
 - For e.g.: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F .
- We denote the *closure* of F by F^+ .

Closure of a Set of Functional Dependencies

- We can find F^+ , the closure of F , by repeatedly applying **Armstrong's Axioms**:
 - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
 - if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ **(augmentation)**
 - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
 - **sound** (generate only functional dependencies that actually hold), and
 - **complete** (generate all functional dependencies that hold).

Example

- $R = (A, B, C, G, H, I)$

$$F = \{ A \rightarrow B$$

$$A \rightarrow C$$

$$CG \rightarrow H$$

$$CG \rightarrow I$$

$$B \rightarrow H\}$$

- some members of F^+

- $A \rightarrow H$

- ▶ by transitivity from $A \rightarrow B$ and $B \rightarrow H$

- $AG \rightarrow I$

- ▶ by augmenting $A \rightarrow C$ with G , to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$

- $CG \rightarrow HI$

- ▶ by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

Procedure for Computing F^+

- To compute the closure of a set of functional dependencies F :

$F^+ = F$

repeat

for each functional dependency f in F^+

 apply reflexivity and augmentation rules on f

 add the resulting functional dependencies to F^+

for each pair of functional dependencies f_1 and f_2 in F^+

if f_1 and f_2 can be combined using transitivity

then add the resulting functional dependency to F^+

until F^+ does not change any further

NOTE: We shall see an alternative procedure for this task later

Closure of Functional Dependencies (Cont.)

■ Additional rules:

- If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (**union**)
- If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (**decomposition**)
- If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \delta$ holds, then $\alpha\beta \rightarrow \delta$ holds (**pseudotransitivity**)

The above rules can be inferred from Armstrong's axioms.

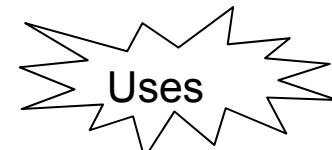
Closure of Attribute Sets

- Given a set of attributes α , define the ***closure*** of α **under F** (denoted by α^+) as the set of attributes that are functionally determined by α under F
- Algorithm to compute α^+ , the closure of α under F

```
result :=  $\alpha$ ;  
while (changes to result) do  
  for each  $\beta \rightarrow \gamma$  in  $F$  do  
    begin  
      if  $\beta \subseteq result$  then result := result  $\cup$   $\gamma$   
    end
```

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- $(AG)^+$
 1. $result = AG$
 2. $result = ABCG$ ($A \rightarrow C$ and $A \rightarrow B$)
 3. $result = ABCGH$ ($CG \rightarrow H$ and $CG \subseteq AGBC$)
 4. $result = ABCGHI$ ($CG \rightarrow I$ and $CG \subseteq AGBCH$)
- Is AG a candidate key?
 1. Is AG a super key?
 1. Does $AG \rightarrow R? \Rightarrow Is (AG)^+ \supseteq R$
 2. Is any subset of AG a superkey?
 1. Does $A \rightarrow R? \Rightarrow Is (A)^+ \supseteq R$
 2. Does $G \rightarrow R? \Rightarrow Is (G)^+ \supseteq R$



Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
 - To test if α is a superkey, we compute α^+ , and check if α^+ contains all attributes of R .
- Testing functional dependencies
 - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.
 - That is, we compute α^+ by using attribute closure, and then check if it contains β .
 - Is a simple and cheap test, and very useful
- Computing closure of F
 - For each $\gamma \subseteq R$, we find the closure γ^+ , and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
 - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$
 - Parts of a functional dependency may be redundant
 - ▶ E.g.: on RHS: $\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$ can be simplified to
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
 - ▶ E.g.: on LHS: $\{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$ can be simplified to
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
- Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F , having no redundant dependencies or redundant parts of dependencies

Ex 1

Compute the closure of the following set F of functional dependencies for relation schema $R = (A, B, C, D, E)$.

$$A \rightarrow BC$$

$$CD \rightarrow E$$

$$B \rightarrow D$$

$$E \rightarrow A$$

- We can find F^+ , the closure of F , by repeatedly applying **Armstrong's Axioms:**

- if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
- if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$ **(augmentation)**
- if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**

- Additional rules:
 - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds **(union)**
 - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)**
 - If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds **(pseudotransitivity)**

Ex 2 (1 cont.)

Suppose that we decompose the schema $R = (A, B, C, D, E)$ into

$$\begin{aligned}(A, B, C) \\ (A, D, E).\end{aligned}$$

Show that this decomposition is a lossless-join decomposition if the following set F of functional dependencies holds:

$$\begin{aligned}A \rightarrow BC \\ CD \rightarrow E \\ B \rightarrow D \\ E \rightarrow A\end{aligned}$$

A decomposition $\{R_1, R_2\}$ is a lossless-join decomposition if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$. Let $R_1 = (A, B, C)$, $R_2 = (A, D, E)$, and $R_1 \cap R_2 = A$. Since A is a candidate key in $F^+ A \rightarrow ABC$. Therefore $R_1 \cap R_2 \rightarrow R_1$.

Extraneous Attributes

Attribute is extraneous if we can remove it without changing the closure of functional dependencies.

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .
 - Attribute A is **extraneous** in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
 - Attribute A is **extraneous** in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F .
- Note: implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one
- Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
 - B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (i.e. the result of dropping B from $AB \rightarrow C$).
- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
 - C is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting C

Testing if an Attribute is Extraneous

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .
- To test if attribute $A \in \alpha$ is extraneous in α
 1. compute $(\{\alpha\} - A)^+$ using the dependencies in F
 2. check that $(\{\alpha\} - A)^+$ contains β ; if it does, A is extraneous in α
- To test if attribute $A \in \beta$ is extraneous in β
 1. compute α^+ using only the dependencies in
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\},$$
 2. check that α^+ contains A ; if it does, A is extraneous in β

Canonical Cover

- A **canonical cover** for F is a set of dependencies F_c such that
 - F logically implies all dependencies in F_c , and
 - F_c logically implies all dependencies in F , and
 - No functional dependency in F_c contains an extraneous attribute, and
 - Each left side of functional dependency in F_c is unique.
- To compute a canonical cover for F :
repeat
 - Use the union rule to replace any dependencies in F
 $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$
 - Find a functional dependency $\alpha \rightarrow \beta$ with an
 extraneous attribute either in α or in β
 /* Note: test for extraneous attributes done using F_c , not F^* */
 - If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$**until** F does not change
- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

Computing a Canonical Cover

- $R = (A, B, C)$
 $F = \{A \rightarrow BC$
 $B \rightarrow C$
 $A \rightarrow B$
 $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
 - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
 - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
 - ▶ Yes: in fact, $B \rightarrow C$ is already present!
 - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
 - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
 - ▶ Yes: using transitivity on $A \rightarrow B$ and $B \rightarrow C$.
 - Can use attribute closure of A in more complex cases
- The canonical cover is:
 - $A \rightarrow B$
 - $B \rightarrow C$

Consider the following set F of functional dependencies on the relation schema $r(A, B, C, D, E, F)$:

$$A \rightarrow BCD$$

$$BC \rightarrow DE$$

$$B \rightarrow D$$

$$D \rightarrow A$$

- a. Compute B^+ .
- b. Prove (using Armstrong's axioms) that AF is a superkey.
- c. Compute a canonical cover for the above set of functional dependencies F ; give each step of your derivation with an explanation.

■ **Armstrong's Axioms:**

- if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
- if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ **(augmentation)**
- if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**

Algorithm to compute α^+ , the closure of α under F

```

result :=  $\alpha$ ;
while (changes to result) do
  for each  $\beta \rightarrow \gamma$  in  $F$  do
    begin
      if  $\beta \subseteq result$  then  $result := result \cup \gamma$ 
    end
  
```

To compute a canonical cover for F :

repeat

 Use the union rule to replace any dependencies in F

$\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$

 Find a functional dependency $\alpha \rightarrow \beta$ with an
 extraneous attribute either in α or in β

 /* Note: test for extraneous attributes done using F_c , not F */

 If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$

until F does not change

Algorithm to compute α^+ , the closure of α under F

```
result :=  $\alpha$ ;  
while (changes to result) do  
    for each  $\beta \rightarrow \gamma$  in  $F$  do  
        begin  
            if  $\beta \subseteq result$  then  $result := result \cup \gamma$   
        end
```

■ Armstrong's Axioms:

- if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (reflexivity)
- if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ (augmentation)
- if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (transitivity)

■ Additional rules:

- If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta \gamma$ holds (union)
- If $\alpha \rightarrow \beta \gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (decomposition)
- If $\alpha \rightarrow \beta$ holds and $\gamma \beta \rightarrow \delta$ holds, then $\alpha \gamma \rightarrow \delta$ holds (pseudotransitivity)

To compute a canonical cover for F :

repeat

 Use the union rule to replace any dependencies in F

$\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$

 Find a functional dependency $\alpha \rightarrow \beta$ with an
 extraneous attribute either in α or in β

 /* Note: test for extraneous attributes done using F_c , not F^* /

 If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$

until F does not change

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+ :
 - $R_1 \cap R_2 \rightarrow R_1$
 - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

Example

- $R = (A, B, C)$
 $F = \{A \rightarrow B, B \rightarrow C\}$
 - Can be decomposed in two different ways

- $R_1 = (A, B), R_2 = (B, C)$
 - Lossless-join decomposition:

$$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$

- $R_1 = (A, B), R_2 = (A, C)$
 - Lossless-join decomposition:

$$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

- Not dependency preserving
(cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i .
 - ▶ A decomposition is **dependency preserving**, if
$$(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$$
 - ▶ If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Testing for Dependency Preservation

- To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of R into R_1, R_2, \dots, R_n we apply the following test (with attribute closure done with respect to F)
 - $result = \alpha$
while (changes to $result$) do
 for each R_i in the decomposition
 $t = (result \cap R_i)^+ \cap R_i$
 $result = result \cup t$
 - If $result$ contains all attributes in β , then the functional dependency $\alpha \rightarrow \beta$ is preserved.
- We apply the test on all dependencies in F to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute F^+ and $(F_1 \cup F_2 \cup \dots \cup F_n)^+$

Example

- $R = (A, B, C)$
 $F = \{A \rightarrow B$
 $\quad B \rightarrow C\}$
Key = {A}
- R is not in BCNF
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
 - R_1 and R_2 in BCNF
 - Lossless-join decomposition
 - Dependency preserving

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
 1. compute α^+ (the attribute closure of α), and
 2. verify that it includes all attributes of R , that is, it is a superkey of R .
- **Simplified test:** To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than checking all dependencies in F^+ .
 - If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.
- However, **simplified test using only F is incorrect when testing a relation in a decomposition of R**
 - Consider $R = (A, B, C, D, E)$, with $F = \{ A \rightarrow B, BC \rightarrow D \}$
 - ▶ Decompose R into $R_1 = (A, B)$ and $R_2 = (A, C, D, E)$
 - ▶ Neither of the dependencies in F contain only attributes from (A, C, D, E) so we might be misled into thinking R_2 satisfies BCNF.
 - ▶ In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.

Testing Decomposition for BCNF

- To check if a relation R_i in a decomposition of R is in BCNF,
 - Either test R_i for BCNF with respect to the **restriction** of F to R_i (that is, all FDs in F^+ that contain only attributes from R_i)
 - or use the original set of dependencies F that hold on R , but with the following test:
 - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i .
- ▶ If the condition is violated by some $\alpha \rightarrow \beta$ in F , the dependency
$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$
can be shown to hold on R_i , and R_i violates BCNF.
- ▶ We use above dependency to decompose R_i

BCNF Decomposition Algorithm

```
result := {R};  
done := false;  
compute  $F^+$ ;  
while (not done) do  
  if (there is a schema  $R_i$  in result that is not in BCNF)  
    then begin  
      let  $\alpha \rightarrow \beta$  be a nontrivial functional dependency that  
      holds on  $R_i$  such that  $\alpha \rightarrow R_i$  is not in  $F^+$ ,  
      and  $\alpha \cap \beta = \emptyset$ ;  
      result := (result -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );  
    end  
  else done := true;
```

Note: each R_i is in BCNF, and decomposition is lossless-join.

Example of BCNF Decomposition

- $R = (A, B, C)$
 $F = \{A \rightarrow B$
 $\quad B \rightarrow C\}$
Key = {A}
- R is not in BCNF ($B \rightarrow C$ but B is not superkey)
- Decomposition
 - $R_1 = (B, C)$
 - $R_2 = (A, B)$

Example of BCNF Decomposition

- *class (course_id, title, dept_name, credits, sec_id, semester, year, building, room_number, capacity, time_slot_id)*
- Functional dependencies:
 - $course_id \rightarrow title, dept_name, credits$
 - $building, room_number \rightarrow capacity$
 - $course_id, sec_id, semester, year \rightarrow building, room_number, time_slot_id$
- A candidate key $\{course_id, sec_id, semester, year\}$.
- BCNF Decomposition:
 - $course_id \rightarrow title, dept_name, credits$ holds
 - ▶ but $course_id$ is not a superkey.
 - We replace *class* by:
 - ▶ *course(course_id, title, dept_name, credits)*
 - ▶ *class-1 (course_id, sec_id, semester, year, building, room_number, capacity, time_slot_id)*

BCNF Decomposition (Cont.)

- *course* is in BCNF
 - How do we know this?
- *building, room_number*→*capacity* holds on *class-1*
 - but $\{building, room_number\}$ is not a superkey for *class-1*.
 - We replace *class-1* by:
 - ▶ *classroom* (*building, room_number, capacity*)
 - ▶ *section* (*course_id, sec_id, semester, year, building, room_number, time_slot_id*)
- *classroom* and *section* are in BCNF.

BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving

- $R = (J, K, L)$

$$F = \{JK \rightarrow L \\ L \rightarrow K\}$$

Two candidate keys = JK and JL

- R is not in BCNF

- Any decomposition of R will fail to preserve

$$JK \rightarrow L$$

This implies that testing for $JK \rightarrow L$ requires a join

Third Normal Form: Motivation

- There are some situations where
 - BCNF is not dependency preserving, and
 - efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form (3NF)
 - Allows some redundancy (with resultant problems; we will see examples later)
 - But functional dependencies can be checked on individual relations without computing a join.
 - There is always a lossless-join, dependency-preserving decomposition into 3NF.

3NF Example

- Relation *dept_advisor*:

- $\text{dept_advisor}(s_ID, i_ID, \text{dept_name})$
 $F = \{s_ID, \text{dept_name} \rightarrow i_ID, i_ID \rightarrow \text{dept_name}\}$
- Two candidate keys: s_ID , dept_name , and i_ID, s_ID
- R is in 3NF
 - ▶ $s_ID, \text{dept_name} \rightarrow i_ID$
 - $s_ID, \text{dept_name}$ is a superkey
 - ▶ $i_ID \rightarrow \text{dept_name}$
 - dept_name is contained in a candidate key

Redundancy in 3NF

- There is some redundancy in this schema
- Example of problems due to redundancy in 3NF

- $R = (J, K, L)$

- $F = \{JK \rightarrow L, L \rightarrow K\}$

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
<i>null</i>	l_2	k_2

- repetition of information (e.g., the relationship l_1, k_1)
 - $(i_ID, dept_name)$
- need to use null values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J).
 - $(i_ID, dept_name)$ if there is no separate relation mapping instructors to departments

Testing for 3NF

- Optimization: Need to check only FDs in F , need not check all FDs in F^+ .
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of R
 - this test is rather more expensive, since it involve finding candidate keys
 - testing for 3NF has been shown to be NP-hard
 - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

3NF Decomposition Algorithm

Let F_c be a canonical cover for F ;

$i := 0$;

for each functional dependency $\alpha \rightarrow \beta$ in F_c **do**

if none of the schemas R_j , $1 \leq j \leq i$ contains $\alpha \beta$

then begin

$i := i + 1$;

$R_i := \alpha \beta$

end

if none of the schemas R_j , $1 \leq j \leq i$ contains a candidate key for R

then begin

$i := i + 1$;

$R_i :=$ any candidate key for R ;

end

/* Optionally, remove redundant relations */

repeat

if any schema R_j is contained in another schema R_k

then /* delete R_j */

$R_j = R;;$

$i = i - 1$;

return (R_1, R_2, \dots, R_i)

3NF Decomposition Algorithm

```
let  $F_c$  be a canonical cover for  $F$ ;  
 $i := 0$ ;  
for each functional dependency  $\alpha \rightarrow \beta$  in  $F_c$   
     $i := i + 1$ ;  
     $R_i := \alpha \beta$ ;  
if none of the schemas  $R_j$ ,  $j = 1, 2, \dots, i$  contains a candidate key for  $R$   
then  
     $i := i + 1$ ;  
     $R_i :=$  any candidate key for  $R$ ;  
/* Optionally, remove redundant relations */  
repeat  
    if any schema  $R_j$  is contained in another schema  $R_k$   
    then  
        /* Delete  $R_j$  */  
         $R_j := R_i$ ;  
         $i := i - 1$ ;  
    until no more  $R_j$ s can be deleted  
return  $(R_1, R_2, \dots, R_i)$ 
```

3NF Decomposition Algorithm (Cont.)

- Above algorithm ensures:
 - each relation schema R_i is in 3NF
 - decomposition is dependency preserving and lossless-join

3NF Decomposition: An Example

- Relation schema:

$\text{cust_banker_branch} = (\underline{\text{customer_id}}, \underline{\text{employee_id}}, \text{branch_name}, \text{type})$

- The functional dependencies for this relation schema are:

1. $\text{customer_id}, \text{employee_id} \rightarrow \text{branch_name}, \text{type}$
2. $\text{employee_id} \rightarrow \text{branch_name}$
3. $\text{customer_id}, \text{branch_name} \rightarrow \text{employee_id}$

- We first compute a canonical cover

- branch_name is extraneous in the r.h.s. of the 1st dependency
- No other attribute is extraneous, so we get $F_C =$

$\text{customer_id}, \text{employee_id} \rightarrow \text{type}$
 $\text{employee_id} \rightarrow \text{branch_name}$
 $\text{customer_id}, \text{branch_name} \rightarrow \text{employee_id}$

3NF Decomposition Example (Cont.)

- The **for** loop generates following 3NF schema:

$(customer_id, employee_id, type)$

$(\underline{employee_id}, branch_name)$

$(customer_id, branch_name, employee_id)$

- Observe that $(customer_id, employee_id, type)$ contains a candidate key of the original schema, so no further relation schema needs be added
- At end of for loop, detect and delete schemas, such as $(\underline{employee_id}, branch_name)$, which are subsets of other schemas
 - result will not depend on the order in which FDs are considered
- The resultant simplified 3NF schema is:

$(customer_id, employee_id, type)$

$(customer_id, branch_name, employee_id)$

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
 - the decomposition is lossless
 - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
 - the decomposition is lossless
 - it may not be possible to preserve dependencies.

Design Goals

- Goal for a relational database design is:
 1. BCNF.
 2. Lossless join.
 3. Dependency preservation.
- If we cannot achieve this, we accept one of
 - Lack of dependency preservation
 - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.

Can specify FDs using assertions, but they are expensive to test, (and currently not supported by any of the widely used databases!)
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Overall Database Design Process

- We have assumed schema R is given
 - R could have been generated when converting E-R diagram to a set of tables.
 - R could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
 - Normalization breaks R into smaller relations.
 - R could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
 - Example: an *employee* entity with attributes *department_name* and *building*, and a functional dependency $\text{department_name} \rightarrow \text{building}$
 - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary

Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *prereqs* along with *course_id*, and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
 - faster lookup
 - extra space and extra execution time for updates
 - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as

course	<i>prereq</i>
--------	---------------

 - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:
 - Instead of *earnings* (*company_id*, *year*, *amount*), use
 - *earnings_2004*, *earnings_2005*, *earnings_2006*, etc., all on the schema (*company_id*, *earnings*).
 - ▶ Above are in BCNF, but make querying across years difficult and needs new table each year
 - *company_year* (*company_id*, *earnings_2004*, *earnings_2005*, *earnings_2006*)
 - ▶ Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
 - ▶ Is an example of a **crosstab**, where values for one attribute become column names
 - ▶ Used in spreadsheets, and in data analysis tools

Modeling Temporal Data

- **Temporal data** have an association time interval during which the data are *valid*.
- A **snapshot** is the value of the data at a particular point in time
- Several proposals to extend ER model by adding valid time to
 - attributes, e.g., address of an instructor at different points in time
 - entities, e.g., time duration when a student entity exists
 - relationships, e.g., time during which an instructor was associated with a student as an advisor.
- But no accepted standard
- Adding a temporal component results in functional dependencies like
$$ID \rightarrow street, city$$
not to hold, because the address varies over time
- A **temporal functional dependency** $X^T \rightarrow Y$ holds on schema R if the functional dependency $X \rightarrow Y$ holds on all snapshots for all legal instances $r(R)$.

Modeling Temporal Data (Cont.)

- In practice, database designers may add start and end time attributes to relations
 - E.g., $\text{course}(\text{course_id}, \text{course_title})$ is replaced by $\text{course}(\text{course_id}, \text{course_title}, \text{start}, \text{end})$
 - ▶ Constraint: no two tuples can have overlapping valid times
 - Hard to enforce efficiently
- Foreign key references may be to current version of data, or to data at a point in time
 - E.g., student transcript should refer to course information at the time the course was taken

End of Chapter

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Proof of Correctness of 3NF Decomposition Algorithm

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Correctness of 3NF Decomposition Algorithm

- 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in F_c)
- Decomposition is lossless
 - A candidate key (C) is in one of the relations R_i in decomposition
 - Closure of candidate key under F_c must contain all attributes in R .
 - Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in R_i

Correctness of 3NF Decomposition Algorithm (Cont'd.)

Claim: if a relation R_i is in the decomposition generated by the above algorithm, then R_i satisfies 3NF.

- Let R_i be generated from the dependency $\alpha \rightarrow \beta$
- Let $\gamma \rightarrow B$ be any non-trivial functional dependency on R_i . (We need only consider FDs whose right-hand side is a single attribute.)
- Now, B can be in either β or α but not in both. Consider each case separately.

Correctness of 3NF Decomposition (Cont'd.)

- Case 1: If B in β :
 - If γ is a superkey, the 2nd condition of 3NF is satisfied
 - Otherwise α must contain some attribute not in γ
 - Since $\gamma \rightarrow B$ is in F^+ it must be derivable from F_c , by using attribute closure on γ .
 - Attribute closure not have used $\alpha \rightarrow \beta$. If it had been used, α must be contained in the attribute closure of γ , which is not possible, since we assumed γ is not a superkey.
 - Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$ (since $\gamma \subseteq \alpha \beta$, and $B \notin \gamma$ since $\gamma \rightarrow B$ is non-trivial)
 - Then, B is extraneous in the right-hand side of $\alpha \rightarrow \beta$; which is not possible since $\alpha \rightarrow \beta$ is in F_c .
 - Thus, if B is in β then γ must be a superkey, and the second condition of 3NF must be satisfied.

Correctness of 3NF Decomposition (Cont'd.)

- Case 2: B is in α .
 - Since α is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.
 - In fact, we cannot show that γ is a superkey.
 - This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.



Chapter 9: Application Design and Development



Chapter 9: Application Design and Development

- Application Programs and User Interfaces
- Web Fundamentals
- Servlets and JSP
- Application Architectures
- Rapid Application Development
- Application Performance
- Application Security
- Encryption and Its Applications



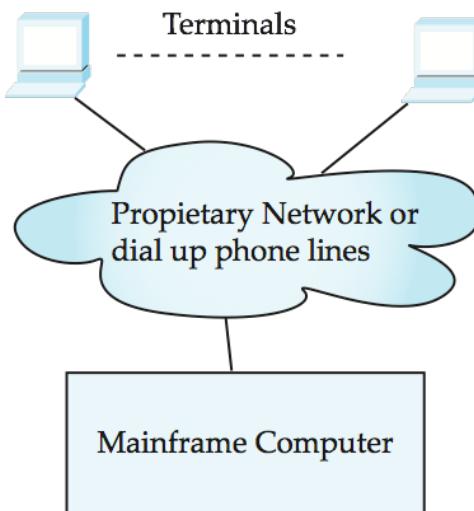
Application Programs and User Interfaces

- Most database users do *not* use a query language like SQL
- An application program acts as the intermediary between users and the database
 - Applications split into
 - ▶ front-end
 - ▶ middle layer
 - ▶ backend
- Front-end: user interface
 - Forms
 - Graphical user interfaces
 - Many interfaces are Web-based

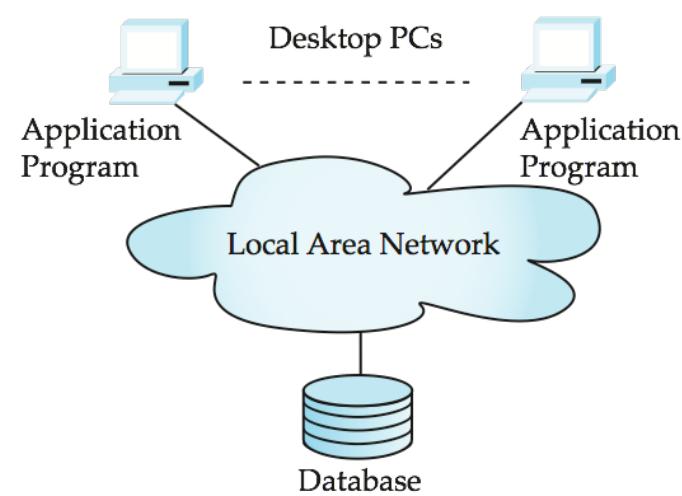


Application Architecture Evolution

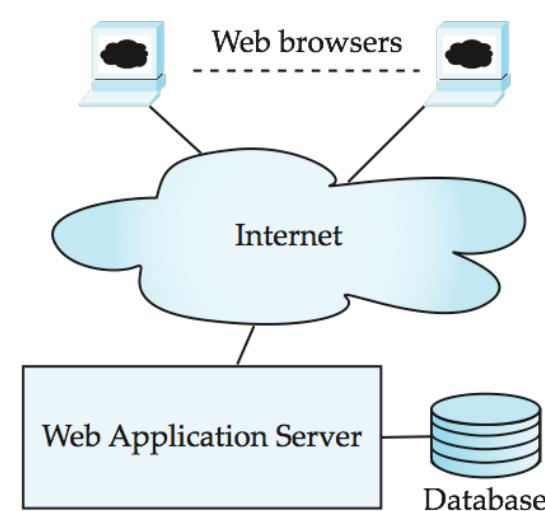
- Three distinct era's of application architecture
 - mainframe (1960's and 70's)
 - personal computer era (1980's)
 - Web era (1990's onwards)



(a) Mainframe Era



(b) Personal Computer Era



(c) Web era



Web Interface

- Web browsers have become the de-facto standard user interface to databases
 - Enable large numbers of users to access databases from anywhere
 - Avoid the need for downloading/installing specialized code, while providing a good graphical user interface
 - ▶ Javascript, Flash and other scripting languages run in browser, but are downloaded transparently
 - Examples: banks, airline and rental car reservations, university course registration and grading, and so on.



The World Wide Web

- The Web is a distributed information system based on hypertext.
- Most Web documents are hypertext documents formatted via the HyperText Markup Language (HTML)
- HTML documents contain
 - text along with font specifications, and other formatting instructions
 - hypertext links to other documents, which can be associated with regions of the text.
 - **forms**, enabling users to enter data which can then be sent back to the Web server



Uniform Resources Locators

- In the Web, functionality of pointers is provided by Uniform Resource Locators (URLs).

- URL example:

<http://www.acm.org/sigmod>

- The first part indicates how the document is to be accessed
 - ▶ “http” indicates that the document is to be accessed using the Hyper Text Transfer Protocol.
- The second part gives the unique name of a machine on the Internet.
- The rest of the URL identifies the document within the machine.
- The local identification can be:
 - ▶ The path name of a file on the machine, or
 - ▶ An identifier (path name) of a program, plus arguments to be passed to the program
 - E.g., <http://www.google.com/search?q=silberschatz>



HTML and HTTP

- HTML provides formatting, hypertext link, and image display features
 - including tables, stylesheets (to alter default formatting), etc.
- HTML also provides input features
 - ▶ Select from a set of options
 - Pop-up menus, radio buttons, check lists
 - ▶ Enter values
 - Text boxes
 - Filled in input sent back to the server, to be acted upon by an executable at the server
- HyperText Transfer Protocol (HTTP) used for communication with the Web server



Sample HTML Source Text

```
<html>
<body>
    <table border>
        <tr> <th>ID</th> <th>Name</th> <th>Department</th> </tr>
        <tr> <td>00128</td> <td>Zhang</td> <td>Comp. Sci.</td> </tr>
        ....
    </table>
    <form action="PersonQuery" method=get>
        Search for:
        <select name="personotype">
            <option value="student" selected>Student </option>
            <option value="instructor"> Instructor </option>
        </select> <br>
        Name: <input type=text size=20 name="name">
        <input type=submit value="submit">
    </form>
</body> </html>
```



Display of Sample HTML Source

ID	Name	Department
00128	Zhang	Comp. Sci.
12345	Shankar	Comp. Sci.
19991	Brandt	History

Search for:

Name:

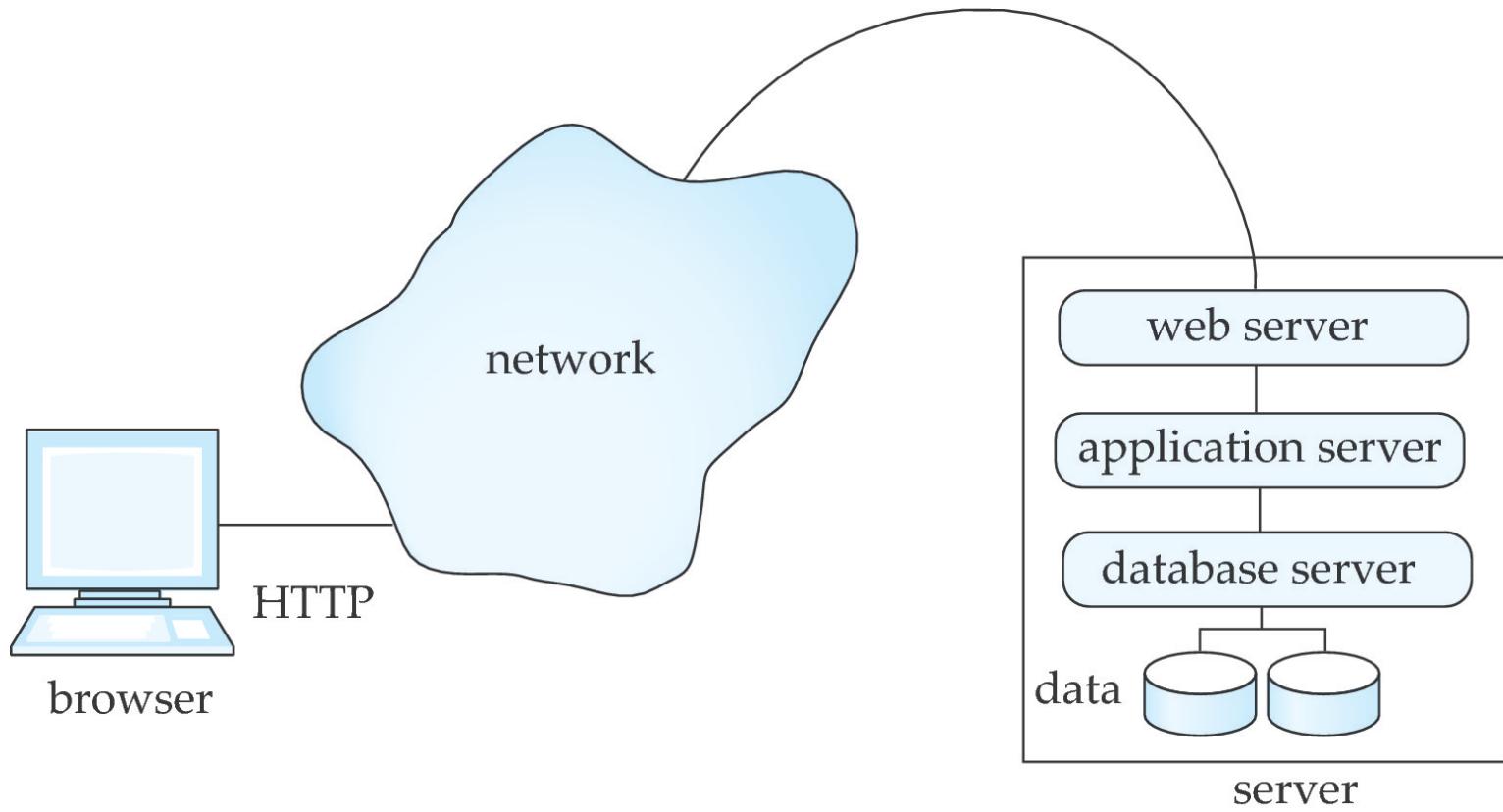


Web Servers

- A Web server can easily serve as a front end to a variety of information services.
- The document name in a URL may identify an executable program, that, when run, generates a HTML document.
 - When an HTTP server receives a request for such a document, it executes the program, and sends back the HTML document that is generated.
 - The Web client can pass extra arguments with the name of the document.
- To install a new service on the Web, one simply needs to create and install an executable that provides that service.
 - The Web browser provides a graphical user interface to the information service.
- Common Gateway Interface (CGI): a standard interface between web and application server



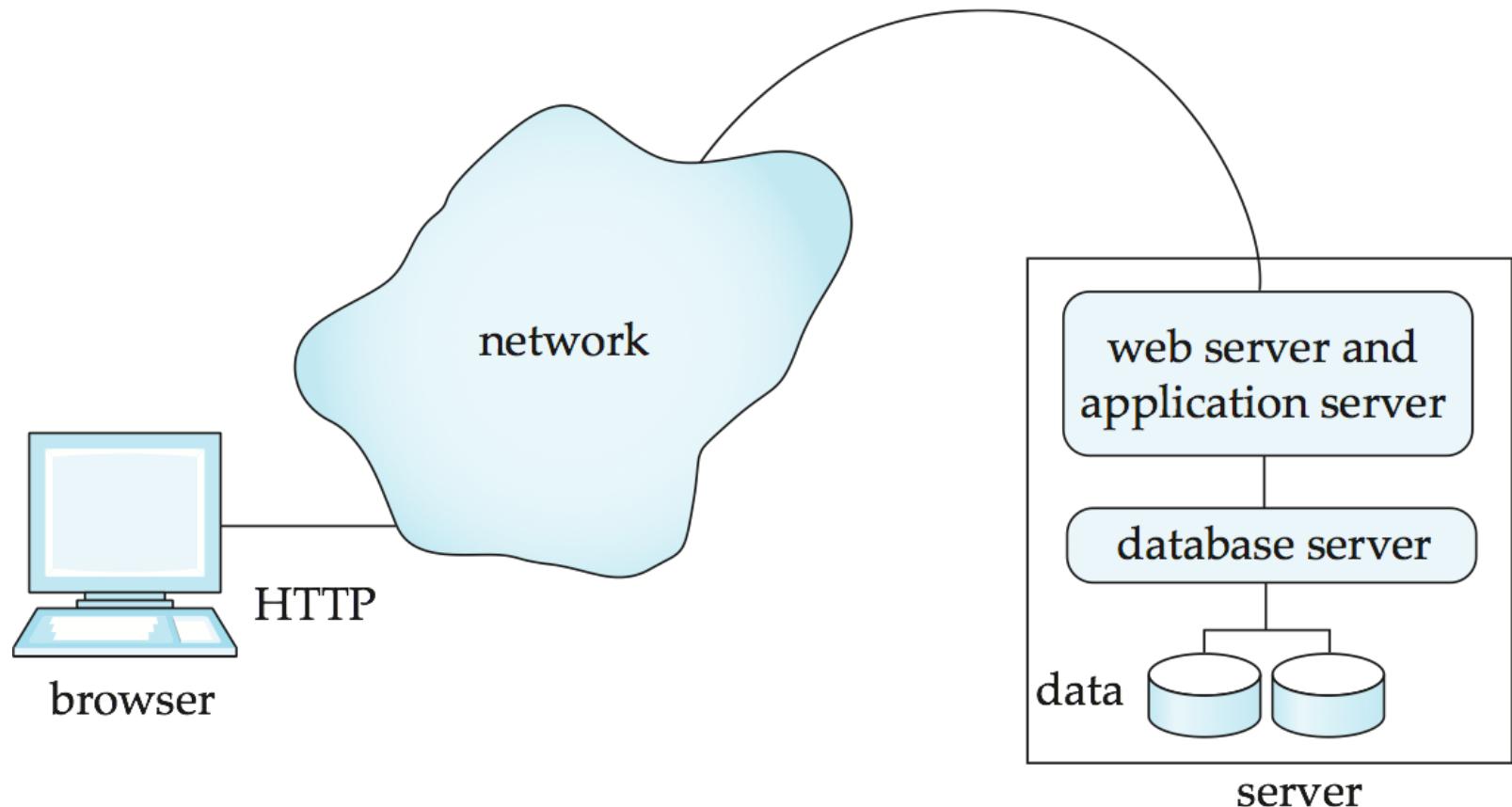
Three-Layer Web Architecture





Two-Layer Web Architecture

- Multiple levels of indirection have overheads
Alternative: two-layer architecture





HTTP and Sessions

- The HTTP protocol is **connectionless**
 - That is, once the server replies to a request, the server closes the connection with the client, and forgets all about the request
 - In contrast, Unix logins, and JDBC/ODBC connections stay connected until the client disconnects
 - ▶ retaining user authentication and other information
 - Motivation: reduces load on server
 - ▶ operating systems have tight limits on number of open connections on a machine
- Information services need session information
 - E.g., user authentication should be done only once per session
- Solution: use a **cookie**



Sessions and Cookies

- A **cookie** is a small piece of text containing identifying information
 - Sent by server to browser
 - ▶ Sent on first interaction, to identify session
 - Sent by browser to the server that created the cookie on further interactions
 - ▶ part of the HTTP protocol
 - Server saves information about cookies it issued, and can use it when serving a request
 - ▶ E.g., authentication information, and user preferences
- Cookies can be stored permanently or for a limited time



Servlets

- Java Servlet specification defines an API for communication between the Web/application server and application program running in the server
 - E.g., methods to get parameter values from Web forms, and to send HTML text back to client
- Application program (also called a servlet) is loaded into the server
 - Each request spawns a new thread in the server
 - ▶ thread is closed once the request is serviced



Example Servlet Code

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class PersonQueryServlet extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response)
                      throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HEAD><TITLE> Query Result</TITLE></HEAD>");
        out.println("<BODY>");
        ..... BODY OF SERVLET (next slide) ...
        out.println("</BODY>");
        out.close();
    }
}
```



Example Servlet Code

```
String persontype = request.getParameter("persontype");
String number = request.getParameter("name");
if(persontype.equals("student")) {
    ... code to find students with the specified name ...
    ... using JDBC to communicate with the database ..
    out.println("<table BORDER COLS=3>");
    out.println(" <tr> <td>ID</td> <td>Name: </td>" + " <td>Department</td> </tr>");
    for(... each result ...){
        ... retrieve ID, name and dept name
        ... into variables ID, name and deptname
        out.println("<tr> <td>" + ID + "</td>" + "<td>" + name + "</td>" + "<td>" + deptname
                    + "</td></tr>");
    };
    out.println("</table>");
}
else {
    ... as above, but for instructors ...
}
```



Servlet Sessions

- Servlet API supports handling of sessions
 - Sets a cookie on first interaction with browser, and uses it to identify session on further interactions
- To check if session is already active:
 - `if (request.getSession(false) == true)`
 - ▶ .. then existing session
 - ▶ else .. redirect to authentication page
 - authentication page
 - ▶ check login/password
 - ▶ `request.getSession(true)`: creates new session
- Store/retrieve attribute value pairs for a particular session
 - `session.setAttribute("userid", userid)`
 - `session.getAttribute("userid")`



Servlet Support

- Servlets run inside application servers such as
 - Apache Tomcat, Glassfish, JBoss
 - BEA Weblogic, IBM WebSphere and Oracle Application Servers
- Application servers support
 - deployment and monitoring of servlets
 - Java 2 Enterprise Edition (J2EE) platform supporting objects, parallel processing across multiple application servers, etc



Server-Side Scripting

- Server-side scripting simplifies the task of connecting a database to the Web
 - Define an HTML document with embedded executable code/SQL queries.
 - Input values from HTML forms can be used directly in the embedded code/SQL queries.
 - When the document is requested, the Web server executes the embedded code/SQL queries to generate the actual HTML document.
- Numerous server-side scripting languages
 - JSP, PHP
 - General purpose scripting languages: VBScript, Perl, Python



Java Server Pages (JSP)

- A JSP page with embedded Java code

```
<html>
<head> <title> Hello </title> </head>
<body>
<% if (request.getParameter("name") == null)
{ out.println("Hello World"); }
else { out.println("Hello, " + request.getParameter("name")); }
%>
</body>
</html>
```

- JSP is compiled into Java + Servlets
- JSP allows new tags to be defined, in tag libraries
 - such tags are like library functions, can be used for example to build rich user interfaces such as paginated display of large datasets



PHP

- PHP is widely used for Web server scripting
- Extensive libraries including for database access using ODBC

```
<html>
  <head> <title> Hello </title> </head>
  <body>
    <?php if (!isset($_REQUEST['name'])) {
      echo "Hello World";
    }
    else { echo "Hello, " + $_REQUEST['name'];
    }
  ?>
  </body>
</html>
```



Client Side Scripting

- Browsers can fetch certain scripts (**client-side scripts**) or programs along with documents, and execute them in “safe mode” at the client site
 - Javascript
 - Macromedia Flash and Shockwave for animation/games
 - VRML
 - Applets
- Client-side scripts/programs allow documents to be active
 - E.g., animation by executing programs at the local site
 - E.g., ensure that values entered by users satisfy some correctness checks
 - Permit flexible interaction with the user.
 - ▶ Executing programs at the client site speeds up interaction by avoiding many round trips to server



Client Side Scripting and Security

- Security mechanisms needed to ensure that malicious scripts do not cause damage to the client machine
 - Easy for limited capability scripting languages, harder for general purpose programming languages like Java
- E.g., Java's security system ensures that the Java applet code does not make any system calls directly
 - Disallows dangerous actions such as file writes
 - Notifies the user about potentially dangerous actions, and allows the option to abort the program or to continue execution.



Javascript

- Javascript very widely used
 - forms basis of new generation of Web applications (called Web 2.0 applications) offering rich user interfaces
- Javascript functions can
 - check input for validity
 - modify the displayed Web page, by altering the underling **document object model (DOM)** tree representation of the displayed HTML text
 - communicate with a Web server to fetch data and modify the current page using fetched data, without needing to reload/refresh the page
 - ▶ forms basis of AJAX technology used widely in Web 2.0 applications
 - ▶ E.g. on selecting a country in a drop-down menu, the list of states in that country is automatically populated in a linked drop-down menu



Javascript

- Example of Javascript used to validate form input

```
<html> <head>
  <script type="text/javascript">
    function validate() {
      var credits=document.getElementById("credits").value;
      if (isNaN(credits)|| credits<=0 || credits>=16) {
        alert("Credits must be a number greater than 0 and less than 16");
        return false
      }
    }
  </script>
</head> <body>
  <form action="createCourse" onsubmit="return validate()">
    Title: <input type="text" id="title" size="20"><br />
    Credits: <input type="text" id="credits" size="2"><br />
    <input type="submit" value="Submit">
  </form>
</body> </html>
```



Application Architectures

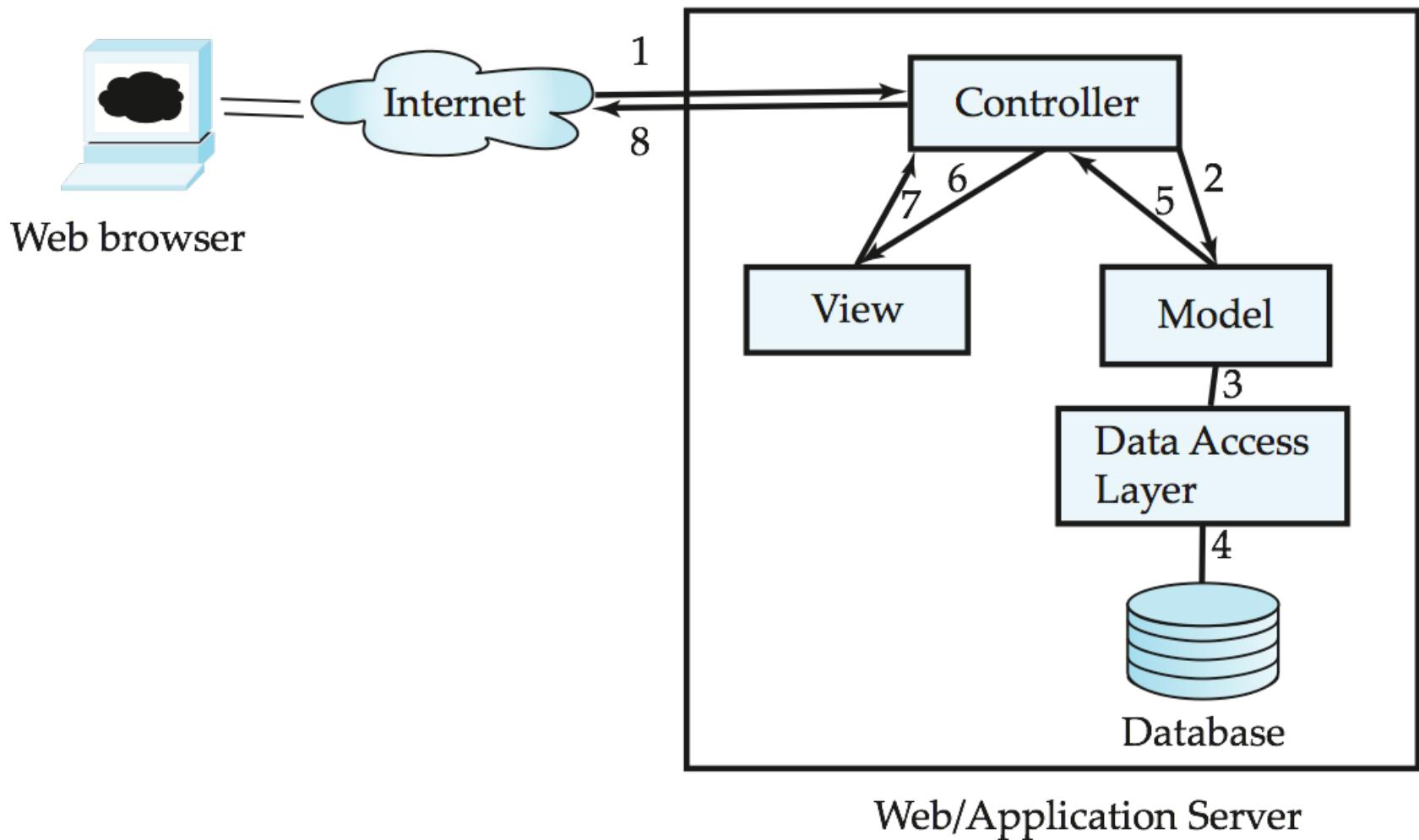


Application Architectures

- Application layers
 - Presentation or user interface
 - ▶ **model-view-controller (MVC)** architecture
 - **model**: business logic
 - **view**: presentation of data, depends on display device
 - **controller**: receives events, executes actions, and returns a view to the user
 - **business-logic** layer
 - ▶ provides high level view of data and actions on data
 - often using an object data model
 - ▶ hides details of data storage schema
 - **data access** layer
 - ▶ interfaces between business logic layer and the underlying database
 - ▶ provides mapping from object model of business layer to relational model of database



Application Architecture





Business Logic Layer

- Provides abstractions of entities
 - e.g. students, instructors, courses, etc
- Enforces **business rules** for carrying out actions
 - E.g. student can enroll in a class only if she has completed prerequisites, and has paid her tuition fees
- Supports **workflows** which define how a task involving multiple participants is to be carried out
 - E.g. how to process application by a student applying to a university
 - Sequence of steps to carry out task
 - Error handling
 - ▶ e.g. what to do if recommendation letters not received on time
 - Workflows discussed in Section 26.2



Object-Relational Mapping

- Allows application code to be written on top of object-oriented data model, while storing data in a traditional relational database
 - alternative: implement object-oriented or object-relational database to store object model
 - ▶ has not been commercially successful
- Schema designer has to provide a mapping between object data and relational schema
 - e.g. Java class *Student* mapped to relation *student*, with corresponding mapping of attributes
 - An object can map to multiple tuples in multiple relations
- Application opens a session, which connects to the database
- Objects can be created and saved to the database using `session.save(object)`
 - mapping used to create appropriate tuples in the database
- Query can be run to retrieve objects satisfying specified predicates



Object-Relational Mapping and Hibernate (Cont.)

- The **Hibernate** object-relational mapping system is widely used
 - public domain system, runs on a variety of database systems
 - supports a query language that can express complex queries involving joins
 - ▶ translates queries into SQL queries
 - allows relationships to be mapped to sets associated with objects
 - ▶ e.g. courses taken by a student can be a set in Student object
 - See book for Hibernate code example
- The **Entity Data Model** developed by Microsoft
 - provides an entity-relationship model directly to application
 - maps data between entity data model and underlying storage, which can be relational
 - Entity SQL language operates directly on Entity Data Model



Web Services

- Allow data on Web to be accessed using remote procedure call mechanism
- Two approaches are widely used
 - **Representation State Transfer (REST)**: allows use of standard HTTP request to a URL to execute a request and return data
 - ▶ returned data is encoded either in XML, or in **JavaScript Object Notation (JSON)**
 - **Big Web Services**:
 - ▶ uses XML representation for sending request data, as well as for returning results
 - ▶ standard protocol layer built on top of HTTP
 - ▶ See Section 23.7.3



Disconnected Operations

- Tools for applications to use the Web when connected, but operate locally when disconnected from the Web
 - E.g. Google Gears browser plugin
 - ▶ Provide a local database, a local Web server and support for execution of JavaScript at the client
 - ▶ JavaScript code using Gears can function identically on any OS/browser platform
 - Adobe AIR software provides similar functionality outside of Web browser



Rapid Application Development

- A lot of effort is required to develop Web application interfaces
 - more so, to support rich interaction functionality associated with Web 2.0 applications
- Several approaches to speed up application development
 - Function library to generate user-interface elements
 - Drag-and-drop features in an IDE to create user-interface elements
 - Automatically generate code for user interface from a declarative specification
- Above features have been used as part of **rapid application development (RAD)** tools even before advent of Web
- Web application development frameworks
 - Java Server Faces (JSF) includes JSP tag library
 - Ruby on Rails
 - ▶ Allows easy creation of simple **CRUD** (create, read, update and delete) interfaces by code generation from database schema or object model



ASP.NET and Visual Studio

- ASP.NET provides a variety of controls that are interpreted at server, and generate HTML code
- Visual Studio provides drag-and-drop development using these controls
 - E.g. menus and list boxes can be associated with DataSet object
 - Validator controls (constraints) can be added to form input fields
 - ▶ JavaScript to enforce constraints at client, and separately enforced at server
 - User actions such as selecting a value from a menu can be associated with actions at server
 - DataGrid provides convenient way of displaying SQL query results in tabular format



Application Performance



Improving Web Server Performance

- Performance is an issue for popular Web sites
 - May be accessed by millions of users every day, thousands of requests per second at peak time
- Caching techniques used to reduce cost of serving pages by exploiting commonalities between requests
 - At the server site:
 - ▶ Caching of JDBC connections between servlet requests
 - a.k.a. **connection pooling**
 - ▶ Caching results of database queries
 - Cached results must be updated if underlying database changes
 - ▶ Caching of generated HTML
 - At the client's network
 - ▶ Caching of pages by Web proxy



Application Security



SQL Injection

- Suppose query is constructed using
 - "select * from instructor where name = '" + name + "'"
- Suppose the user, instead of entering a name, enters:
 - X' or 'Y' = 'Y
- then the resulting statement becomes:
 - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
 - which is:
 - ▶ select * from instructor where name = 'X' or 'Y' = 'Y'
 - User could have even used
 - ▶ X'; update instructor set salary = salary + 10000; --
- Prepared statement internally uses:
"select * from instructor where name = 'X\' or \'Y\' = \'Y"
- **Always use prepared statements, with user inputs as parameters**
- Is the following prepared statemen secure?
 - conn.prepareStatement("select * from instructor where name = '" + name + "'")



Cross Site Scripting

- HTML code on one page executes action on another page
 - E.g. <img src =
<http://mybank.com/transfermoney?amount=1000&toaccount=14523>>
 - Risk: if user viewing page with above code is currently logged into mybank, the transfer may succeed
 - Above example simplistic, since GET method is normally not used for updates, but if the code were instead a script, it could execute POST methods
- Above vulnerability called **cross-site scripting (XSS)** or **cross-site request forgery (XSRF or CSRF)**
- **Prevent your web site from being used to launch XSS or XSRF attacks**
 - Disallow HTML tags in text input provided by users, using functions to detect and strip such tags
- **Protect your web site from XSS/XSRF attacks launched from other sites**
 - ..next slide



Cross Site Scripting

■ Protect your web site from XSS/XSRF attacks launched from other sites

- Use **referer** value (URL of page from where a link was clicked) provided by the HTTP protocol, to check that the link was followed from a valid page served from same site, not another site
- Ensure IP of request is same as IP from where the user was authenticated
 - ▶ prevents hijacking of cookie by malicious user
- Never use a GET method to perform any updates
 - ▶ This is actually recommended by HTTP standard



Password Leakage

- Never store passwords, such as database passwords, in clear text in scripts that may be accessible to users
 - E.g. in files in a directory accessible to a web server
 - ▶ Normally, web server will execute, but not provide source of script files such as file.jsp or file.php, but source of editor backup files such as file.jsp~, or .file.jsp.swp may be served
- Restrict access to database server from IPs of machines running application servers
 - Most databases allow restriction of access by source IP address



Application Authentication

- Single factor authentication such as passwords too risky for critical applications
 - guessing of passwords, sniffing of packets if passwords are not encrypted
 - passwords reused by user across sites
 - spyware which captures password
- Two-factor authentication
 - e.g. password plus one-time password sent by SMS
 - e.g. password plus one-time password devices
 - ▶ device generates a new pseudo-random number every minute, and displays to user
 - ▶ user enters the current number as password
 - ▶ application server generates same sequence of pseudo-random numbers to check that the number is correct.



Application Authentication

- **Man-in-the-middle** attack
 - E.g. web site that pretends to be mybank.com, and passes on requests from user to mybank.com, and passes results back to user
 - Even two-factor authentication cannot prevent such attacks
- Solution: authenticate Web site to user, using digital certificates, along with secure http protocol
- **Central authentication** within an organization
 - application redirects to central authentication service for authentication
 - avoids multiplicity of sites having access to user's password
 - LDAP or Active Directory used for authentication



Single Sign-On

- **Single sign-on** allows user to be authenticated once, and applications can communicate with authentication service to verify user's identity without repeatedly entering passwords
- **Security Assertion Markup Language (SAML)** standard for exchanging authentication and authorization information across security domains
 - e.g. user from Yale signs on to external application such as acm.org using userid joe@yale.edu
 - application communicates with Web-based authentication service at Yale to authenticate user, and find what the user is authorized to do by Yale (e.g. access certain journals)
- **OpenID** standard allows sharing of authentication across organizations
 - e.g. application allows user to choose Yahoo! as OpenID authentication provider, and redirects user to Yahoo! for authentication



Application-Level Authorization

- Current SQL standard does not allow fine-grained authorization such as “students can see their own grades, but not other’s grades”
 - Problem 1: Database has no idea who are application users
 - Problem 2: SQL authorization is at the level of tables, or columns of tables, but not to specific rows of a table
- One workaround: use views such as

```
create view studentTakes as
select *
from takes
where takes.ID = syscontext.user_id()
```

- where syscontext.user_id() provides end user identity
 - ▶ end user identity must be provided to the database by the application
- Having multiple such views is cumbersome



Application-Level Authorization (Cont.)

- Currently, authorization is done entirely in application
- Entire application code has access to entire database
 - large surface area, making protection harder
- Alternative: **fine-grained (row-level) authorization** schemes
 - extensions to SQL authorization proposed but not currently implemented
 - Oracle Virtual Private Database (VPD) allows predicates to be added transparently to all SQL queries, to enforce fine-grained authorization
 - ▶ e.g. add `ID= sys_context.user_id()` to all queries on student relation if user is a student



Audit Trails

- Applications must log actions to an audit trail, to detect who carried out an update, or accessed some sensitive data
- Audit trails used after-the-fact to
 - detect security breaches
 - repair damage caused by security breach
 - trace who carried out the breach
- Audit trails needed at
 - Database level, and at
 - Application level



Encryption



Encryption

- Data may be *encrypted* when database authorization provisions do not offer sufficient protection.
- Properties of good encryption technique:
 - Relatively simple for authorized users to encrypt and decrypt data.
 - Encryption scheme depends not on the secrecy of the algorithm but on the secrecy of a parameter of the algorithm called the encryption key.
 - Extremely difficult for an intruder to determine the encryption key.
- **Symmetric-key encryption:** same key used for encryption and for decryption
- **Public-key encryption** (a.k.a. **asymmentric-key encryption**): use different keys for encryption and decryption
 - encryption key can be public, decryption key secret



Encryption (Cont.)

- *Data Encryption Standard (DES)* substitutes characters and rearranges their order on the basis of an encryption key which is provided to authorized users via a secure mechanism. Scheme is no more secure than the key transmission mechanism since the key has to be shared.
- *Advanced Encryption Standard (AES)* is a new standard replacing DES, and is based on the Rijndael algorithm, but is also dependent on shared secret keys.
- *Public-key encryption* is based on each user having two keys:
 - *public key* – publicly published key used to encrypt data, but cannot be used to decrypt data
 - *private key* -- key known only to individual user, and used to decrypt data. Need not be transmitted to the site doing encryption.

Encryption scheme is such that it is impossible or extremely hard to decrypt data given only the public key.

- The RSA public-key encryption scheme is based on the hardness of factoring a very large number (100's of digits) into its prime components



Encryption (Cont.)

- **Hybrid schemes** combining public key and private key encryption for efficient encryption of large amounts of data
- Encryption of small values such as identifiers or names vulnerable to **dictionary attacks**
 - especially if encryption key is publicly available
 - but even otherwise, statistical information such as frequency of occurrence can be used to reveal content of encrypted data
 - Can be deterred by adding extra random bits to the end of the value, before encryption, and removing them after decryption
 - ▶ same value will have different encrypted forms each time it is encrypted, preventing both above attacks
 - ▶ extra bits are called **salt bits**



Encryption in Databases

- Database widely support encryption
- Different levels of encryption:
 - **disk block**
 - ▶ every disk block encrypted using key available in database-system software.
 - ▶ Even if attacker gets access to database data, decryption cannot be done without access to the key.
 - **Entire relations, or specific attributes of relations**
 - ▶ non-sensitive relations, or non-sensitive attributes of relations need not be encrypted
 - ▶ however, attributes involved in primary/foreign key constraints cannot be encrypted.
- Storage of encryption or decryption keys
 - typically, single master key used to protect multiple encryption/decryption keys stored in database
- Alternative: encryption/decryption is done in application, before sending values to the database



Encryption and Authentication

- Password based authentication is widely used, but is susceptible to sniffing on a network.
- **Challenge-response** systems avoid transmission of passwords
 - DB sends a (randomly generated) challenge string to user.
 - User encrypts string and returns result.
 - DB verifies identity by decrypting result
 - Can use public-key encryption system by DB sending a message encrypted using user's public key, and user decrypting and sending the message back.
- **Digital signatures** are used to verify authenticity of data
 - E.g., use private key (in reverse) to encrypt data, and anyone can verify authenticity by using public key (in reverse) to decrypt data. Only holder of private key could have created the encrypted data.
 - Digital signatures also help ensure **nonrepudiation**: sender cannot later claim to have not created the data



End of Chapter



Digital Certificates

- **Digital certificates** are used to verify authenticity of public keys.
- Problem: when you communicate with a web site, how do you know if you are talking with the genuine web site or an imposter?
 - Solution: use the public key of the web site
 - Problem: how to verify if the public key itself is genuine?
- Solution:
 - Every client (e.g., browser) has public keys of a few root-level **certification authorities**
 - A site can get its name/URL and public key signed by a certification authority: signed document is called a **certificate**
 - Client can use public key of certification authority to verify certificate
 - Multiple levels of certification authorities can exist. Each certification authority
 - ▶ presents its own public-key certificate signed by a higher level authority, and
 - ▶ uses its private key to sign the certificate of other web sites/authorities



A formatted report

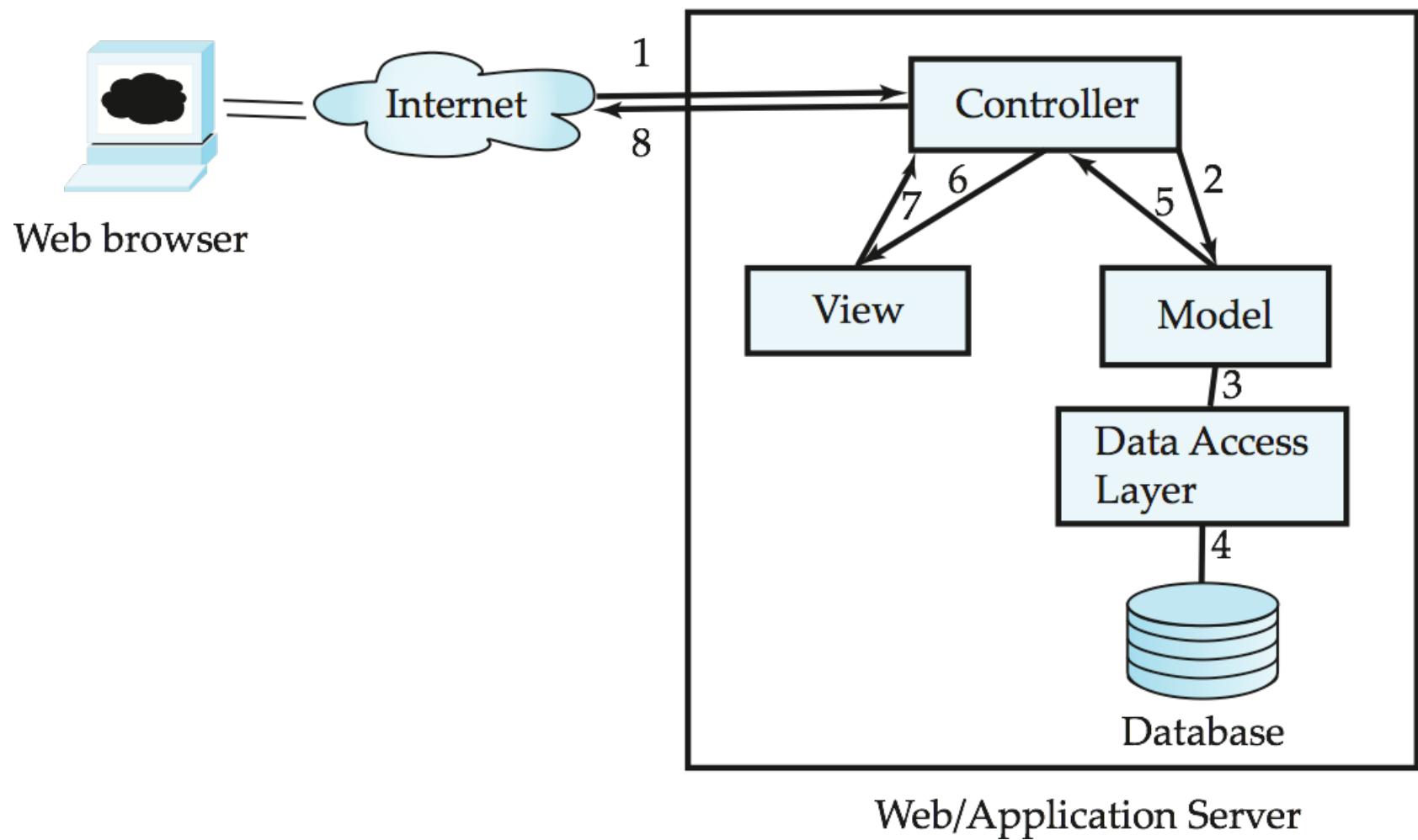
Acme Supply Company, Inc. Quarterly Sales Report

Period: Jan. 1 to March 31, 2009

Region	Category	Sales	Subtotal
North	Computer Hardware	1,000,000	1,500,000
	Computer Software	500,000	
	All categories		
South	Computer Hardware	200,000	600,000
	Computer Software	400,000	
	All categories		
Total Sales		2,100,000	



Figure 9.11





Web Interfaces to Database (Cont.)

2. Dynamic generation of documents

- Limitations of static HTML documents
 - ▶ Cannot customize fixed Web documents for individual users.
 - ▶ Problematic to update Web documents, especially if multiple Web documents replicate data.
- Solution: Generate Web documents dynamically from data stored in a database.
 - ▶ Can tailor the display based on user information stored in the database.
 - E.g., tailored ads, tailored weather and local news, ...
 - ▶ Displayed information is up-to-date, unlike the static Web pages
 - E.g., stock market information, ..

Advanced SQL

CS3043 - Database Systems

Overview

- JDBC
- Prepared Statements and SQL injection
- ODBC
- Embedded SQL
- Procedural constructs in SQL
- Triggers
- Advanced aggregation features
- Authorization

JDBC and ODBC

- API (Application Program Interface) for a program to interact with a database server
- Application makes calls to
 - Connect with the database server
 - Send SQL commands to the database server
 - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
 - Other API's such as ADO.NET sit on top of ODBC
- JDBC (Java Database Connectivity) works with Java

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
 - Open a connection
 - Create a “statement” object
 - Execute queries using the Statement object to send queries and fetch results
 - Close the connections
 - Handle errors using exception mechanism

JDBC example

```
import java.sql.*;

...
static final String DB_URL = "jdbc:mysql://localhost/university";
static String USER;
static String PASSWORD;
static final String QUERY = "SELECT ID,name FROM student";

...
try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASSWORD);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(QUERY)) {
    // Extract data from result set
    while (rs.next()) {
        // Retrieve by column name
        System.out.println("ID: " + rs.getInt("id"));
        System.out.println("Name: " + rs.getString("name"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

JDBC examples (cont'd)

- Update to database

```
try {
    statement.executeUpdate("INSERT INTO instructor VALUES
                           ('77987', 'Kim', 'Physics', 98000)");

} catch (SQLException e) {
    log.error("Unable to access the database server: ", e);
}
```

- Execute query and fetch results

```
ResultSet resultSet = statement.executeQuery("SELECT dept_name, AVG(salary)
                                             FROM instructor GROUP BY dept_name"
                                             );
while (resultSet.next()) {
    System.out.println(resultSet.getString("dept_name") + " : " +
                       resultSet.getFloat(2));
}
```

JDBC Code

- Getting result fields
 - `resultSet.getString("dept_name")`
 - `resultSet.getString(1)`
- Dealing with NULL values
 - If a result set is empty after executing a statement it should be handled properly.
 - As a best practice, always prepare for potential null values after executing a query, since you are not aware of the result set until you execute the query.

equivalent if `dept_name` is the first attribute of the SELECT result

Prepared Statement

```
PreparedStatement preparedStatement = connection.prepareStatement  
        ("INSERT INTO instructor  
         VALUES (?, ?, ?, ?)");  
  
preparedStatement.setString ( 1, "88877" );  
preparedStatement.setString ( 2, "Perry" );  
preparedStatement.setString ( 3, "Finance" );  
preparedStatement.setInt ( 4, 125000 );  
preparedStatement.executeUpdate();
```

- For queries, use `preparedStatement.executeQuery()` which returns a `ResultSet`.
- Always use prepared statements when taking an input from the user and adding it to a query

Prepared Statement

```
PreparedStatement preparedStatement = connection.prepareStatement  
        ("INSERT INTO instructor  
         VALUES (?, ?, ?, ?, ?)");  
  
preparedStatement.setString(1, "88877");  
preparedStatement.setString(2, "Perry");  
preparedStatement.setString(3, "Finance");  
preparedStatement.setInt(4, 125000);  
preparedStatement.executeUpdate();
```

- For queries, use `preparedStatement.executeQuery()` which returns a `ResultSet`.
- Always use prepared statements when taking an input from the user and adding it to a query
 - NEVER create a query by concatenating strings taken as user inputs

```
"INSERT INTO instructor VALUES (' ' + ID + " ', ' " + name + " ', " + " ' +  
dept name + " ', " ' balance + ")"
```

- What if the name is “D’Souza”?

SQL Injection

- Suppose a query is constructed using
 - `"select * from instructor where name = '" + name + "'"`
- Suppose the user enters the following, instead of entering the name
 - `X' or 'Y' = 'Y`
- Then the resulting query becomes:
 - `"select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"`
 - which is
`select * from instructor where name = 'X' or 'Y' = 'Y'`
- Prepared Statements internally sanitize the inputs.

`select * from instructor where name = 'X\' or \'Y\' = \'Y'`
- Always use Prepared Statements if the parameters for a query are taken as user inputs.

Transaction control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
 - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
 - `conn.setAutoCommit(false);`
- Transactions must then be committed or rolled back explicitly
 - `conn.commit();` or
 - `conn.rollback();`
- `conn.setAutoCommit(true)` turns on automatic commit.

Other JDBC features

- Handling large object types
 - `getBlob()` and `getClob()` can be used to return objects of type Blob or Clob, respectively.
 - associate an open stream with Java Blob or Clob object to update large objects

```
blob.setBlob(int parameterIndex, InputStream inputStream)
```

- Metadata features
 - After executing a query to get the result set, you can access the metadata of the result set by,

```
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

ODBC - Open Database Connectivity Standard

- Standard for an application program to communicate with a database server.
- Application program interface (API) to
 - open a connection with a database,
 - send queries and updates,
 - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
- Was defined originally for Basic and C, versions available for many languages.
- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Must also specify types of arguments:
 - SQL_NTS denotes previous argument is a null-terminated string.

ODBC example

```
int ODBCexample()
{
    RETCODE error;
    HENV env; /* environment */
    HDBC conn; /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "avipasswd", SQL_NTS);

    { .... Do actual work ... }

    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

Read more about ODBC from Chapter 4
of the recommended text.

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded SQL*.
- An embedded SQL program must be pre-processed by a special preprocessor prior to compilation.
 - The preprocessor replaces embedded SQL commands with host language declarations that allow run-time execution.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement> **END_EXEC**

- Note: this varies by language
 - for example, the Java embedding uses **#SQL { };**

SQLJ

- JDBC is overly dynamic - errors cannot be caught by the Java compiler
- SQLJ: embedded SQL in Java
- How does embedding help to detect errors early?

```
#sql iterator deptInfoIter ( String dept_name, int avgSal);
deptInfoIter iter = null;
#sql iter = { select dept_name,avg(salary) from
                instructor group by dept_name };
while (iter.next()) {
    String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
}
iter.close();
```

Procedural Constructs in SQL

Procedural Extensions and Stored Procedures

- SQL provides a **module** language
 - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
- Stored Procedures
 - Can store procedures in the database
 - then execute them using the **call** statement
 - permit external applications to operate on the database without knowing about internal details
- Object-oriented aspects of these features are covered in Chapter 22 (Object Based Databases)

SQL Functions

- SQL:1999 supports functions and procedures
 - Functions/procedures can be written in SQL itself, or in an external programming language.
 - Functions are particularly useful with specialized data types such as images and geometric objects.
 - Example: functions to check if polygons overlap, or to compare images for similarity.
 - Some database systems support **table-valued functions**, which can return a relation as a result.
- SQL:1999 also supports a rich set of imperative constructs, including
 - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
returns integer
begin
    declare d_count integer;
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
end
```

- Find the department name and budget of all departments with more than 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name) > 12
```

Table Functions

- SQL:2003 added functions that return a relation as a result
 - Example: Return all accounts owned by a given customer

```
create function instructors_of (dept_name char(20)
                               returns table ( ID varchar(5),
                                                name varchar(20),
                                                dept_name varchar(20),
                                                salary numeric(8,2))
                               return table
                               (select ID, name, dept_name, salary
                                from instructor
                                where instructor.dept_name = instructors_of.dept_name)
```

- Usage

```
select *
from table (instructors_of ('Music'))
```

SQL Procedures

- The *dept_count* function could instead be written as procedure:

```
create procedure dept_count_proc (in dept_name varchar(20),
                                   out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.
- Procedures and functions can be invoked also from dynamic SQL
- SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ

Procedural Constructs

- Warning: most database systems implement their own variant of the standard syntax below
 - read your system manual to see what works on your system
- Compound statement: **begin ... end**,
 - May contain multiple SQL statements between **begin** and **end**.
 - Local variables can be declared within a compound statements
- **While** and **repeat** statements :

```
declare n integer default 0;  
while n < 10 do  
    set n = n + 1  
end while
```

```
repeat  
    set n = n - 1  
until n = 0  
end repeat
```

Procedural Constructs

- For loop
 - Permits iteration over all results of a query

Example:

```
declare n integer default 0;  
for r as  
    select budget from department  
    where dept_name = 'Music'  
do  
    set n = n - r.budget  
end for
```

External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),  
                                out count integer)  
language C  
external name '/usr/avi/bin/dept_count_proc'
```

```
create function dept_count(dept_name varchar(20))  
returns integer  
language C  
external name '/usr/avi/bin/dept_count'
```

External Language Routines (Cont.)

- Benefits of external language functions/procedures:
 - more efficient for many operations, and more expressive power.
- Drawbacks
 - Code to implement function may need to be loaded into the database system and executed in the database system's address space.
 - risk of accidental corruption of database structures
 - security risk, allowing users access to unauthorized data
 - There are alternatives, which provide security at the cost of potentially worse performance.
 - Use **sandbox** techniques
 - run external language functions/procedures in a separate process, with no access to the memory of the database process
 - Direct execution in the database system's space is used when efficiency is more important than security.

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
 - Specify the conditions under which the trigger is to be executed.
 - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
- Syntax illustrated here may not work exactly on your database system; check the system manuals.

Trigger example

- E.g. *time_slot_id* is not a primary key of *time_slot relation*, so we cannot create a foreign key constraint from *section relation* to *time_slot relation*.
- Alternative: use triggers on *section* and *time_slot* to enforce integrity constraints

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
        select time_slot_id
        from time_slot)/* time_slot_id not
                           present in time_slot */
begin
    rollback
end;
```

Trigger example (cont'd)

```
create trigger timeslot_check2 after delete on time_slot
referencing old row as orow
for each row
when (orow.time_slot_id not in (
        select time_slot_id
        from time_slot) /* last tuple for time slot id
                           deleted from time slot */
      and orow.time_slot_id in (
        select time_slot_id
        from section)) /* and time_slot_id still
                           referenced from section*/
begin
  rollback
end;
```

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
 - E.g., after **update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
 - **referencing old row as** : for deletes and updates
 - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event which can serve as extra constraints.
 - E.g. convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = '')
begin atomic
    set nrow.grade = null;
end;
```

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
 - Use **for each statement** instead of **for each row**
 - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
- Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
 - maintaining summary data (e.g., total salary of each department)
 - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
 - Databases today provide built in materialized view facilities to maintain summary data
 - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
 - Define methods to update fields
 - Carry out actions as part of the update methods instead of through a trigger

When Not To Use Triggers

- Risk of unintended execution of triggers, for example, when
 - loading data from a backup copy
 - replicating updates at a remote site
 - Trigger execution can be disabled before such actions.
- Other risks with triggers:
 - Error leading to failure of critical transactions that set off the trigger
 - Cascading execution

Advanced Aggregation Features

Ranking

- Ranking is done in conjunction with an order by specification.
- Suppose we are given a relation; *student_grades*(*ID*, *GPA*) giving the grade-point average of each student. Find the rank of each student.

```
select ID, rank() over (order by GPA desc) as s_rank_rank  
from student_grades
```

- An extra **order by** clause is needed to get them in sorted order

```
select ID, rank() over (order by GPA desc) as s_rank  
from student_grades  
order by s_rank
```

- Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
 - **dense_rank** does not leave gaps, so next dense rank would be 2
- Supported from MySQL 8.0.2. Syntax may differ, please check the manual.

Windowing

- Used to smooth out random variations.
 - E.g., **moving average**: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”
- **Window specification** in SQL:
 - Given relation *sales(date value) , value*)

```
select date, sum(value) over
(order by date between rows 1 preceding and 1 following)
from sales
```
- Examples of other window specifications:
 - **between rows unbounded preceding and current**
 - **rows unbounded preceding**
 - **range between 10 preceding and current row**
 - All rows with values between current row value –10 to current value
 - **range interval 10 day preceding**
 - Not including current row
- Supported from MySQL 8.0.2.

Authorization

Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

```
grant <privilege list>  
on <relation name or view name> to <user list>
```

- <user list> is:
 - a user-id
 - **public**, which allows all valid users the privilege granted
 - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the required privileges on the specified item (or be the database administrator).

Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view

Example: grant users U_1 , U_2 , and U_3 **select** authorization on the *instructor* relation:

```
grant select on instructor to  $U_1$ ,  $U_2$ ,  $U_3$ 
```

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

```
revoke <privilege list>  
      on <relation name or view name> from <user list>
```

- Example:

```
revoke select on branch from U1 U2 U3
```

- <privilege-list> may be **all** to revoke all privileges the revoker may hold.
- If <revoker-list> includes **public**, all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantors, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Roles

- **create role** instructor;
- **grant** *instructor* **to** Amit;
- Privileges can be granted to roles:
 - **grant select on** takes **to** instructor;
- Roles can be granted to users, as well as to other roles
 - **create role** teaching_assistant;
 - **grant** teaching_assistant **to** instructor;
 - Instructor inherits all privileges of teaching_assistant
- Chain of roles
 - **create role** dean;
 - **grant** instructor **to** dean;
 - **grant** dean **to** Satoshi;

Other Authorization Features

- **references** privilege to create foreign key
 - `grant reference (dept_name) on department to Mariano;`
 - why is this required?
- transfer of privileges
 - `grant select on department to Amit with grant option;`
 - `revoke select on department from Amit, Satoshi cascade;`
 - `revoke select on department from Amit, Satoshi restrict;`
- Read Section 4.6 for more details we have omitted here.

Thank you!

OLAP

Online Analytical Processing

Out of the scope of CS3043

OLAP - Online Analytical Processing

- **Online Analytical Processing (OLAP)**
 - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- Data that can be modeled as dimension attributes and measure attributes are called **multidimensional data**.
 - **Measure attributes**
 - measure some value
 - can be aggregated upon
 - e.g., the attribute *number* of the *sales* relation
 - **Dimension attributes**
 - define the dimensions on which measure attributes (or aggregates thereof) are viewed
 - e.g., attributes *item_name*, *color*, and *size* of the *sales* relation

Example - Cross Tabulation of sales by item_name

and also

item_name	color	clothes_size	quantity
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	1
...
...

Sales relation

clothes_size all

item_name	color			
	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

The table above is an example of a **cross-tabulation (cross-tab)**, also referred to as a **pivot-table**.

- Values for one of the dimension attributes form the row headers
- Values for another dimension attribute form the column headers
- Other dimension attributes are listed on top
- Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

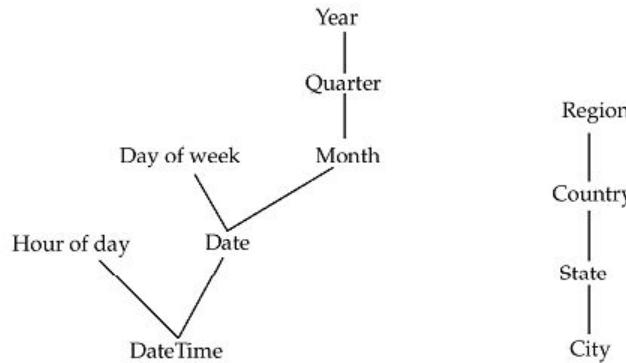
Data Cube

- A **data cube** is a multidimensional generalization of a cross-tab
- Can have n dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube

		item_name					clothes_size			
		skirt	dress	shirt	pants	all	all	large	medium	small
color		dark	8	20	14	20	62	4	16	34
		pastel	35	10	7	2	54	9	18	21
color		white	10	8	28	5	48	42	45	77
		all	53	38	49	27	164	all	large	medium

Hierarchies on Dimensions

- **Hierarchy** on dimension attributes: lets dimensions to be viewed at different levels of detail
 - E.g., the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year



a) Time Hierarchy



b) Location Hierarchy

Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
 - Can drill down or roll up on a hierarchy

clothes_size: **all**

	category	item_name	color			total
			dark	pastel	white	
womenswear	skirt	8	8	10	53	88
	dress	20	20	5	35	
	subtotal	28	28	15		
menswear	pants	14	14	28	49	76
	shirt	20	20	5	27	
	subtotal	34	34	33		
total		62	62	48		164

Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations
 - the value **all** is used to represent aggregates.
 - The SQL standard actually uses null values in place of **all** despite confusion with regular null values.

item_name	color	clothes_size	quantity
skirt	dark	all	8
skirt	pastel	all	35
skirt	white	all	10
skirt	all	all	53
dress	dark	all	20
dress	pastel	all	10
dress	white	all	5
dress	all	all	35
shirt	dark	all	14
shirt	pastel	all	7
shirt	White	all	28
shirt	all	all	49
pant	dark	all	20
pant	pastel	all	2
pant	white	all	5
pant	all	all	27
all	dark	all	62
all	pastel	all	54
all	white	all	48
all	all	all	164

Online Analytical Processing Operations

- **Pivoting:** changing the dimensions used in a cross-tab
- **Slicing:** creating a cross-tab for fixed values only
 - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.
- **Rollup:** moving from finer-granularity data to a coarser granularity
- **Drill down:** The opposite operation - that of moving from coarser-granularity data to finer-granularity data

Reading material

Extended Aggregation to Support OLAP

- The **cube** operation computes union of **group by**'s on every subset of the specified attributes
Example relation for this section

sales(item_name, color, clothes_size, quantity)

- E.g. consider the query

```
select item_name, color, size, sum(number)  
from sales  
group by cube(item_name, color, size)
```

- This computes the union of eight different groupings of the *sales* relation:

```
{ (item_name, color, size), (item_name, color),  
(item_name, size), (color, size),  
(item_name), (color), (size) ( ) } , ( ) }
```

- where () denotes an empty **group by** list.
- For each grouping, the result contains the null value for attributes not present in the grouping.

Online Analytical Processing Operations

- Relational representation of cross-tab that we saw earlier, but with *null* in place of **all**, can be computed by

```
select item_name, color, sum(number)
      from sales
      group by cube(item_name, color)
```

- The function **grouping()** can be applied on an attribute
 - Returns 1 if the value is a null value representing all, and returns 0 in all other cases.

```
select item_name, color, size, sum(number),
       grouping(item_name) as item_name_flag,
       grouping(color) as color_flag,
       grouping(size) as size_flag,
      from sales
      group by cube(item_name, color, size)
```

Online Analytical Processing Operations

- Can use the function **decode()** in the **select** clause to replace such nulls by a value such as **all**
E.g., replace *item_name* in first query by
decode(grouping(item_name), 1, 'all', item_name)

Extended Aggregation (Cont.)

- The **rollup** construct generates union on every prefix of specified list of attributes
E.g.,

```
select item_name, color, size, sum(number)
from sales
group by rollup(item_name, color, size)
```

- Generates union of four groupings:
 $\{ (item_name, color, size), (item_name, color), (item_name), () \}$
- Rollup can be used to generate aggregates at multiple levels of a hierarchy.

E.g., suppose table *itemcategory*(*item_name*, *category*) gives the category of each item. Then,

```
select category, item_name, sum(number)
from sales, itemcategory
where sales.item_name = itemcategory.item_name
group by rollup(category, item_name)
```

would give a hierarchical summary by *item_name* and by *category*.

Extended Aggregation (Cont.)

- Multiple rollups and cubes can be used in a single group by clause
 - Each generates set of group by lists, cross product of sets gives overall set of group by lists
- E.g.,

```
select item name, color, size _name, color, size, sum(number)
from sales
group by rollup(item_name), rollup(color, size)
```

generates the groupings

```
{item_name, ()} X {(color, size), (color), ()}  
= { (item_name, color, size), (item_name, color),  
    (item_name), (color, size), (color), () }
```

OLAP Implementation

- The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems.
- OLAP implementations using only relational database features are called **relational OLAP (ROLAP)** systems
- Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems.

Procedural Constructs - Exception Handling

- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_classroom_seats condition  
  
declare exit handler for out_of_classroom_seats  
  
begin  
  
...  
  
.. signal out_of_classroom_seats  
  
end
```

- The handler here is **exit** -- causes enclosing **begin..end** to be exited
- Other actions possible on exception

Quiz

	popcorn	oil amt	batch	yield
1	plain	little	large	8.2
2	gourmet	little	large	8.6
3	plain	lots	large	10.4
4	gourmet	lots	large	9.2
5	plain	little	small	9.9
6	gourmet	little	small	12.1
7	plain	lots	small	10.6
8	gourmet	lots	small	18.0
9	plain	little	large	8.8
10	gourmet	little	large	8.2
11	plain	lots	large	8.8
12	gourmet	lots	large	9.8
13	plain	little	small	10.1
14	gourmet	little	small	15.9
15	plain	lots	small	7.4
16	gourmet	lots	small	16.0

Create the data cube for the following table

Explain how you find the average yield for plain popcorn with little oil amount.



Storage and File Structure Indexing and Hashing Query Processing and Optimization Transactions

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
 - data loss on power failure or system crash
 - physical failure of the storage device
- Can differentiate storage into:
 - **volatile storage:** loses contents when power is switched off
 - **non-volatile storage:**
 - ▶ Contents persist even when power is switched off.
 - ▶ Includes secondary and tertiary storage, as well as battery-backed up main-memory.



Physical Storage Media

- **Cache** – fastest and most costly form of storage; volatile; managed by the computer system hardware.
- **Main memory:**
 - fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
 - generally too small (or too expensive) to store the entire database
 - ▶ capacities of up to a few Gigabytes widely used currently
 - ▶ Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
- **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.



Physical Storage Media (Cont.)

□ Flash memory

- Data survives power failure
- Data can be written at a location only once, but location can be erased and written to again
 - ▶ Can support only a limited number (10K – 1M) of write/erase cycles.
 - ▶ Erasing of memory has to be done to an entire bank of memory
- Reads are roughly as fast as main memory
- But writes are slow (few microseconds), erase is slower
- Widely used in embedded devices such as digital cameras, phones, and USB keys



Physical Storage Media (Cont.)

□ Magnetic-disk

- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- Data must be moved from disk to main memory for access, and written back for storage
 - ▶ Much slower access than main memory (more on this later)
- **direct-access** – possible to read data on disk in any order, unlike magnetic tape
- Capacities range up to roughly 1.5 TB as of 2009
 - ▶ Much larger capacity and cost/byte than main memory/flash memory
 - ▶ Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
- Survives power failures and system crashes
 - ▶ disk failure can destroy data, but is rare



Physical Storage Media (Cont.)

□ Optical storage

- non-volatile, data is read optically from a spinning disk using a laser
- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- Blu-ray disks: 27 GB to 54 GB
- Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
- Reads and writes are slower than with magnetic disk
- **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data



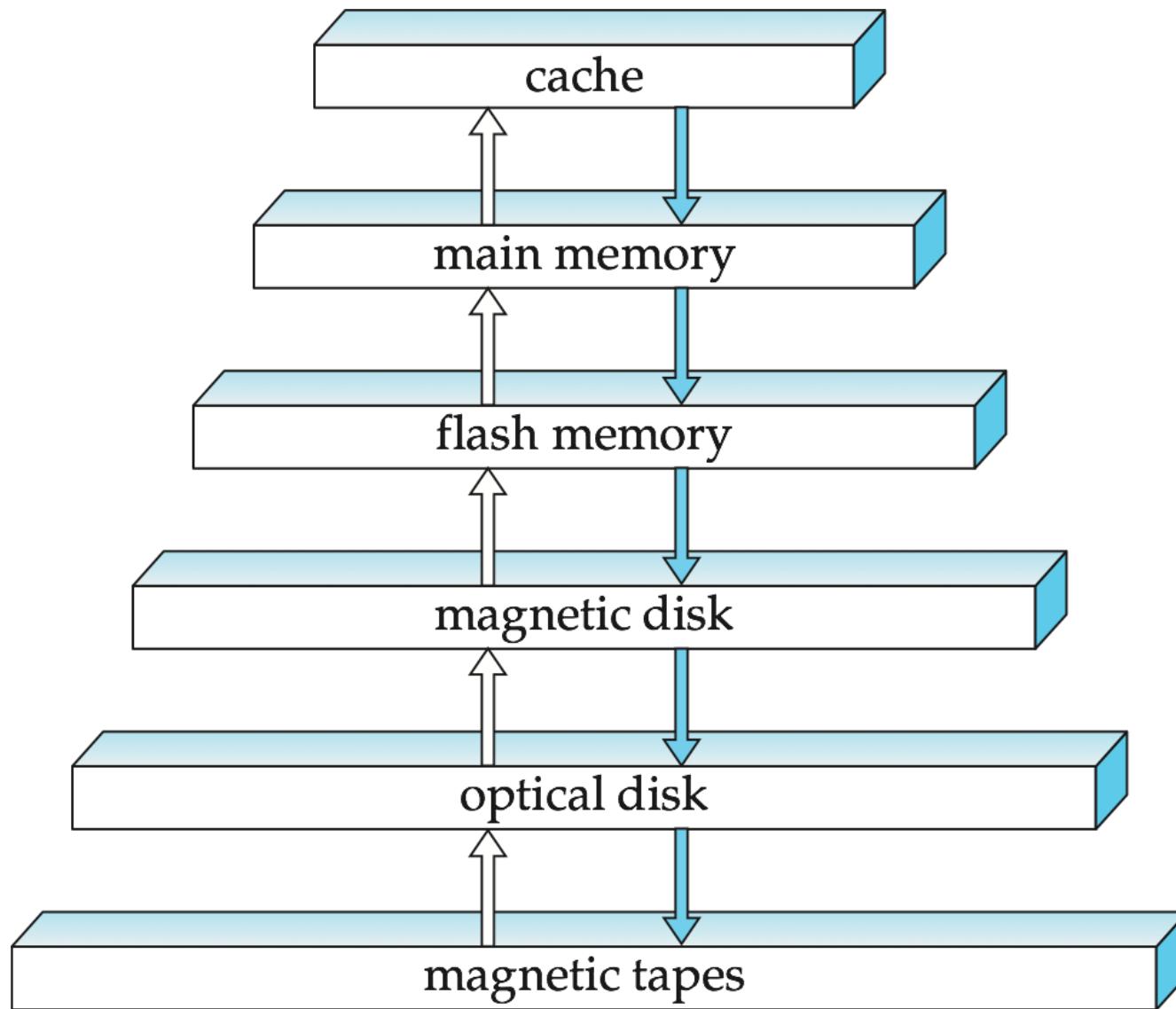
Physical Storage Media (Cont.)

□ Tape storage

- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** – much slower than disk
- very high capacity (40 to 300 GB tapes available)
- tape can be removed from drive \Rightarrow storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
 - ▶ hundreds of terabytes (1 terabyte = 10^9 bytes) to even multiple **petabytes** (1 petabyte = 10^{12} bytes)



Storage Hierarchy



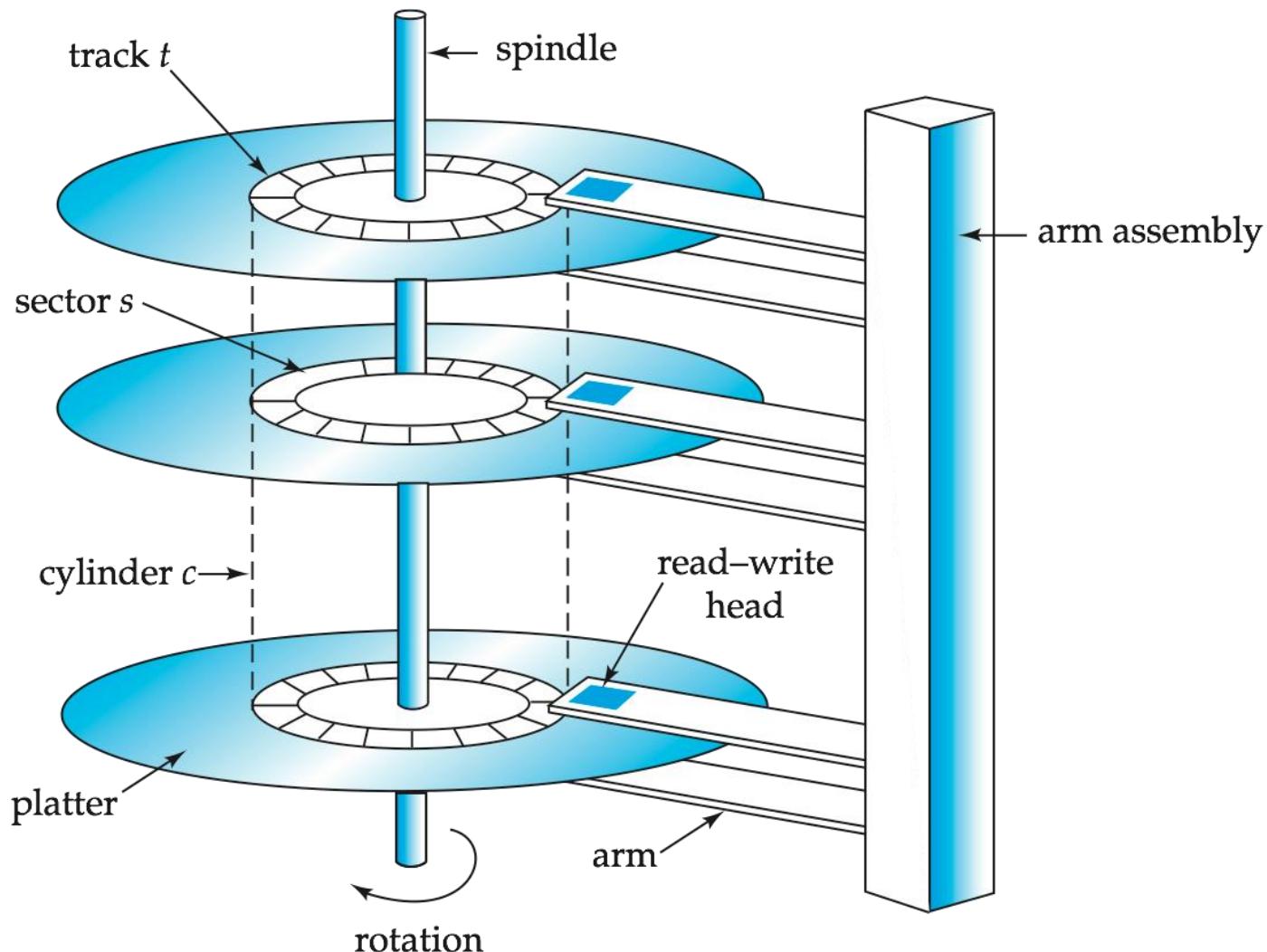


Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
 - also called **on-line storage**
 - E.g. flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
 - also called **off-line storage**
 - E.g. magnetic tape, optical storage



Magnetic Hard Disk Mechanism



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives



Magnetic Disks

- **Read-write head**
 - Positioned very close to the platter surface (almost touching it)
 - Reads or writes magnetically encoded information.
- Surface of platter divided into circular **tracks**
 - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**.
 - A sector is the smallest unit of data that can be read or written.
 - Sector size typically 512 bytes
 - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
 - disk arm swings to position head on right track
 - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
 - multiple disk platters on a single spindle (1 to 5 usually)
 - one head per platter, mounted on a common arm.
- **Cylinder** i consists of i^{th} track of all the platters



Magnetic Disks (Cont.)

- Earlier generation disks were susceptible to head-crashes
 - Surface of earlier generation disks had metal-oxide coatings which would disintegrate on head crash and damage all data on disk
 - Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted
- **Disk controller** – interfaces between the computer system and the disk drive hardware.
 - accepts high-level commands to read or write a sector
 - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
 - Computes and attaches **checksums** to each sector to verify that data is read back correctly
 - ▶ If data is corrupted, with very high probability stored checksum won't match recomputed checksum
 - Ensures successful writing by reading back sector after writing it
 - Performs remapping of bad sectors



Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
 - **Seek time** – time it takes to reposition the arm over the correct track.
 - ▶ Average seek time is 1/2 the worst case seek time.
 - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
 - ▶ 4 to 10 milliseconds on typical disks
 - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
 - ▶ Average latency is 1/2 of the worst case latency.
 - ▶ 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
 - 25 to 100 MB per second max rate, lower for inner tracks
 - Multiple disks may share a controller, so rate that controller can handle is also important
 - ▶ E.g. SATA: 150 MB/sec, SATA-II 3Gb (300 MB/sec)
 - ▶ Ultra 320 SCSI: 320 MB/s, SAS (3 to 6 Gb/sec)
 - ▶ Fiber Channel (FC2Gb or 4Gb): 256 to 512 MB/s



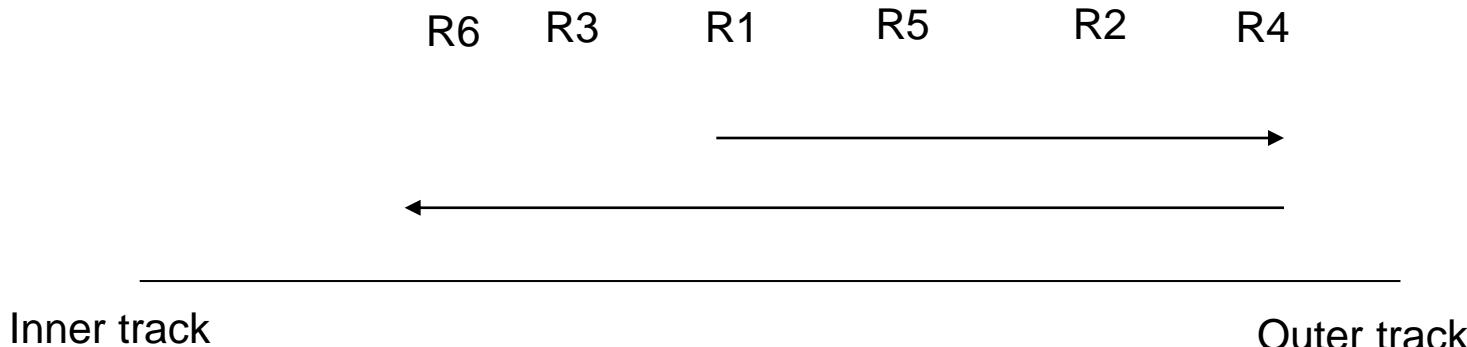
Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
 - Typically 3 to 5 years
 - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
 - ▶ E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
 - MTTF decreases as disk ages



Optimization of Disk-Block Access

- **Block** – a contiguous sequence of sectors from a single track
 - data is transferred between disk and main memory in blocks
 - sizes range from 512 bytes to several kilobytes
 - ▶ Smaller blocks: more transfers from disk
 - ▶ Larger blocks: more space wasted due to partially filled blocks
 - ▶ Typical block sizes today range from 4 to 16 kilobytes
- **Disk-arm-scheduling** algorithms order pending accesses to tracks so that disk arm movement is minimized
 - **elevator algorithm:**





Optimization of Disk Block Access (Cont.)

- **File organization** – optimize block access time by organizing the blocks to correspond to how data will be accessed
 - E.g. Store related information on the same or nearby cylinders.
 - Files may get **fragmented** over time
 - ▶ E.g. if data is inserted to/deleted from the file
 - ▶ Or free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
 - ▶ Sequential access to a fragmented file results in increased disk arm movement
 - Some systems have utilities to **defragment** the file system, in order to speed up file access



Optimization of Disk Block Access (Cont.)

- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM buffer immediately
 - Non-volatile RAM: battery backed up RAM or flash memory
 - Even if power fails, the data is safe and will be written to disk when power returns
 - Controller then writes to disk whenever the disk has no other requests or request has been pending for some time
 - Database operations that require data to be safely stored before continuing can continue without waiting for data to be written to disk
 - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
 - Used exactly like nonvolatile RAM
 - Write to log disk is very fast since no seeks are required
 - No need for special hardware (NV-RAM)
- File systems typically reorder writes to disk to improve performance
 - **Journaling file systems** write data in safe order to NV-RAM or log disk
 - Reordering without journaling: risk of corruption of file system data



Flash Storage

- used widely for storage
- requires page-at-a-time read (page: 512 bytes to 4 KB)
- transfer rate around 20 MB/sec
- **solid state disks**: use multiple flash storage devices to provide higher transfer rate of 100 to 200 MB/sec
- erase is very slow (1 to 2 millisecs)
 - ▶ erase block contains multiple pages
 - ▶ after 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used



RAID

- **RAID: Redundant Arrays of Independent Disks**
 - disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
 - ▶ **high capacity** and **high speed** by using multiple disks in parallel,
 - ▶ **high reliability** by storing data redundantly, so that data can be recovered even if a disk fails
- The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail.
 - E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
 - Techniques for using redundancy to avoid data loss are critical with large numbers of disks
- Originally a cost-effective alternative to large, expensive disks
 - I in RAID originally stood for ``inexpensive''
 - Today RAIDs are used for their higher reliability and bandwidth.
 - ▶ The “I” is interpreted as independent



Improvement of Reliability via Redundancy

- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**)
 - Duplicate every disk. Logical disk consists of two physical disks.
 - Every write is carried out on both disks
 - ▶ Reads can take place from either disk
 - If one disk in a pair fails, data still available in the other
 - ▶ Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
 - Probability of combined event is very small
 - » Except for dependent failure modes such as fire or building collapse or electrical power surges



Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
 1. Load balance multiple small accesses to increase throughput
 2. Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
- **Bit-level striping** – split the bits of each byte across multiple disks
 - In an array of eight disks, write bit i of each byte to disk i .
 - Each access can read data at eight times the rate of a single disk.
 - But seek/access time worse than for a single disk
 - ▶ Bit level striping is not used much any more
- **Block-level striping** – with n disks, block i of a file goes to disk $(i \bmod n) + 1$
 - Requests for different blocks can run in parallel if the blocks reside on different disks
 - A request for a long sequence of blocks can utilize all disks in parallel

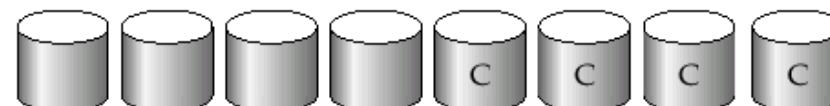


RAID Levels

- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
 - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- **RAID Level 0:** Block striping; non-redundant.
 - Used in high-performance applications where data loss is not critical.
- **RAID Level 1:** Mirrored disks with block striping
 - Offers best write performance.
 - Popular for applications such as storing log files in a database system.



(a) RAID 0: nonredundant striping

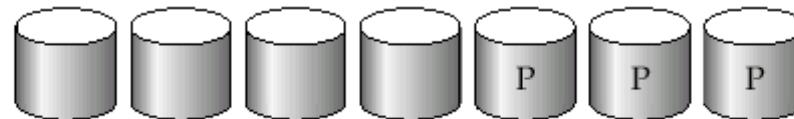


(b) RAID 1: mirrored disks

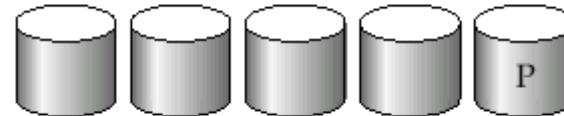


RAID Levels (Cont.)

- **RAID Level 2:** Memory-Style Error-Correcting-Codes (ECC) with bit striping.
- **RAID Level 3:** Bit-Interleaved Parity
 - a single parity bit is enough for error correction, not just detection, since we know which disk has failed
 - ▶ When writing data, corresponding parity bits must also be computed and written to a parity bit disk
 - ▶ To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)



(c) RAID 2: memory-style error-correcting codes



(d) RAID 3: bit-interleaved parity



RAID Levels (Cont.)

- RAID Level 3 (Cont.)

- Faster data transfer than with a single disk, but fewer I/Os per second since every disk has to participate in every I/O.
 - Subsumes Level 2 (provides all its benefits, at lower cost).

- RAID Level 4: Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from N other disks.

- When writing data block, corresponding block of parity bits must also be computed and written to parity disk
 - To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.



(e) RAID 4: block-interleaved parity



RAID Levels (Cont.)

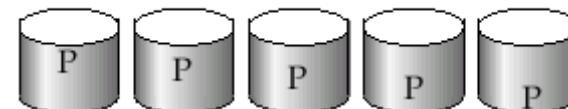
□ RAID Level 4 (Cont.)

- Provides higher I/O rates for independent block reads than Level 3
 - ▶ block read goes to a single disk, so blocks stored on different disks can be read in parallel
- Provides high transfer rates for reads of multiple blocks than no-striping
- Before writing a block, parity data must be computed
 - ▶ Can be done by using old parity block, old value of current block and new value of current block (2 block reads + 2 block writes)
 - ▶ Or by recomputing the parity value using the new values of blocks corresponding to the parity block
 - More efficient for writing large amounts of data sequentially
- Parity block becomes a bottleneck for independent block writes since every block write also writes to parity disk



RAID Levels (Cont.)

- **RAID Level 5:** Block-Interleaved Distributed Parity; partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.
 - E.g., with 5 disks, parity block for n th set of blocks is stored on disk $(n \bmod 5) + 1$. with the data blocks stored on the other 4 disks.



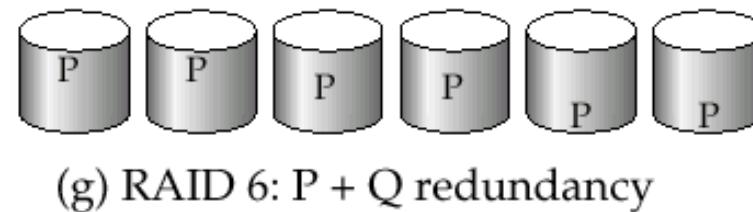
(f) RAID 5: block-interleaved distributed parity

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4



RAID Levels (Cont.)

- RAID Level 5 (Cont.)
 - Higher I/O rates than Level 4.
 - ▶ Block writes occur in parallel if the blocks and their parity blocks are on different disks.
 - Subsumes Level 4: provides same benefits, but avoids bottleneck of parity disk.
- RAID Level 6: P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
 - Better reliability than Level 5 at a higher cost; not used as widely.



(g) RAID 6: P + Q redundancy



Choice of RAID Level

- Factors in choosing RAID level
 - Monetary cost
 - Performance: Number of I/O operations per second, and bandwidth during normal operation
 - Performance during failure
 - Performance during rebuild of failed disk
 - ▶ Including time taken to rebuild failed disk
- RAID 0 is used only when data safety is not important
 - E.g. data can be recovered quickly from other sources
- Level 2 and 4 never used since they are subsumed by 3 and 5
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (level 5) avoids
- Level 6 is rarely used since levels 1 and 5 offer adequate safety for most applications



Choice of RAID Level (Cont.)

- Level 1 provides much better write performance than level 5
 - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
 - Level 1 preferred for high update environments such as log disks
- Level 1 had higher storage cost than level 5
 - disk drive capacities increasing rapidly (50%/year) whereas disk access times have decreased much less (x 3 in 10 years)
 - I/O requirements have increased greatly, e.g. for Web servers
 - When enough disks have been bought to satisfy required rate of I/O, they often have spare storage capacity
 - ▶ so there is often no extra monetary cost for Level 1!
- Level 5 is preferred for applications with low update rate, and large amounts of data
- Level 1 is preferred for all other applications



Optical Disks

- Compact disk-read only memory (CD-ROM)
 - Removable disks, 640 MB per disk
 - Seek time about 100 msec (optical read head is heavier and slower)
 - Higher latency (3000 RPM) and lower data-transfer rates (3-6 MB/s) compared to magnetic disks
- Digital Video Disk (DVD)
 - DVD-5 holds 4.7 GB , and DVD-9 holds 8.5 GB
 - DVD-10 and DVD-18 are double sided formats with capacities of 9.4 GB and 17 GB
 - Blu-ray DVD: 27 GB (54 GB for double sided disk)
 - Slow seek time, for same reasons as CD-ROM
- Record once versions (CD-R and DVD-R) are popular
 - data can only be written once, and cannot be erased.
 - high capacity and long lifetime; used for archival storage
 - Multi-write versions (CD-RW, DVD-RW, DVD+RW and DVD-RAM) also available



Magnetic Tapes

- Hold large volumes of data and provide high transfer rates
 - Few GB for DAT (Digital Audio Tape) format, 10-40 GB with DLT (Digital Linear Tape) format, 100 GB+ with Ultrium format, and 330 GB with Ampex helical scan format
 - Transfer rates from few to 10s of MB/s
- Tapes are cheap, but cost of drives is very high
- Very slow access time in comparison to magnetic and optical disks
 - limited to sequential access.
 - Some formats (Accelis) provide faster seek (10s of seconds) at cost of lower capacity
- Used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another.
- Tape jukeboxes used for very large capacity storage
 - Multiple petabytes (10^{15} bytes)



File Organization, Record Organization and Storage Access



File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.
 - One approach:
 - assume record size is fixed
 - each file has records of one particular type only
 - different files are used for different relations
- This case is easiest to implement; will consider variable length records later.



Fixed-Length Records

- Simple approach:
 - Store record i starting from byte $n * (i - 1)$, where n is the size of each record.
 - Record access is simple but records may cross blocks
 - ▶ Modification: do not allow records to cross block boundaries

- Deletion of record i : alternatives:
 - move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 - move record n to i
 - do not move records, but link all free records on a *free list*

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



Deleting record 3 and compacting

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



Deleting record 3 and moving last record

record 0
record 1
record 2
record 11
record 4
record 5
record 6
record 7
record 8
record 9
record 10

	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000



Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as **pointers** since they “point” to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)

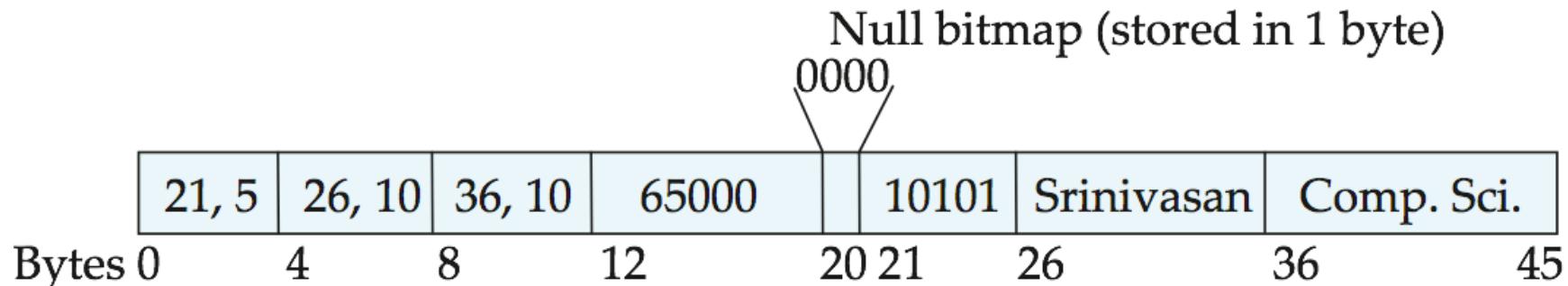
header			
record 0	10101	Srinivasan	Comp. Sci. 65000
record 1			
record 2	15151	Mozart	Music 40000
record 3	22222	Einstein	Physics 95000
record 4			
record 5	33456	Gold	Physics 87000
record 6			
record 7	58583	Califieri	History 62000
record 8	76543	Singh	Finance 80000
record 9	76766	Crick	Biology 72000
record 10	83821	Brandt	Comp. Sci. 92000
record 11	98345	Kim	Elec. Eng. 80000

The diagram illustrates the use of free lists. It shows a table with 12 records. Records 0, 2, 3, 5, 7, 8, 9, 10, and 11 contain data. Records 1, 4, and 6 are empty. Arrows point from the fourth column of records 1, 4, and 6 to the first column of record 5, indicating that record 5 is used to store pointers to the locations of deleted records.



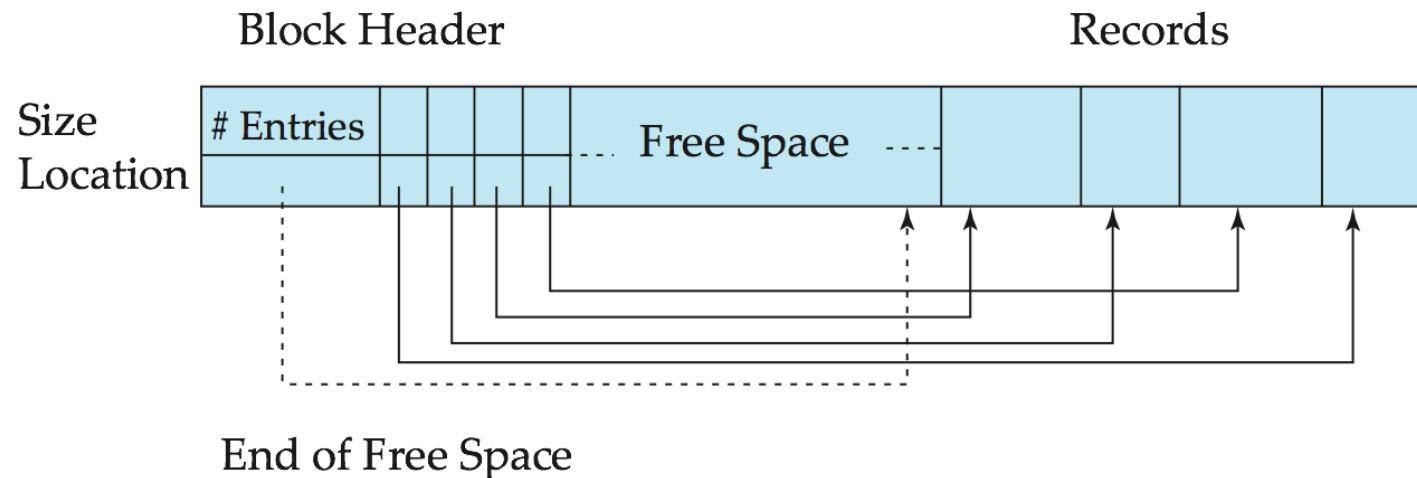
Variable-Length Records

- Variable-length records arise in database systems in several ways:
 - Storage of multiple record types in a file.
 - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
 - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap





Variable-Length Records: Slotted Page Structure



- **Slotted page** header contains:
 - number of record entries
 - end of free space in the block
 - location and size of each record
 - Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
 - Pointers should not point directly to record — instead they should point to the entry for the record in header.



Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file
 - Motivation: store related records on the same block to minimize I/O



Sequential File Organization

- ❑ Suitable for applications that require sequential processing of the entire file
- ❑ The records in the file are ordered by a **search-key**

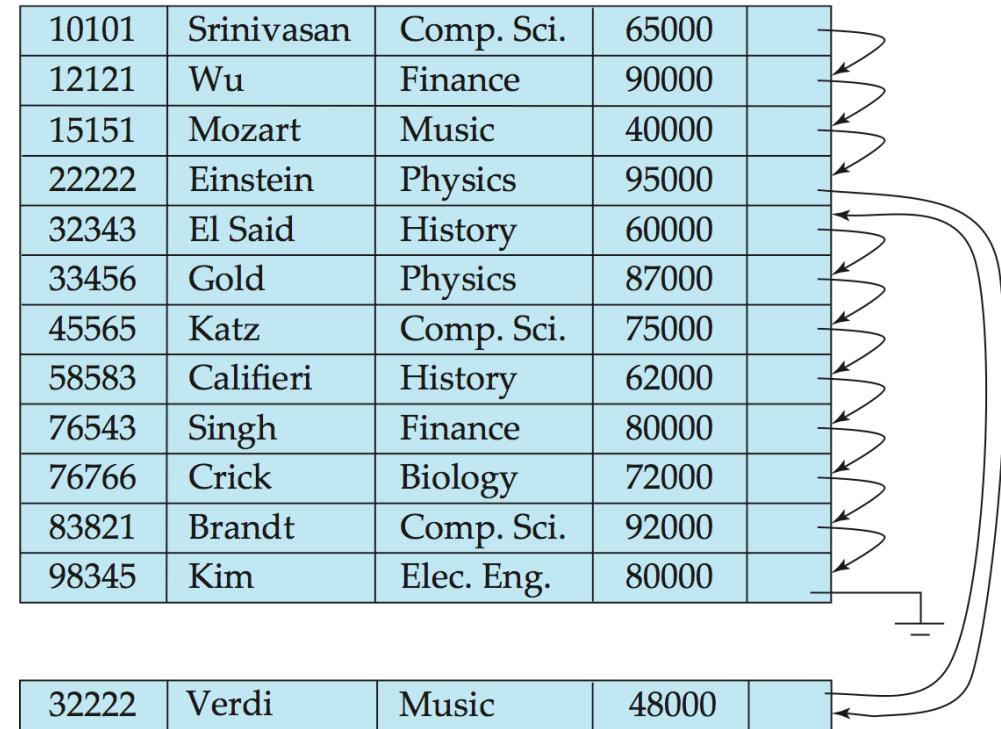
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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

A vertical line of ten curved arrows points from the right edge of the table back towards the left, indicating the sequential reading of the records from bottom to top.



Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion – locate the position where the record is to be inserted
 - if there is free space insert there
 - if no free space, insert the record in an **overflow block**
 - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order





Multitable Clustering File Organization

Store several relations in one file using a **multitable clustering** file organization

department

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

multitable clustering
of *department* and
instructor

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000



Multitable Clustering File Organization (cont.)

- good for queries involving *department* \bowtie *instructor*, and for queries involving one single department and its instructors
- bad for queries involving only *department*
- results in variable size records
- Can add pointer chains to link records of a particular relation

Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	





Data Dictionary Storage

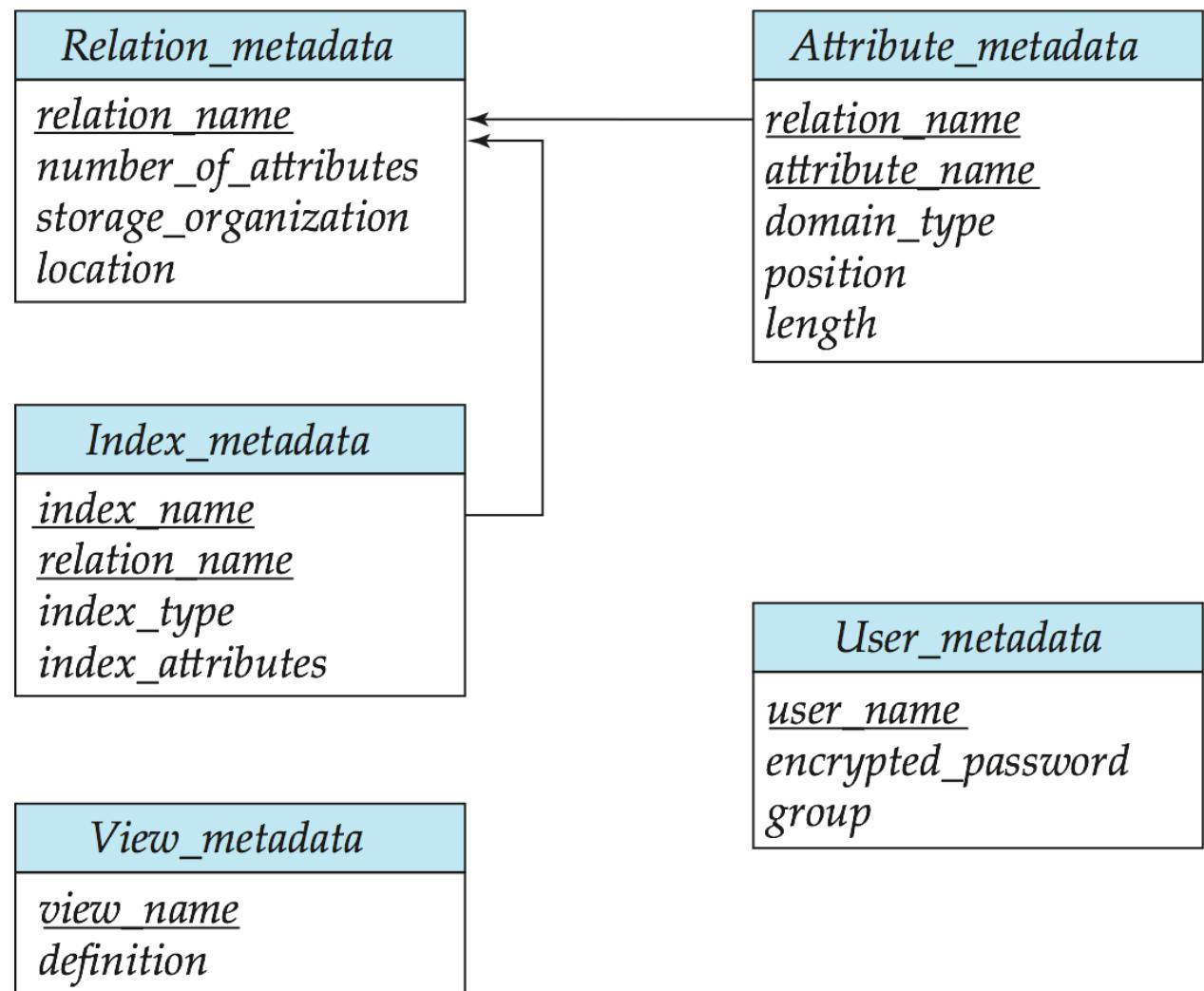
The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

- Information about relations
 - names of relations
 - names, types and lengths of attributes of each relation
 - names and definitions of views
 - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
 - number of tuples in each relation
- Physical file organization information
 - How relation is stored (sequential/hash/...)
 - Physical location of relation
- Information about indices



Relational Representation of System Metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory





Storage Access

- A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.



Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
 1. If the block is already in the buffer, buffer manager returns the address of the block in main memory
 2. If the block is not in the buffer, the buffer manager
 1. Allocates space in the buffer for the block
 1. Replacing (throwing out) some other block, if required, to make space for the new block.
 2. Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
 2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.



Buffer-Replacement Policies

- Most operating systems replace the block **least recently used (LRU strategy)**
- Idea behind LRU – use past pattern of block references as a predictor of future references
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
 - LRU can be a bad strategy for certain access patterns involving repeated scans of data
 - ▶ For example: when computing the join of 2 relations r and s by a nested loops
 - for each tuple tr of r do
 - for each tuple ts of s do
 - if the tuples tr and ts match ...
 - Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable



Buffer-Replacement Policies (Cont.)

- **Pinned block** – memory block that is not allowed to be written back to disk.
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
 - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Buffer managers also support **forced output** of blocks for the purpose of recovery



Introduction to Indexing and Hashing



Indexing: Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer
------------	---------

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
 - **Ordered indices:** search keys are stored in sorted order
 - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.



Index Evaluation Metrics

- Access types supported efficiently. E.g.,
 - records with a specified value in the attribute
 - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead



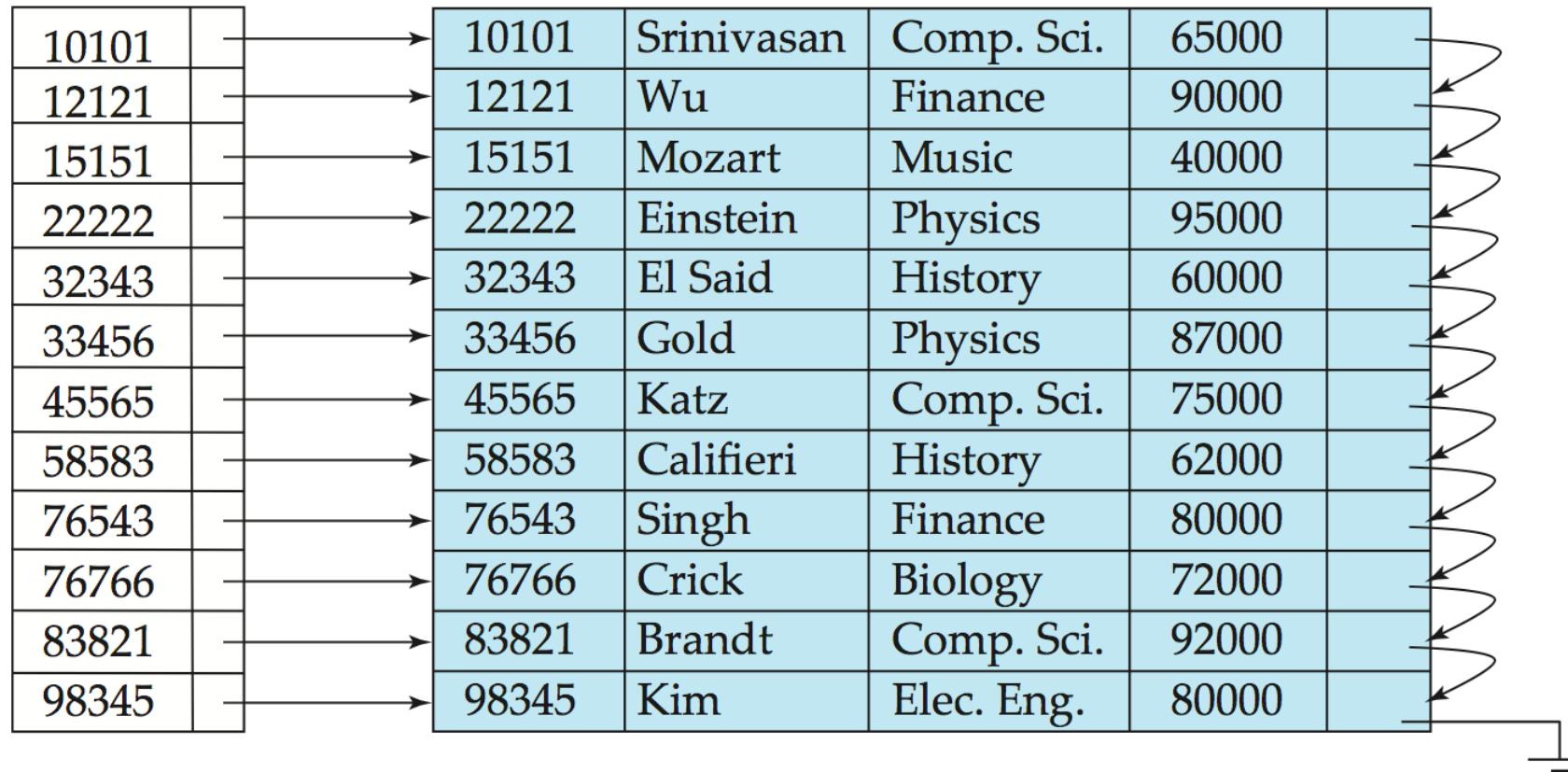
Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **clustering index**
 - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.
- **Index-sequential file**: ordered sequential file with a primary index.



Dense Index Files

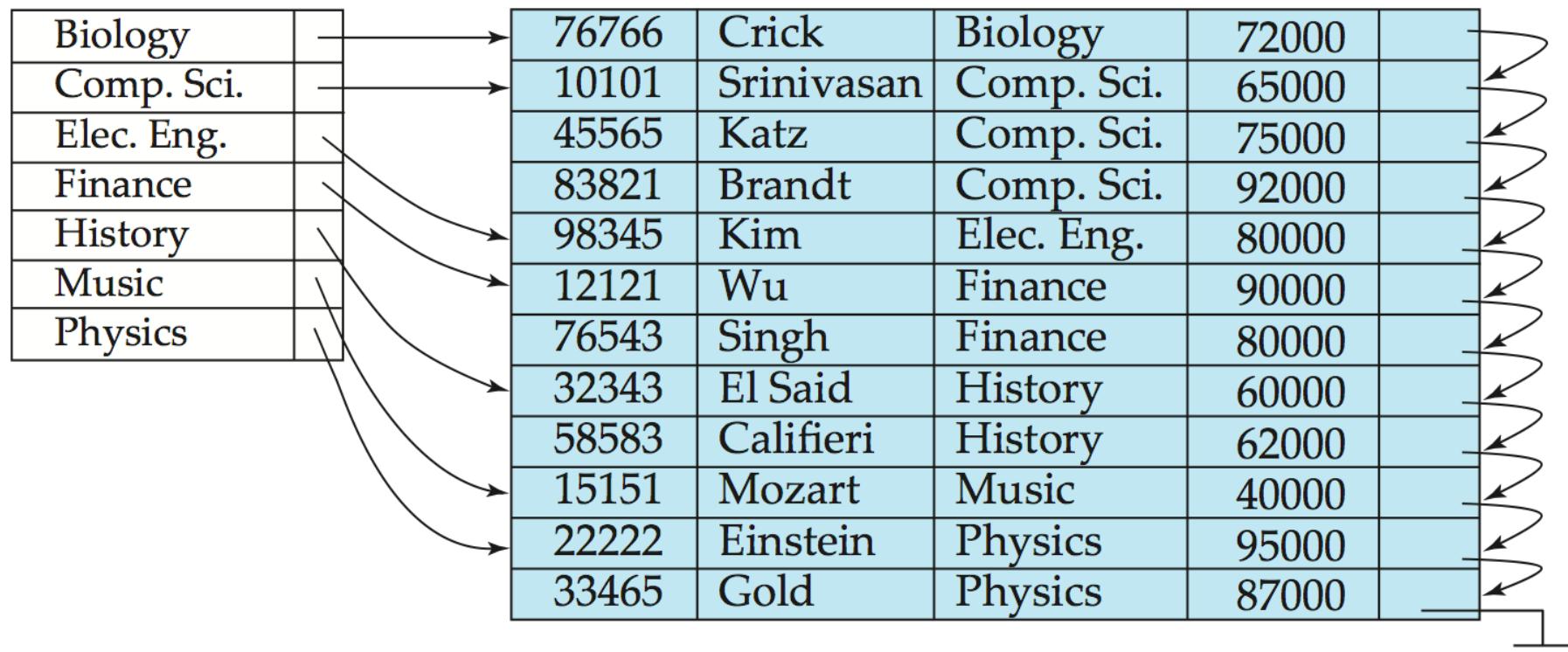
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation





Dense Index Files (Cont.)

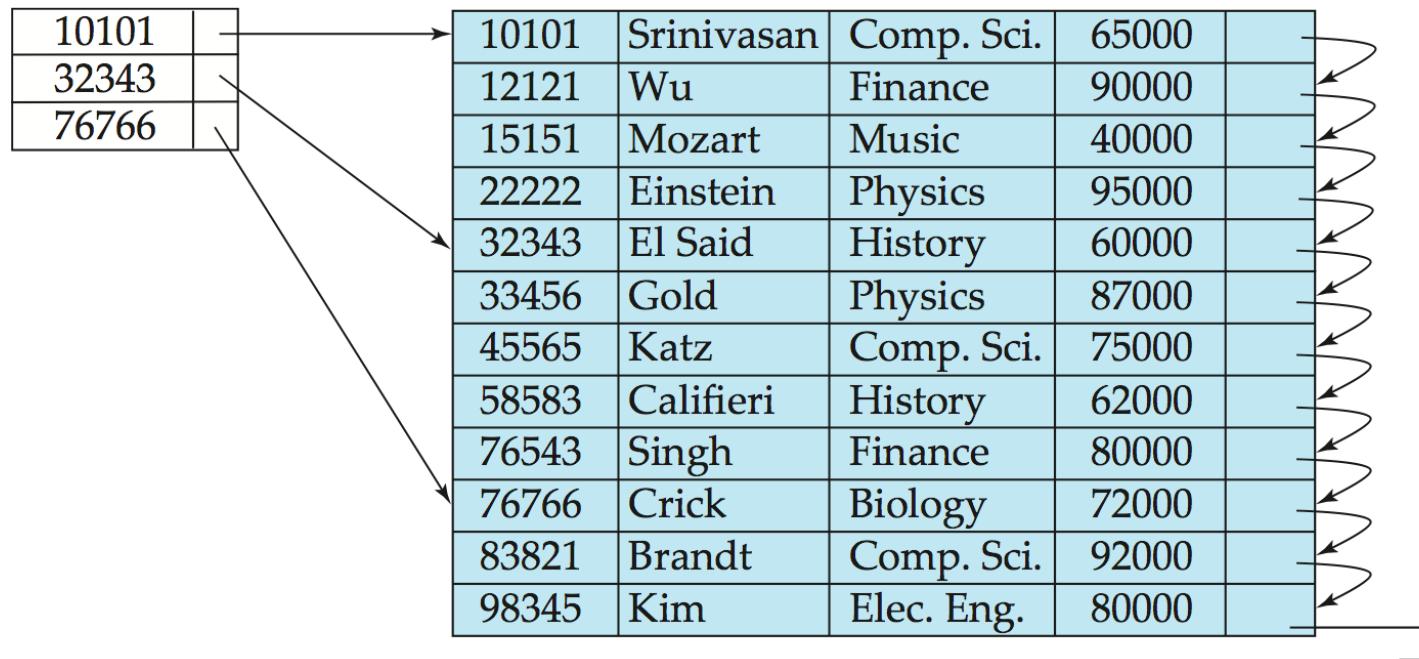
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*





Sparse Index Files

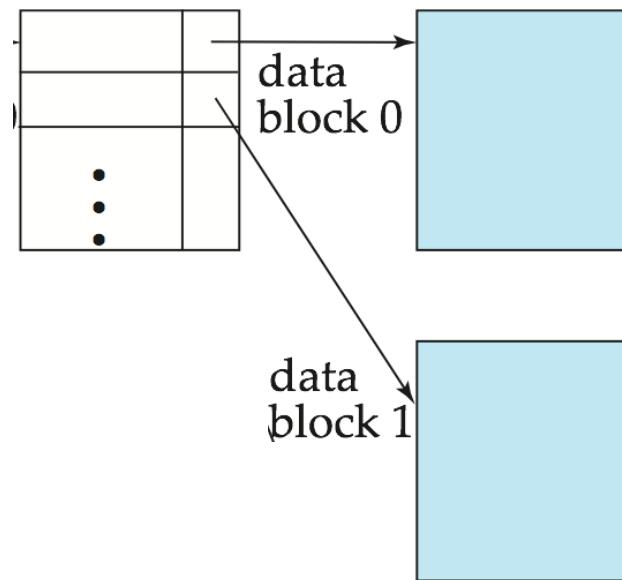
- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to which the index record points





Sparse Index Files (Cont.)

- Compared to dense indices:
 - Less space and less maintenance overhead for insertions and deletions.
 - Generally slower than dense index for locating records.
- **Good tradeoff:** sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



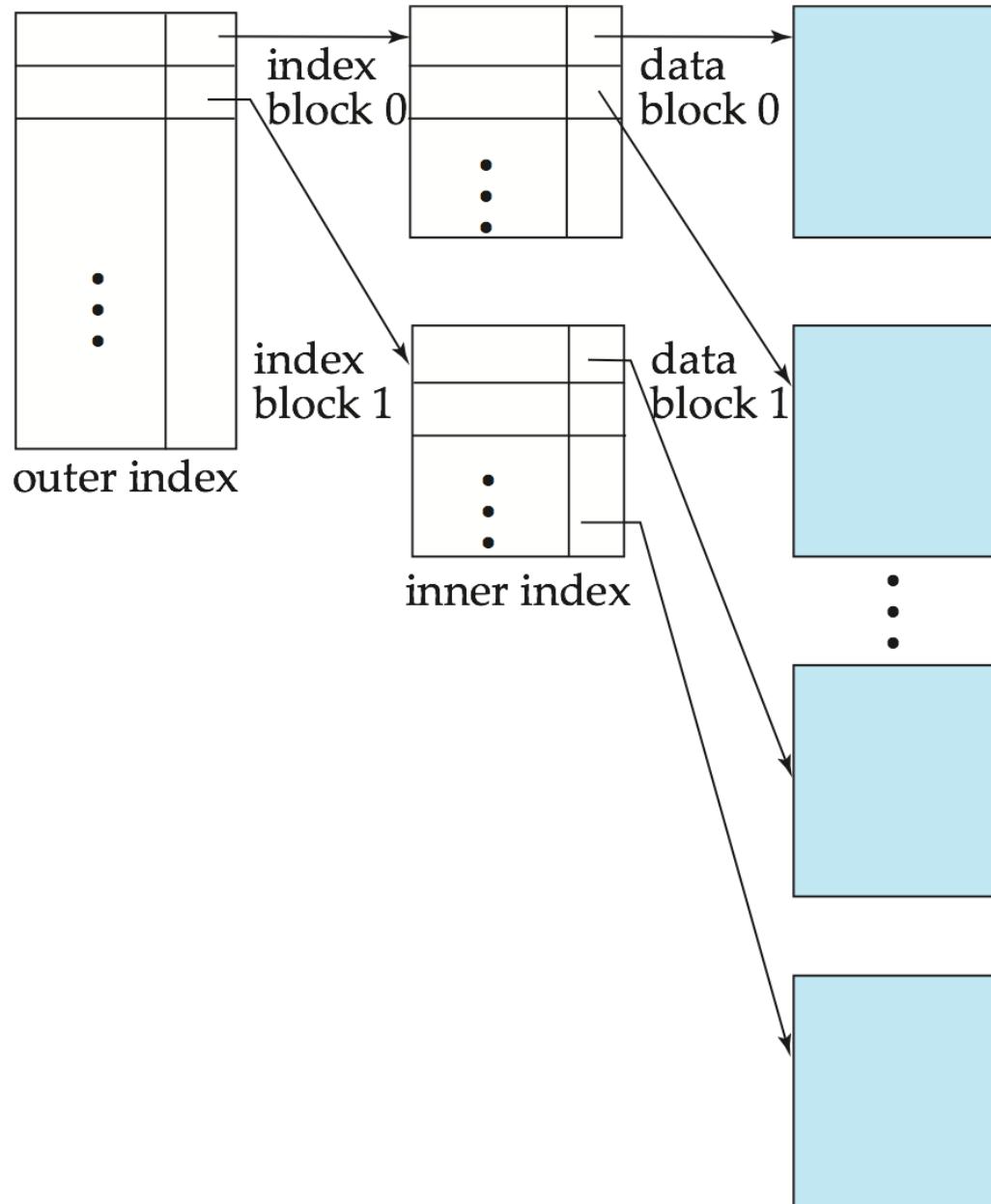


Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
 - outer index – a sparse index of primary index
 - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



Multilevel Index (Cont.)



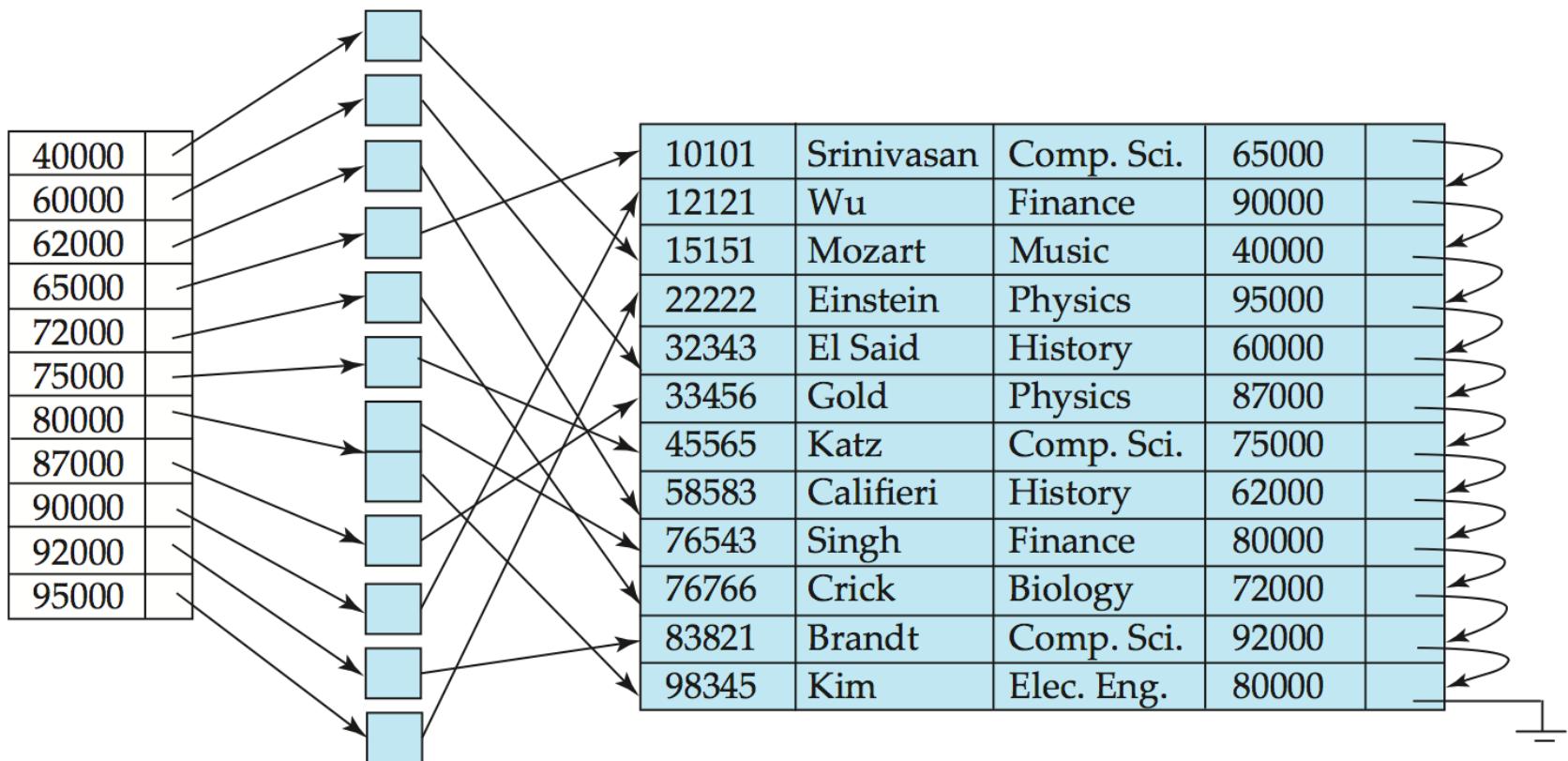


Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
 - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
 - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value



Secondary Indices Example



- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense



Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
 - Each record access may fetch a new block from disk
 - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access



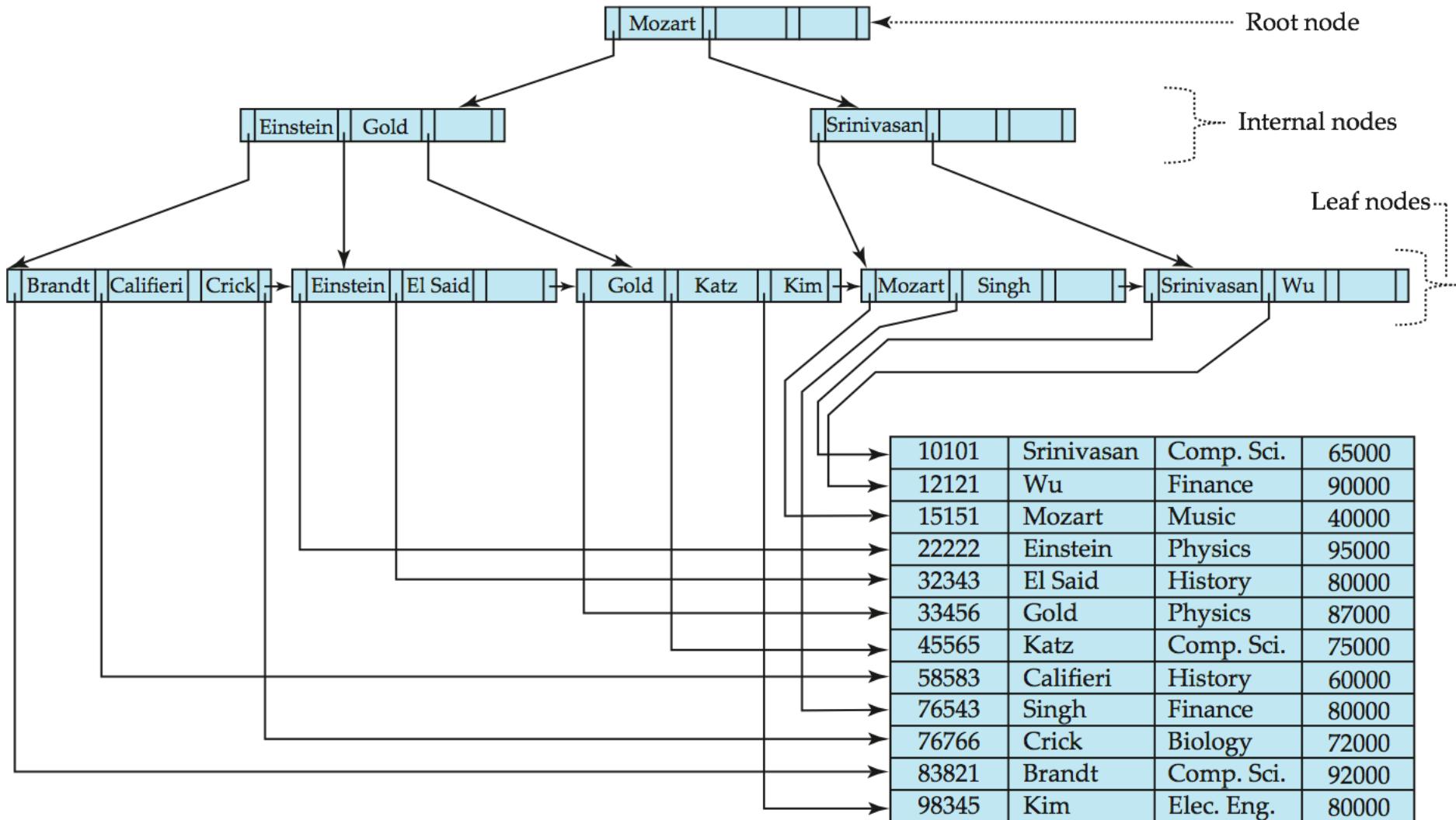
B⁺-Tree Index Files

B⁺-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
 - performance degrades as file grows, since many overflow blocks get created.
 - Periodic reorganization of entire file is required.
- Advantage of B⁺-tree index files:
 - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
 - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B⁺-trees:
 - extra insertion and deletion overhead, space overhead.
- Advantages of B⁺-trees outweigh disadvantages
 - B⁺-trees are used extensively



Example of B+-Tree





Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**.
- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B .
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.



Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key
(See figure in next slide.)

- There are 10 buckets,
- The binary representation of the i th character is assumed to be the integer i .
- The hash function returns the sum of the binary representations of the characters modulo 10
 - E.g. $h(\text{Music}) = 1 \quad h(\text{History}) = 2$
 $h(\text{Physics}) = 3 \quad h(\text{Elec. Eng.}) = 3$



Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key
(see previous slide for details).



Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
 - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .



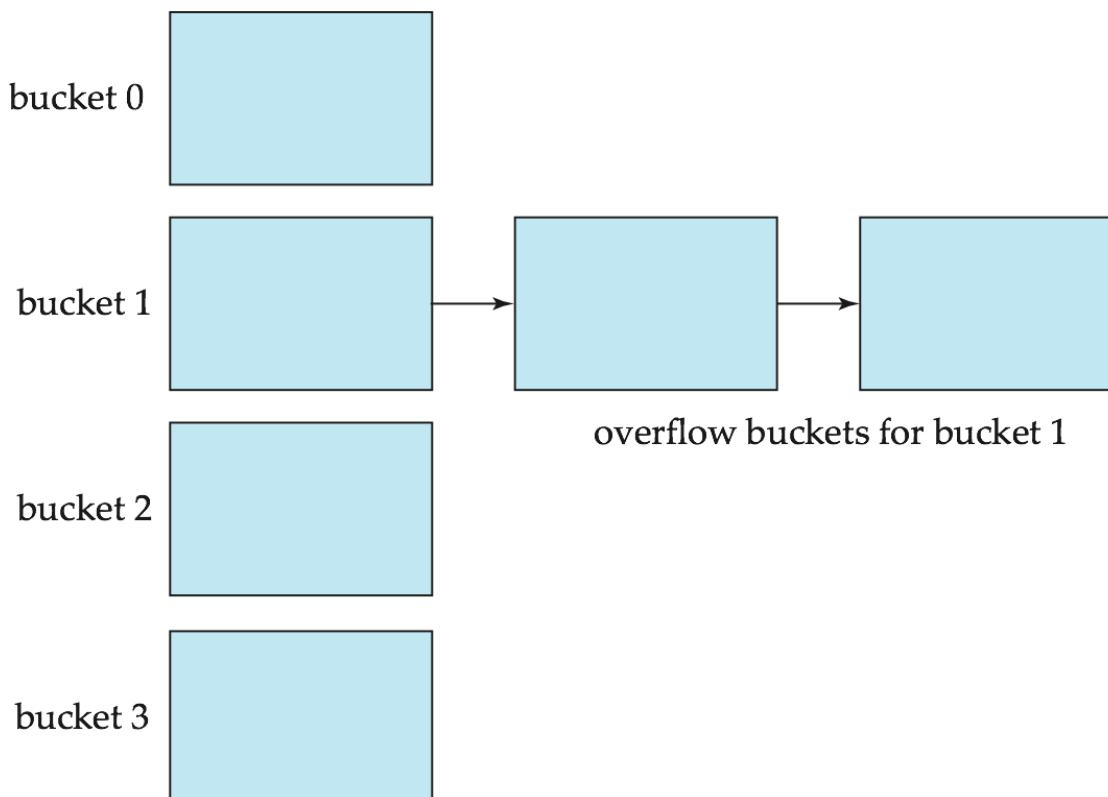
Handling of Bucket Overflows

- Bucket overflow can occur because of
 - Insufficient buckets
 - Skew in distribution of records. This can occur due to two reasons:
 - ▶ multiple records have same search-key value
 - ▶ chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***.



Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
 - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.





Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
 - Given a number n it must be easy to retrieve record n
 - ▶ Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
 - E.g. gender, country, state, ...
 - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)
- A bitmap is simply an array of bits



Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
 - Bitmap has as many bits as records
 - In a bitmap for value v , the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	<i>ID</i>	<i>gender</i>	<i>income_level</i>
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

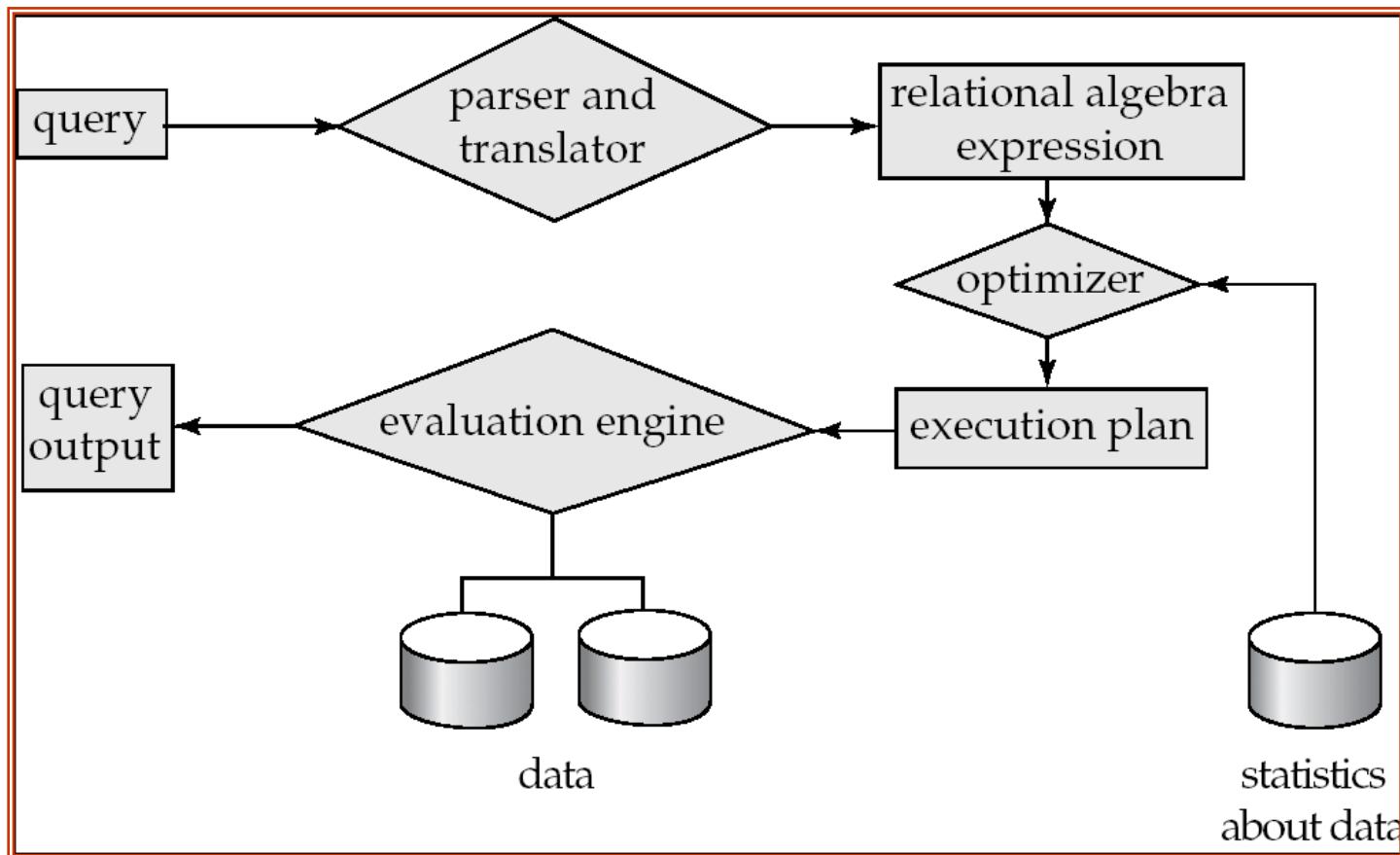


Introduction to Query Processing and Optimization



Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





Basic Steps in Query Processing (Cont.)

- Parsing and translation
 - translate the query into its internal form. This is then translated into relational algebra.
 - Parser checks syntax, verifies relations
- Evaluation
 - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Basic Steps in Query Processing : Optimization (example)

```
select balance  
from account  
where balance < 2500
```

- A relational algebra expression may have many equivalent expressions
 - E.g., $\sigma_{balance < 2500}(\Pi_{balance}(account))$ is equivalent to
$$\Pi_{balance}(\sigma_{balance < 2500}(account))$$
- Each relational algebra operation can be evaluated using one of several different algorithms
 - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
 - E.g., can use an index on *balance* to find accounts with $balance < 2500$,
 - or can perform complete relation scan and discard accounts with $balance \geq 2500$



Basic Steps: Optimization (Cont.)

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
 - Cost is estimated using statistical information from the database catalog
 - ▶ e.g. number of tuples in each relation, size of tuples, etc.



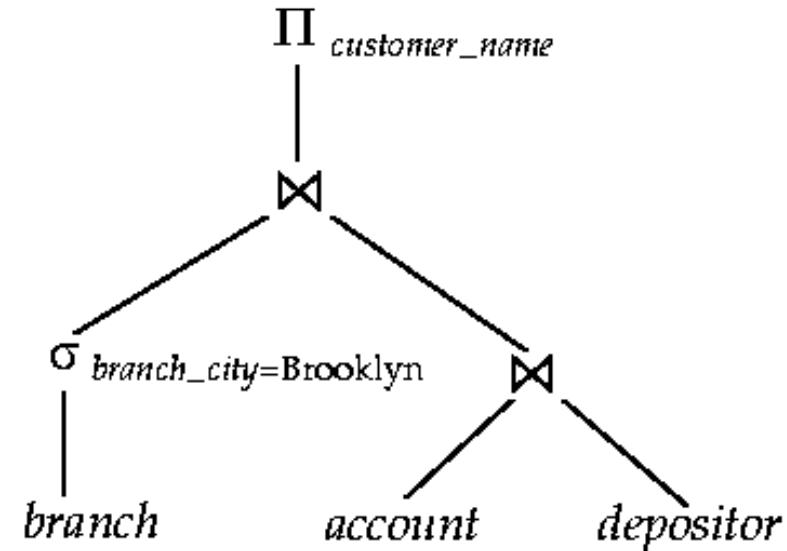
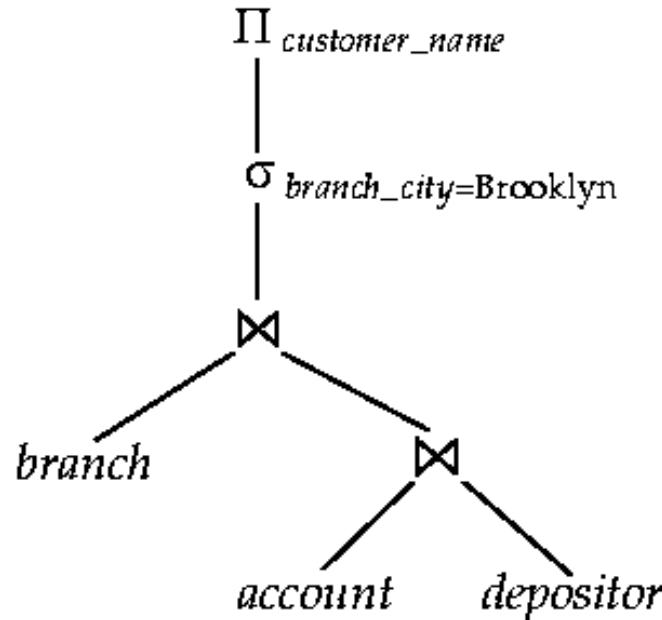
Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
 - Many factors contribute to time cost
 - ▶ disk accesses, CPU, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
 - Number of seeks * average-seek-cost
 - Number of blocks read * average-block-read-cost
 - Number of blocks written * average-block-write-cost
 - ▶ Cost to write a block is greater than cost to read a block
 - data is read back after being written to ensure that the write was successful



Alternative Query Expressions (ex.)

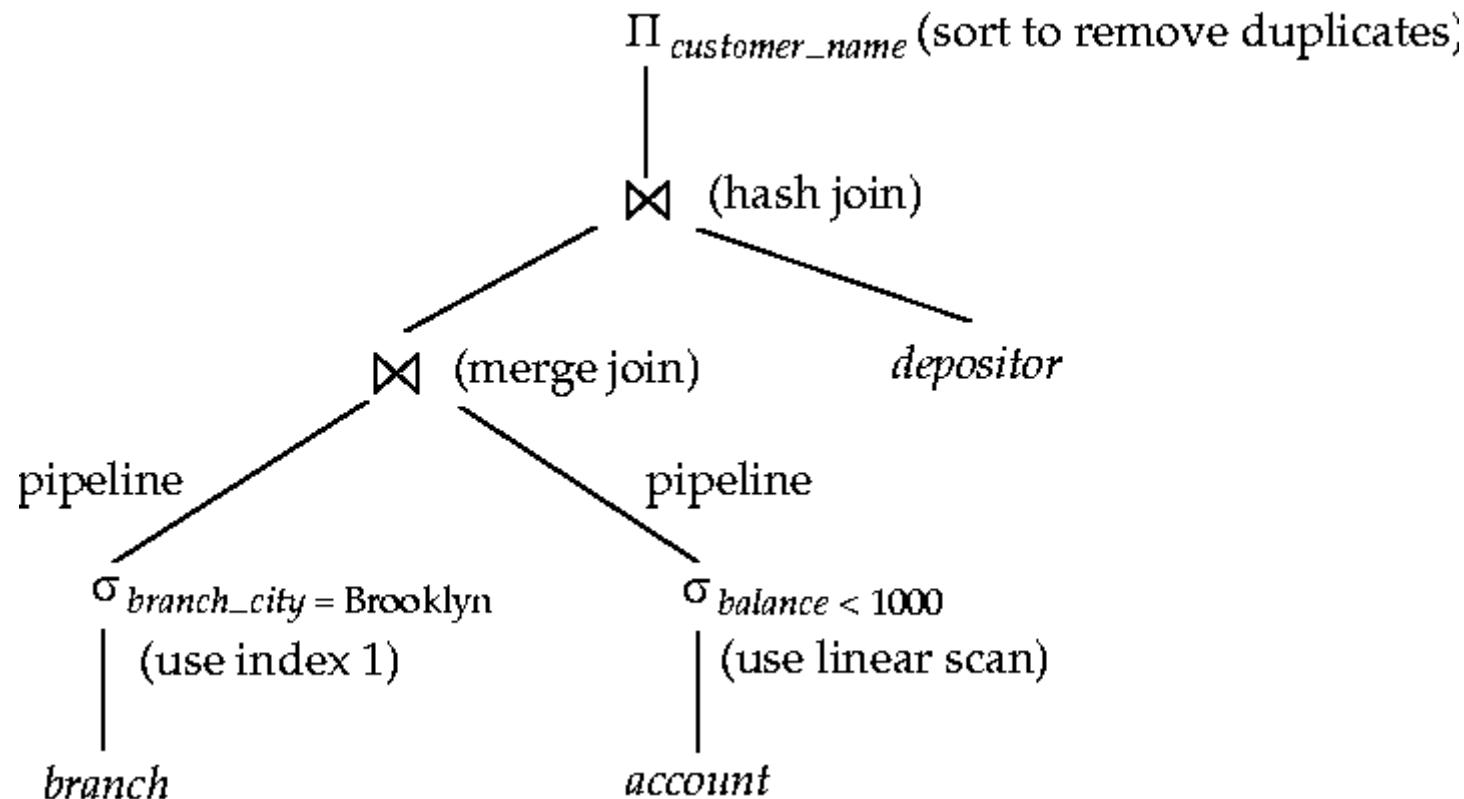
- Alternative ways of evaluating a given query
 - Equivalent expressions
 - Different algorithms for each operation





Query Evaluation Plan (example)

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.





Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
 - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
 1. Generate logically equivalent expressions using **equivalence rules**
 2. Annotate resultant expressions to get alternative query plans
 3. Choose the cheapest plan based on **estimated cost**
- Estimate of plan cost based on:
 - Statistical information about relations. e.g. -
 - ▶ number of tuples, number of distinct values for an attribute
 - Statistics estimation for intermediate results
 - ▶ to compute cost of complex expressions
 - Cost formulae for algorithms, computed using statistics



Transactions



Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully (is committed), the database must be consistent.
- After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Multiple transactions can execute in parallel.
- Two main issues to deal with:
 - Failures of various kinds, such as hardware failures and system crashes
 - Concurrent execution of multiple transactions



ACID Properties of Transactions

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
 - That is, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.



Example of Fund Transfer (Transaction)

- Transaction to transfer \$50 from account A to account B:
 1. **read(A)**
 2. $A := A - 50$
 3. **write(A)**
 4. **read(B)**
 5. $B := B + 50$
 6. **write(B)**
- **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.



Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).
 - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
 - However, executing multiple transactions concurrently has significant benefits, as we will see later.
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist despite failures.

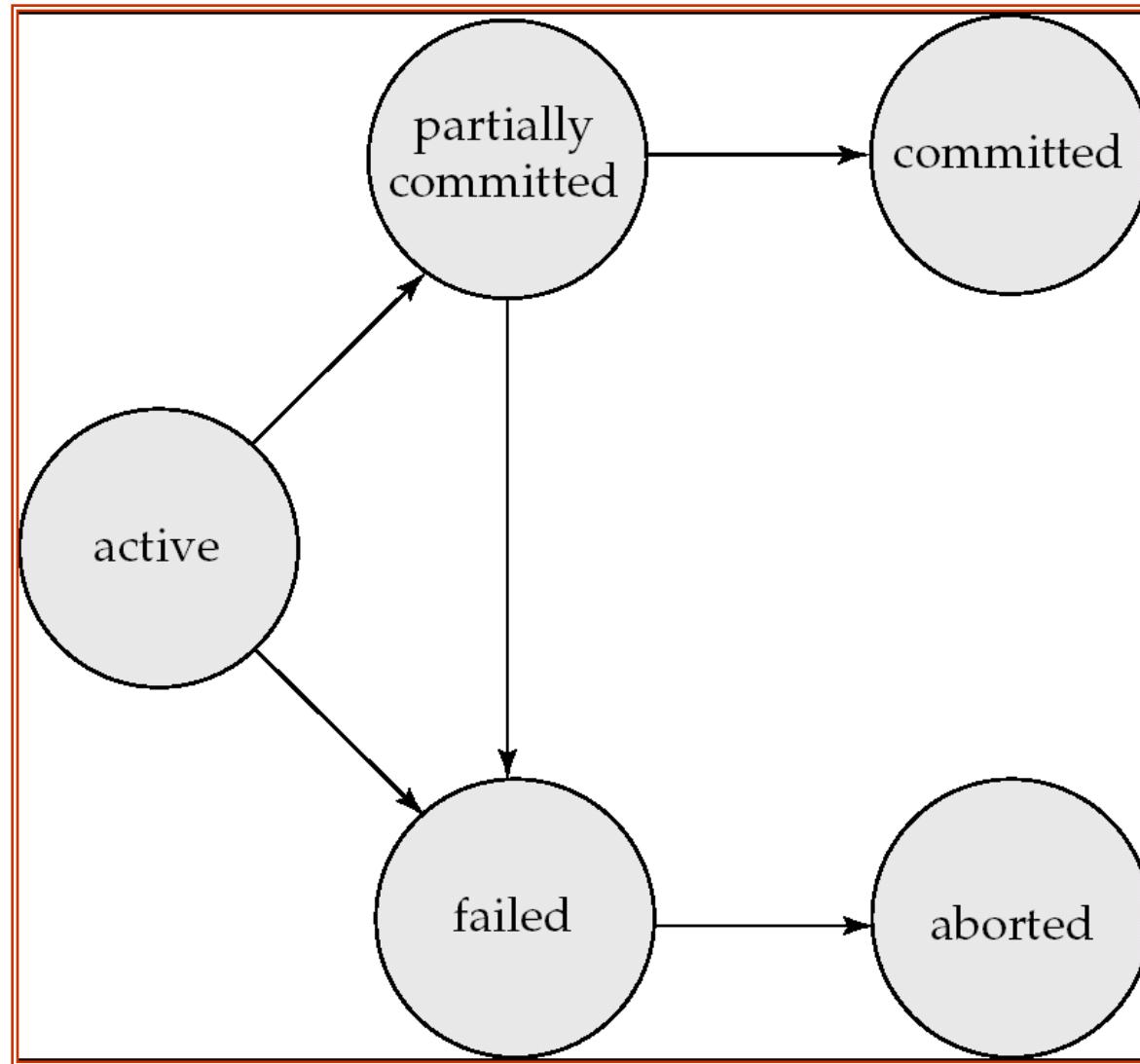


Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
Two options after it has been aborted:
 - restart the transaction; can be done only if no internal logical error
 - kill the transaction
- **Committed** – after successful completion.



Transaction State (Cont.)



Embedded Database Systems

GAYASHAN AMARASINGHE

GAYASHAN@CSE.MRT.AC.LK

Outline

Embedded Database Systems

- Definition
- Motivation
- Embedded database concerns
- Embedded database characteristics
- Embedded database architectures
 - Client-server vs embedded (lightly and deeply embedded)
- Examples

Commercial Embedded Databases (Oracle)

Mini Project

What is an embedded database system?

It is a database management system (DBMS) which is;

- Tightly integrated with an application software
- Transparent to the application end-user (hidden from the end-user)
- Requires little/no maintenance

Related terms: Backend DBs, In-memory DBs, Mobile DBs

Why do we need embedded databases?

- Extremely fast access
- Fault tolerance
- Persistence and reliability
- Fast to market
- Low Total Cost of Ownership (TCO – purchase price and the operational costs)
- No Database Administrator (DBA) required

Concerns of Embedded databases

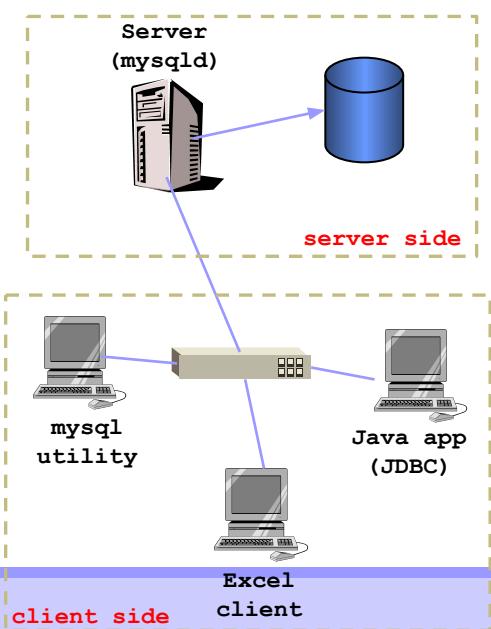
- APIs (SQL, proprietary and native APIs)
- Architectures (lightly embedded, deeply embedded)
- Storage modes (on-disk, in-memory, combined)
- Database modes (relational, object-oriented, entity-attribute-value, network, ...)
- Target markets

Embedded database characteristics

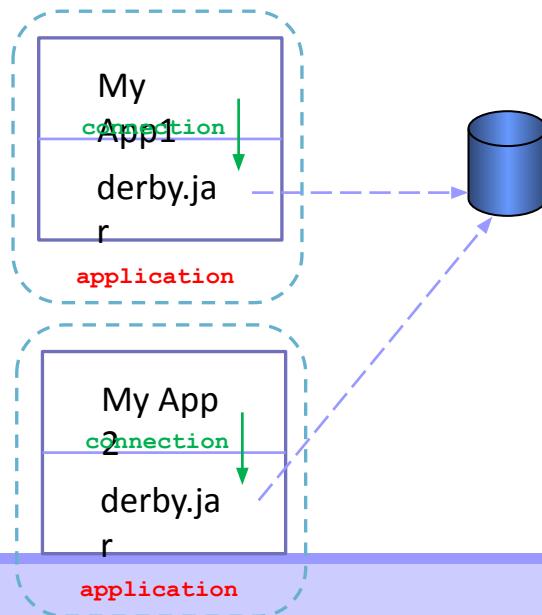
- Embedded in an application
- Can be incorporated in a scripting language
- Inexpensive
- May not scale well (depends on how it is implemented)
- Good transaction control
- Text search support may be minimal
- May not support SQL

Embedded database architectures

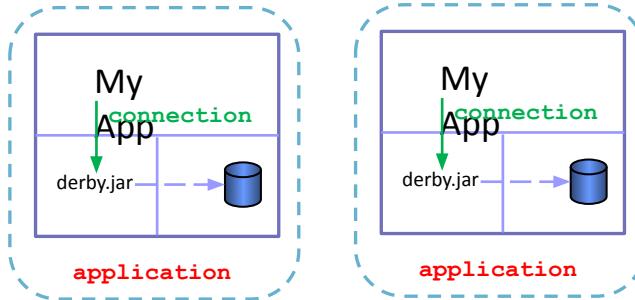
CLIENT-SERVER



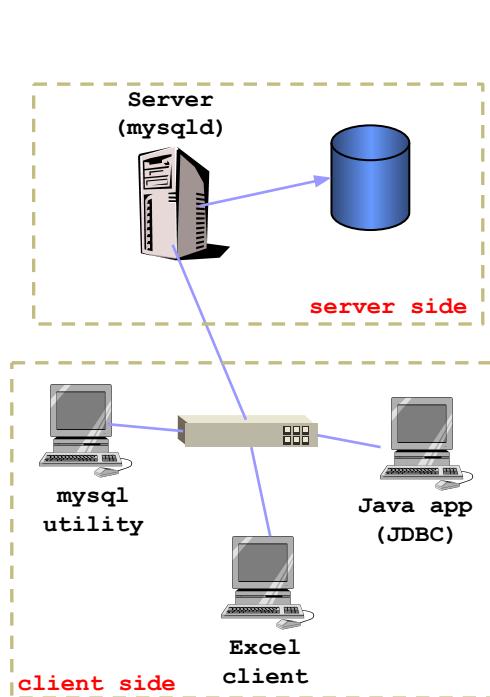
LIGHTLY EMBEDDED



DEEPLY EMBEDDED

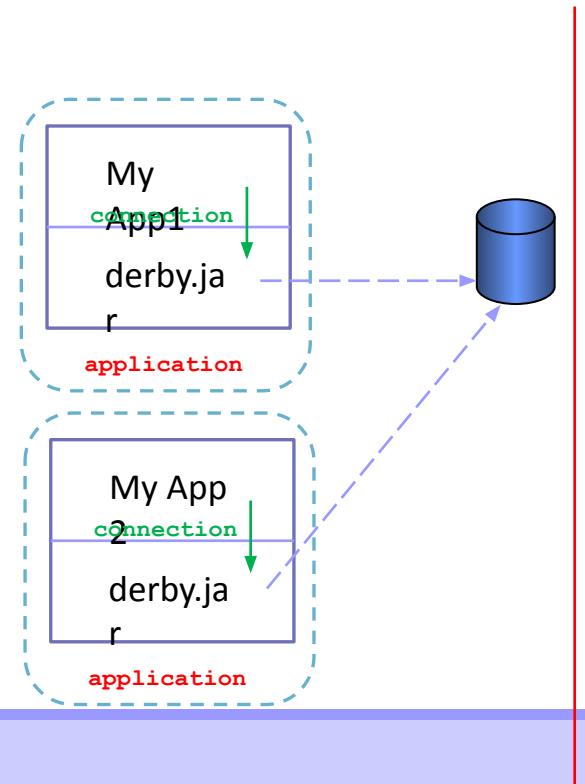


Client Server architecture (traditional)



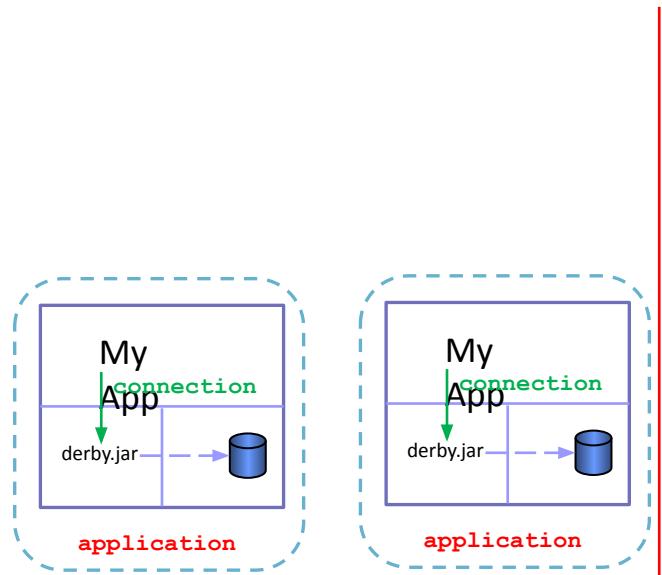
- Database server is a separate process
- Running on a host
- Clients can run on any machine
- Many different programs may be clients
- Supports standard APIs

Lightly Embedded Database architecture



- Database engine installed as part of the application installation on the same machine
- Small footprint
- Full features (mostly)
- Other applications maybe able to access the same database
- Multi device support may be available

Deeply Embedded Database architecture



- Application runs the database ‘inside’ it
- No database engine installation required
- Uses a library that has database capabilities
- Popular with mobile applications
- Can be easily distributed with the application

Example: Hypersonic SQL

- lightweight, *fast* database written in Java
- database can be stored in memory or on disk.
- embed in Java app - no separate server
 - don't need to install database server or disk-based database
- can also run in client-server mode
- useful for development and "demo" systems
- <http://hsqldb.org>



Example: derby

- lightweight, pure Java database
- formerly "Cloudscape", donated to Apache foundation
- only 1 user can connect to database at a time
- embed in Java applications - no separate server
 - similar to HSQLDB
- can also run in client-server mode
- included with JavaEE as "Java DB"
- **<http://db.apache.org/derby>**



Example: SQLITE

- World's most widely distributed database
- written in C
- very small: 350KB binary
- used on Android
- 3rd party JDBC drivers:
 - <http://code.google.com/p/sqlite-jdbc/>
 - <http://www.ch-werner.de/javasqlite/>
 - <http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC>



Example: Berkeley DB

- *libraries* for embedded database using the OS's file system.
- No db manager, No network access, No query language.
- used as data tier for LDAP, sendmail, and many other apps
- very small and *fast* -- faster than any relational DB w/ manager
- **C** and **pure Java** version
 - language bindings for C++, Perl, Python, Ruby, and more
- **bought by Oracle** in 2006: <http://www.oracle.com/database/berkeley-db/index.html>
- still Open Source under the "Sleepycat Public License" and "Sleepycat Commercial License",
- not required to distribute the source code with your app.



Example: Interbase

- developed and marketed by [Embarcadero Technologies](#)
- can operate in both server and embedded modes
- supports database and column level encryption
- very small – server (40MB), client (400KB)
- supports SQL, JDBC, ODBC, ...
- <https://www.embarcadero.com/products/interbase>



Commercial embedded databases (Oracle)

Oracle is a market leader in embedded databases – 23% market share

“Lights out” embedded databases – no database administrator (DBA) required

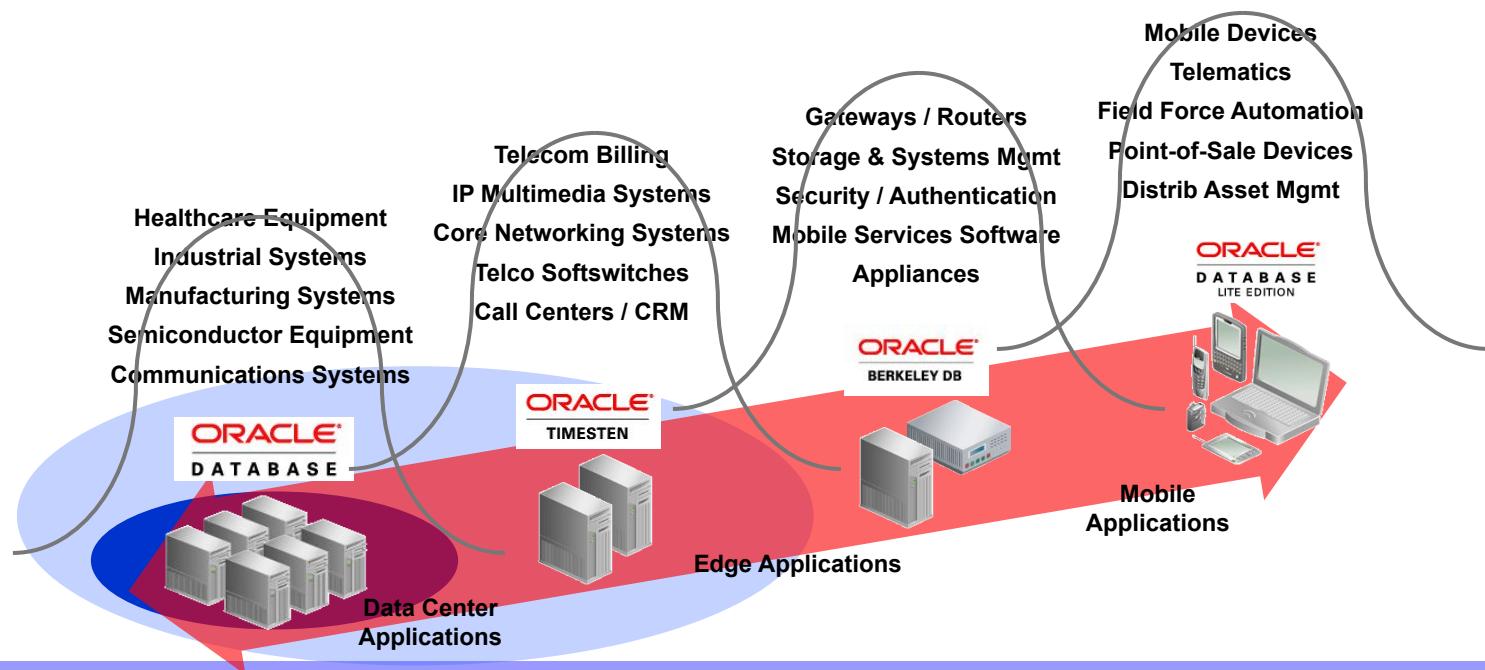
Small footprint (< MB)

Aggressive pricing models

Examples:

- TimesTen
- Berkeley DB
- Database Lite

Where are they deployed?



TimesTen database

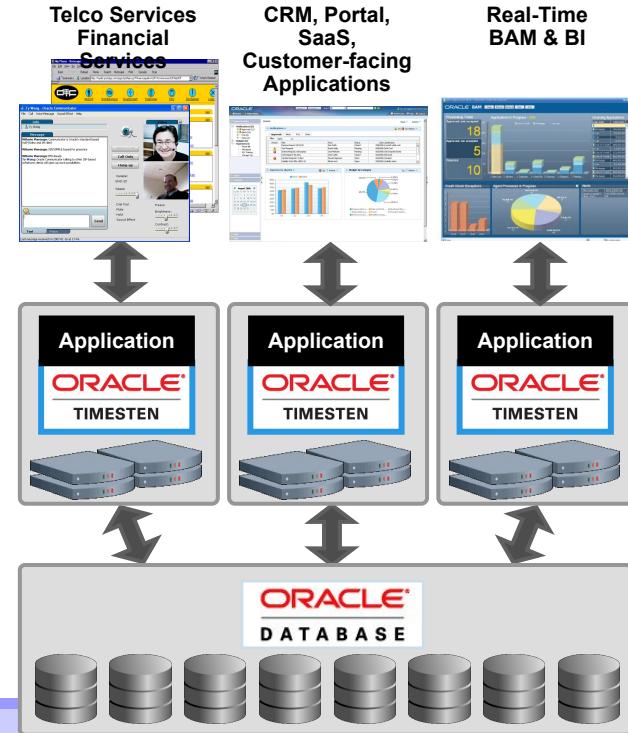
Memory optimized in-memory RDBMS for real time applications

Application-tier relational database

Delivers instant responsiveness and very high throughput

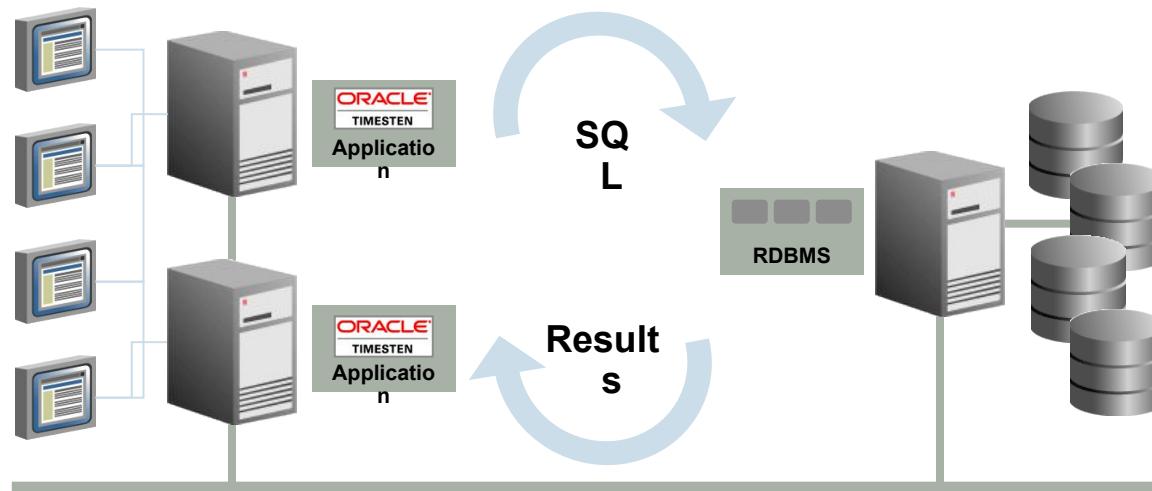
Operates as database of record or as a read/write cache for Oracle Database

Provides replication for high availability and scalability



TimesTen database

Combines database + cache



TimesTen database features

Base product

- Everything runs in-memory – efficiency
- Local disks for persistence and recovery
- Full read/write transactional RDBMS with shares and multi-user access

High availability

- Server pairs on hot-standby
- Replication

Can be used to as a cache to an external oracle database

Oracle 10g vs timesten

Database Characteristic	Oracle Database 10g	Oracle TimesTen In-Memory Database
Data Model	Relational – SQL	Relational – SQL
Target Applications	All	OLTP, some DSS
Optimization	Disk-centric	Memory-centric
Typical Deployment	Database Tier	Application Tier
Architecture	Client / Server	Direct Data Access
Response Time	Milliseconds	Microseconds
Data Capacity	Tens of Terabytes	Tens of Gigabytes
Scalability	Unlimited SMP/Cluster	Good SMP

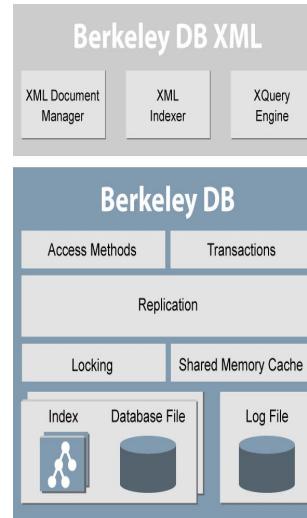
Berkeley db

- High performance database engine
 - Runs directly in application's address space
 - Application-native data storage
 - No SQL layer overhead
- Low total cost of ownership
 - High performance with less hardware
 - Embedded administration
 - Lower development cost: cheaper to buy vs. build

Berkeley db

All three Berkeley DB Products:

- Libraries linked to your application
- Simple, direct, indexed data storage
- Key-value pairs with simple, get-put style API
 - `getDocument/putDocument` for DB XML
- Operate in memory, on disk or both
- Programmatic administration API
- Low latency & high throughput
- ACID transactions and recovery
- Open source



Summary

Embedded databases are databases tightly coupled with applications

- with a small footprint
- requires very little maintenance
- transparent to the end user

Two architectures: lightly vs deeply embedded

Commercial embedded database examples (Oracle)

Embedded DB - Mini Project

Implement the data storage of an Android based expense manager application using an embedded database.

More details on the submission will be provided on Moodle.

Visit: <https://github.com/GayashanNA/SimpleExpenseManager>

References

Database System Concepts, Sixth Edition, Avi Silberschatz, Henry F. Korth, S. Sudarshan

Oracle embedded databases, <https://www.slideshare.net/Prem02/oracle-embedded>

Seltzer, Margo I., and Michael A. Olson. "Challenges in Embedded Database System Administration." USENIX Workshop on Embedded Systems. 1999.

Chapter 11: Introduction to Indexing and Hashing

Indexing: Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form



- Index files are typically much smaller than the original file
- Two basic kinds of indices:
 - Ordered indices: search keys are stored in sorted order
 - Hash indices: search keys are distributed uniformly across “buckets” using a “hash function”.

Index Evaluation Metrics

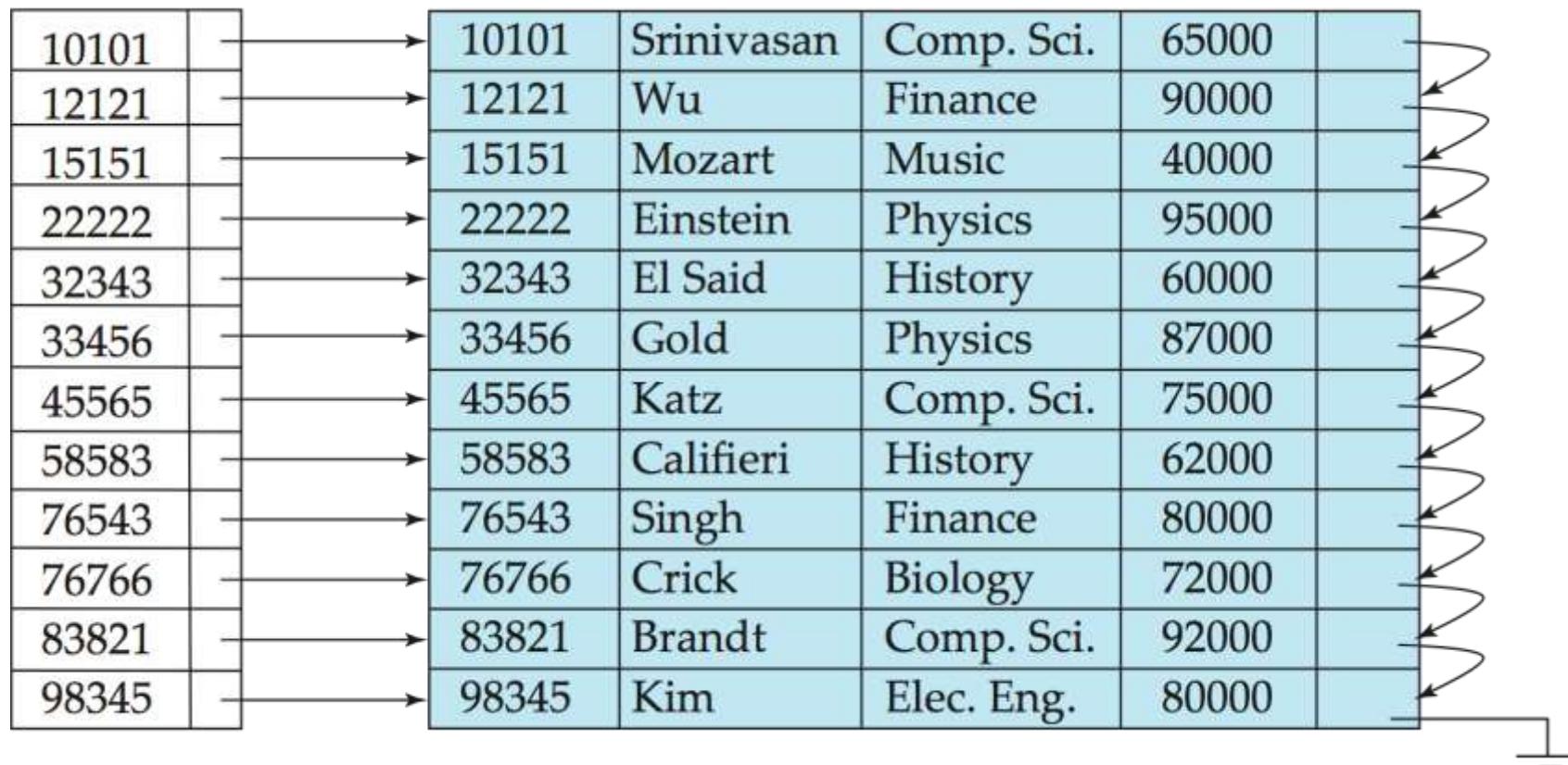
- Access types supported efficiently. E.g.,
 - records with a specified value in the attribute
 - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value.
 - E.g., author catalog in library.
- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **Primary index**.
 - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index:** an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.
- **Index-sequential file:** ordered sequential file with a primary index.

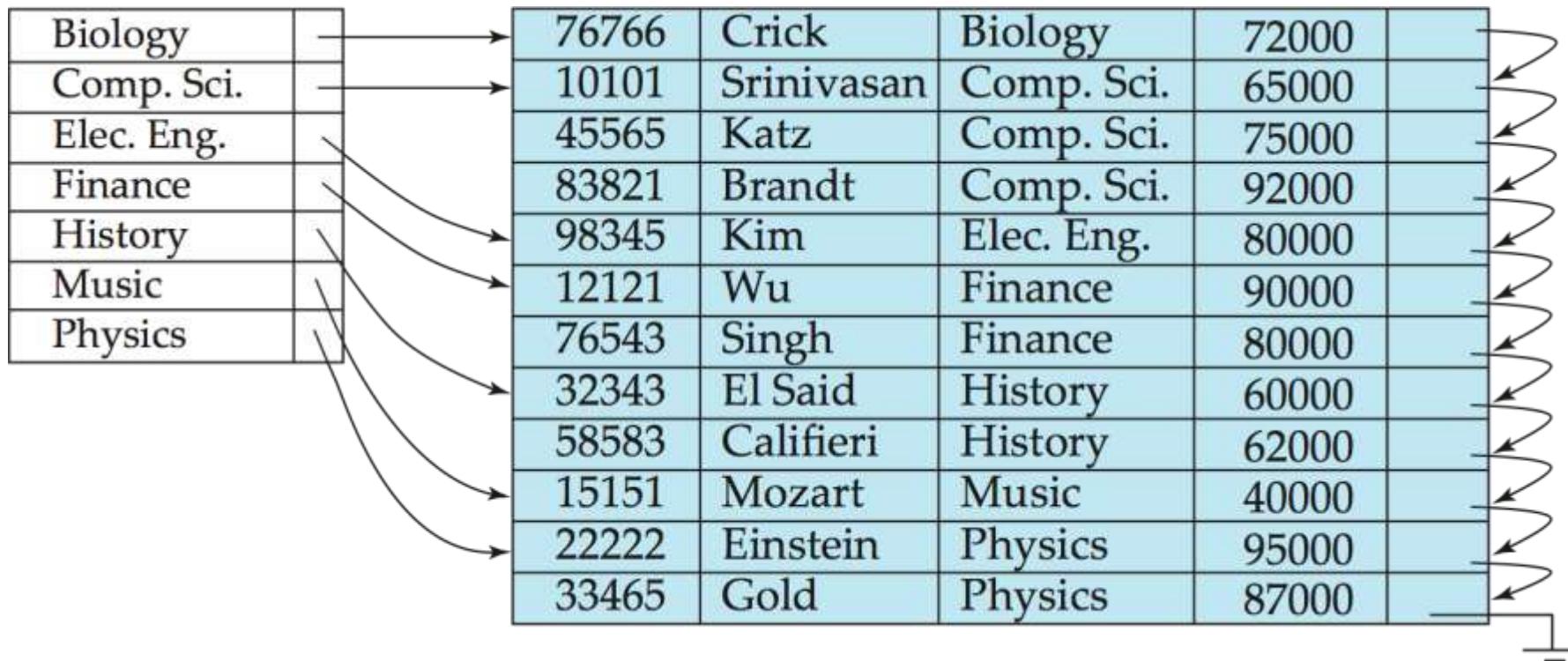
Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
 - E.g. index on ID attribute of instructor relation



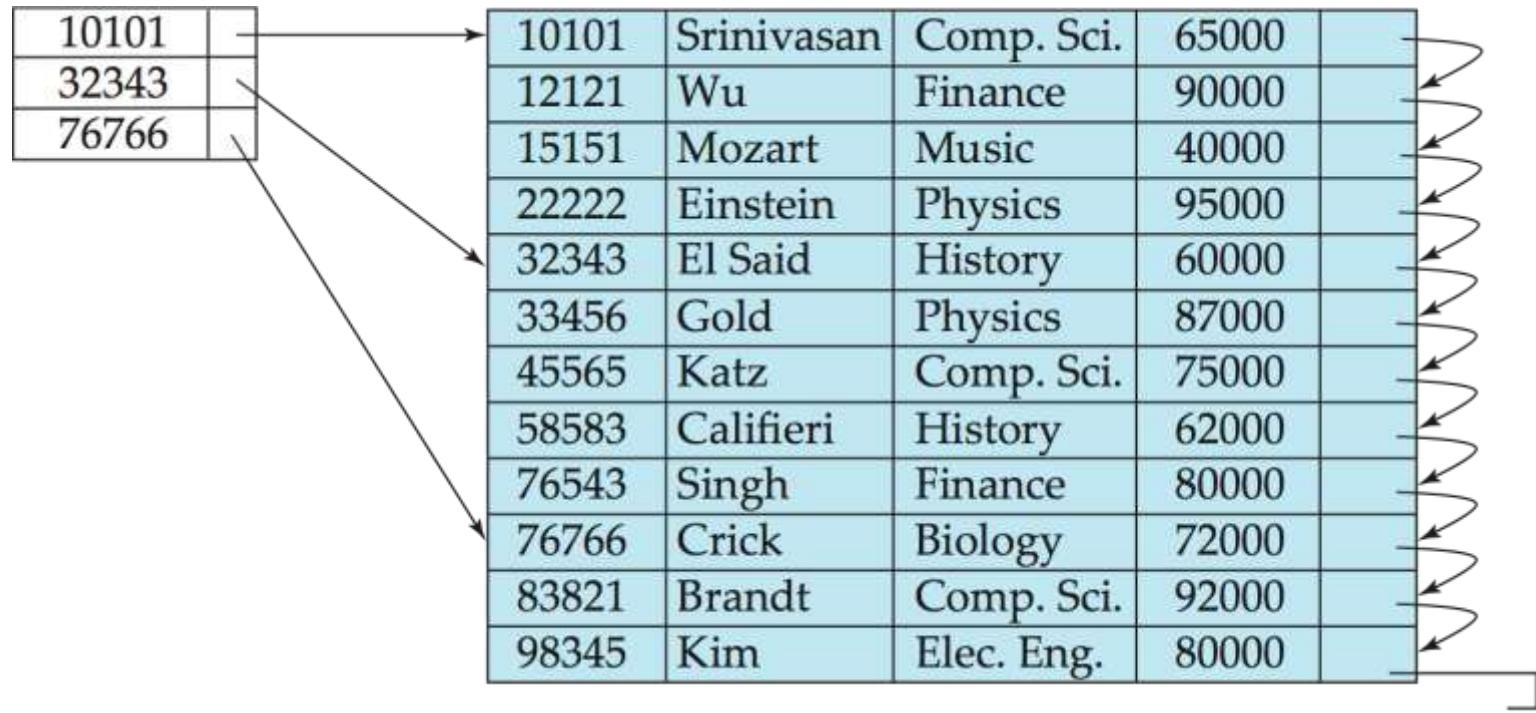
Dense Index Files (Cont.)

- Dense index on dept_name, with instructor file sorted on dept_name



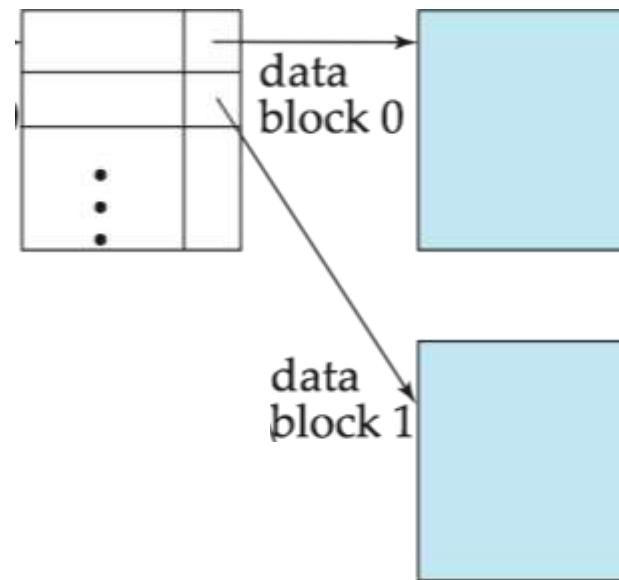
Sparse Index Files

- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value < K
 - Search file sequentially starting at the record to which the index record points



Sparse Index Files (Cont.)

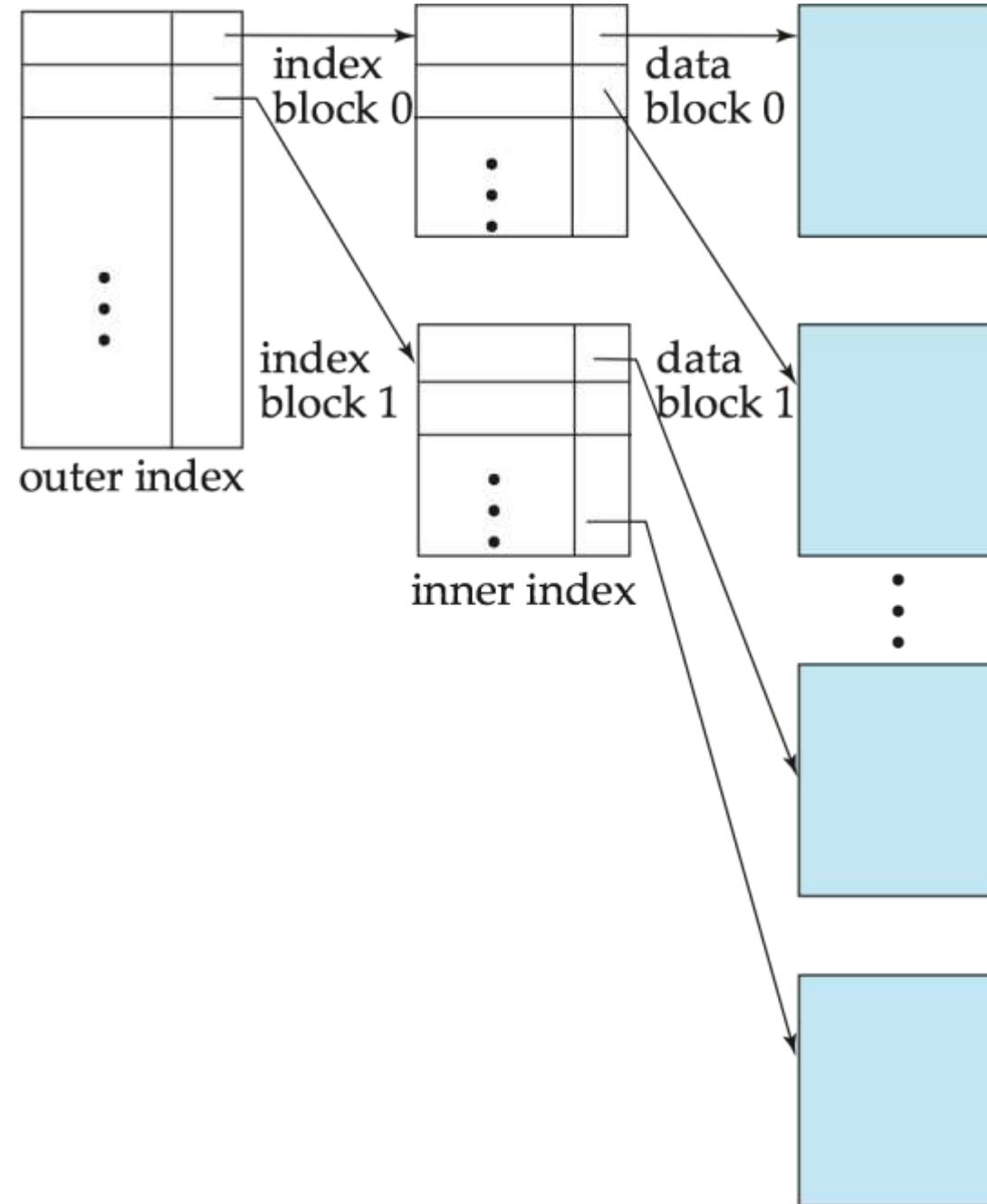
- Compared to dense indices:
 - Less space and less maintenance overhead for insertions and deletions.
 - Generally slower than dense index for locating records.
- **Good tradeoff:** sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
 - outer index – a sparse index of primary index
 - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

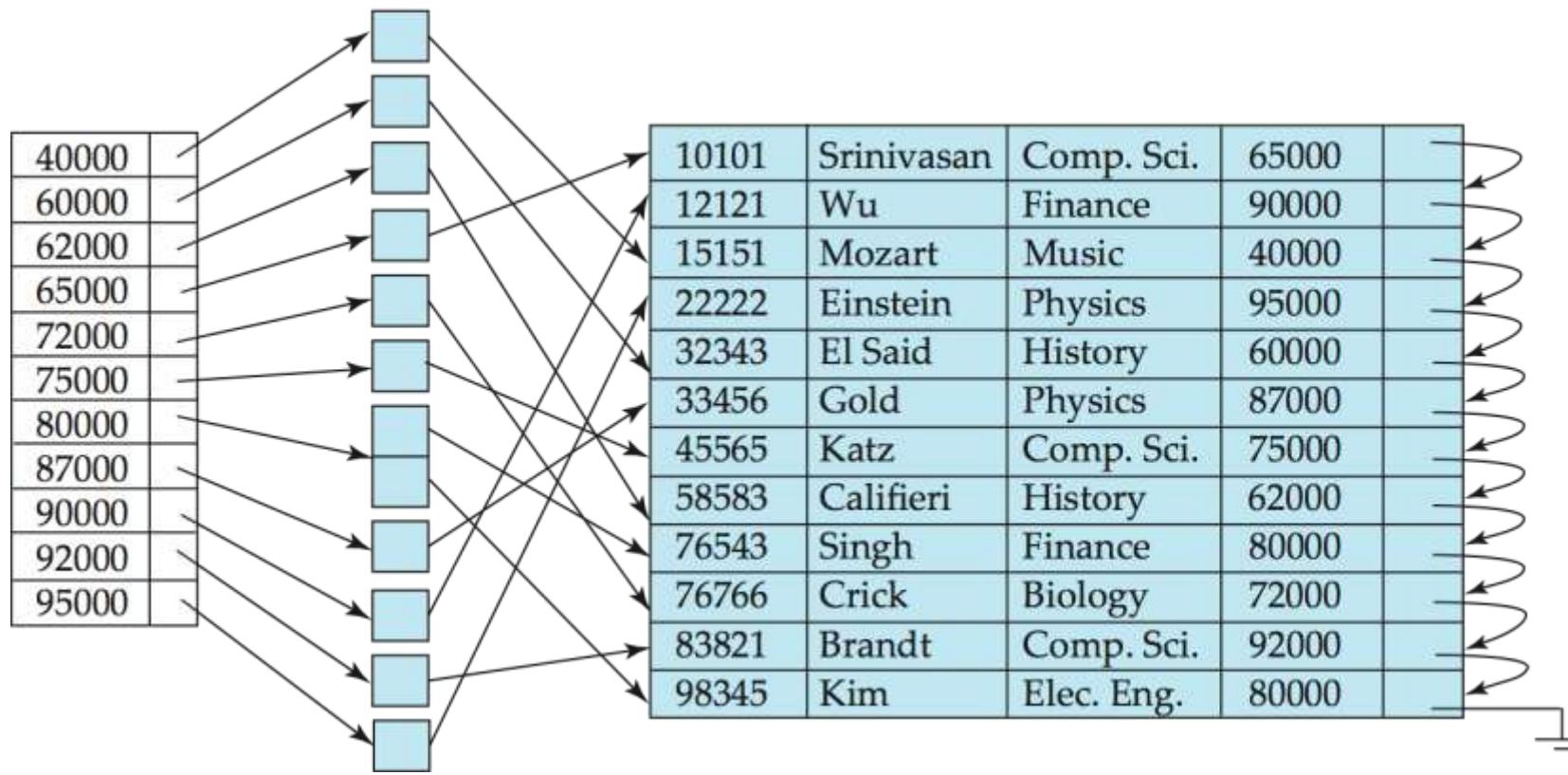
Multilevel Index



Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
 - Example 1: In the instructor relation stored sequentially by ID, we may want to find all instructors in a particular department
 - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

Secondary Indices Example



Secondary index on *salary* field of *instructor*

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Primary and Secondary Indices

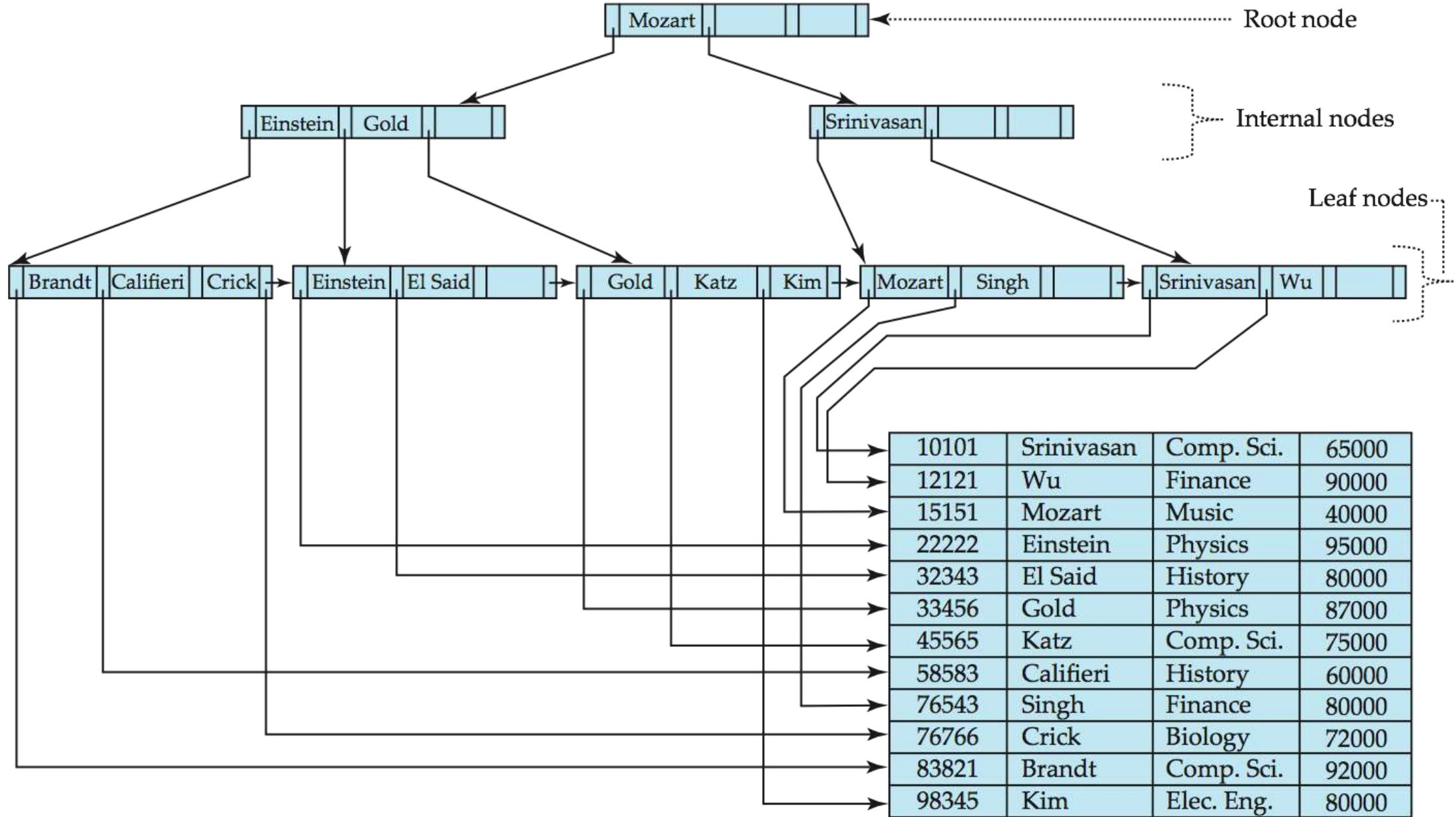
- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
 - Each record access may fetch a new block from disk
 - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

B+-Tree Index Files

B⁺-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
 - performance degrades as file grows, since many overflow blocks get created.
 - Periodic reorganization of entire file is required.
- Advantage of B+-tree index files:
 - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
 - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B+-trees:
 - extra insertion and deletion overhead, space overhead.
- Advantages of B+-trees outweigh disadvantages
 - B+-trees are used extensively

Example of B⁺-Tree



Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**.
- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B .
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key (See figure in next slide.)

- There are 10 buckets,
- The binary representation of the ith character is assumed to be the integer i.
- The hash function returns the sum of the binary representations of the characters modulo 10
 - E.g. $h(\text{Music}) = 1$ $h(\text{History}) = 2$
 $h(\text{Physics}) = 3$ $h(\text{Elec. Eng.}) = 3$

Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key
(see previous slide for details).

Hash Functions

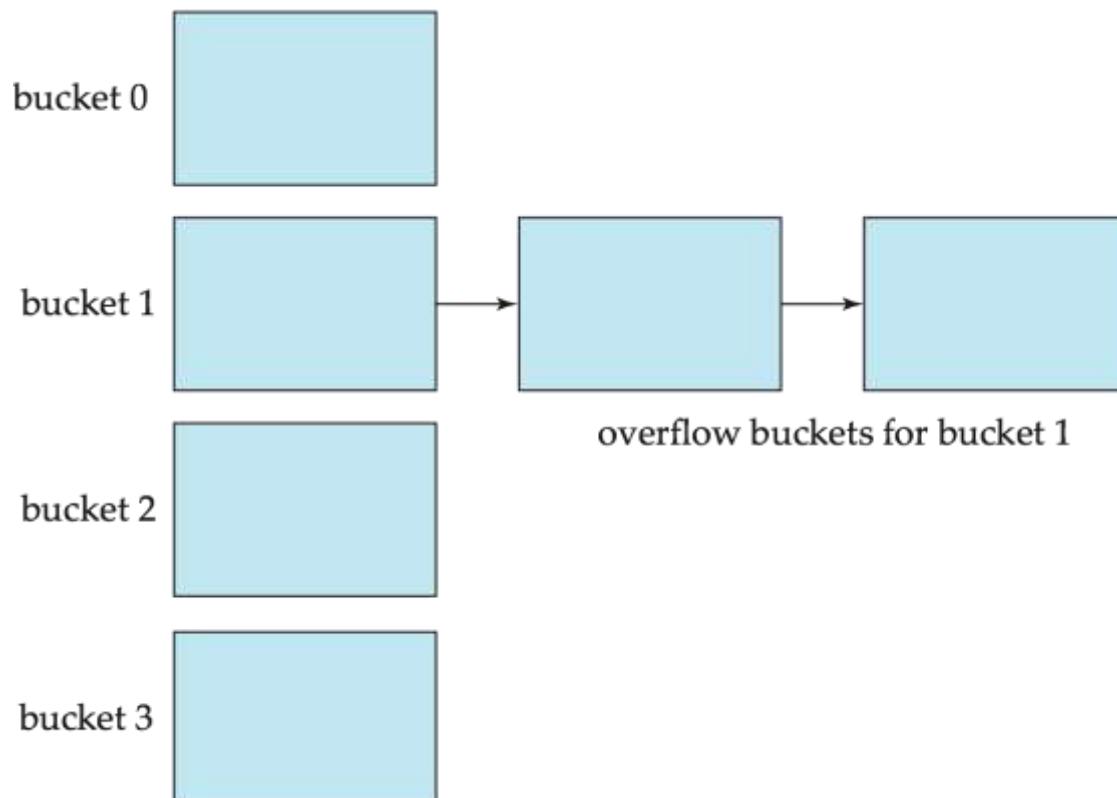
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
 - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

Handling of Bucket Overflows

- Bucket overflow can occur because of
 - Insufficient buckets
 - Skew in distribution of records. This can occur due to two reasons:
 - ▶ multiple records have same search-key value
 - ▶ chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using **overflow buckets**.

Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
 - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.



Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
 - Given a number n it must be easy to retrieve record n
 - ▶ Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
 - E.g. gender, country, state, ...
 - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
 - Bitmap has as many bits as records
 - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	<i>ID</i>	<i>gender</i>	<i>income_level</i>
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for <i>gender</i>		Bitmaps for <i>income_level</i>	
m	10010	L1	10100
f	01101	L2	01000
		L3	00001
		L4	00010
		L5	00000

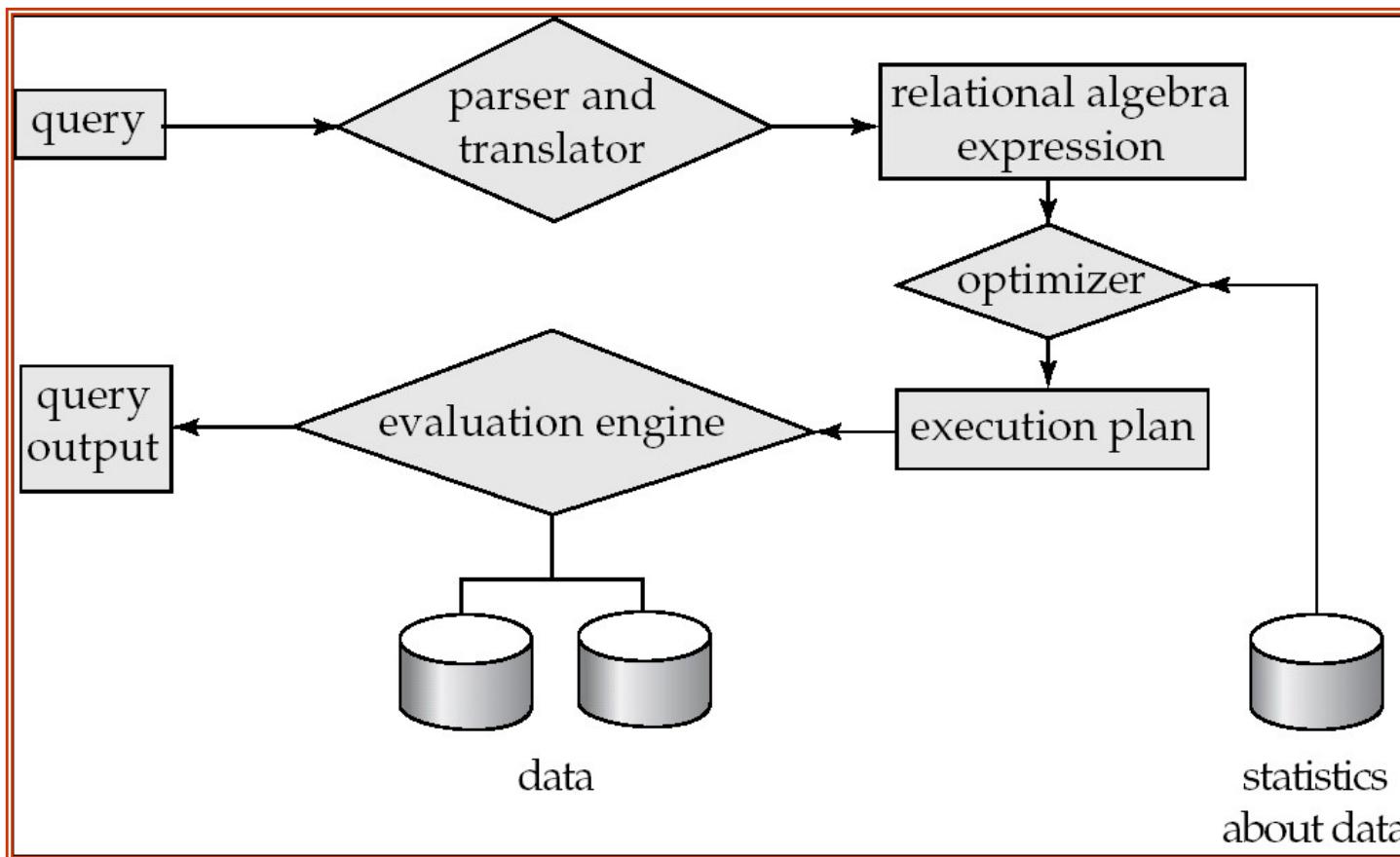


Introduction to Query Processing and Transactions



Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





Basic Steps in Query Processing (Cont.)

- Parsing and translation
 - translate the query into its internal form. This is then translated into relational algebra.
 - Parser checks syntax, verifies relations
- Evaluation
 - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Basic Steps in Query Processing : Optimization (example)

```
select balance  
from account  
where balance < 2500
```

- A relational algebra expression may have many equivalent expressions
 - E.g., $\sigma_{balance < 2500}(\Pi_{balance}(account))$ is equivalent to
$$\Pi_{balance}(\sigma_{balance < 2500}(account))$$
- Each relational algebra operation can be evaluated using one of several different algorithms
 - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
 - E.g., can use an index on *balance* to find accounts with $balance < 2500$,
 - or can perform complete relation scan and discard accounts with $balance \geq 2500$



Basic Steps: Optimization (Cont.)

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
 - Cost is estimated using statistical information from the database catalog
 - ▶ e.g. number of tuples in each relation, size of tuples, etc.



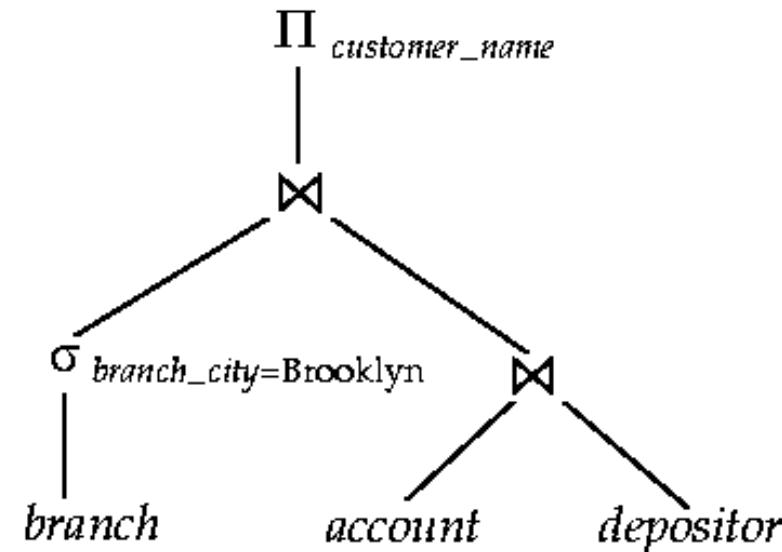
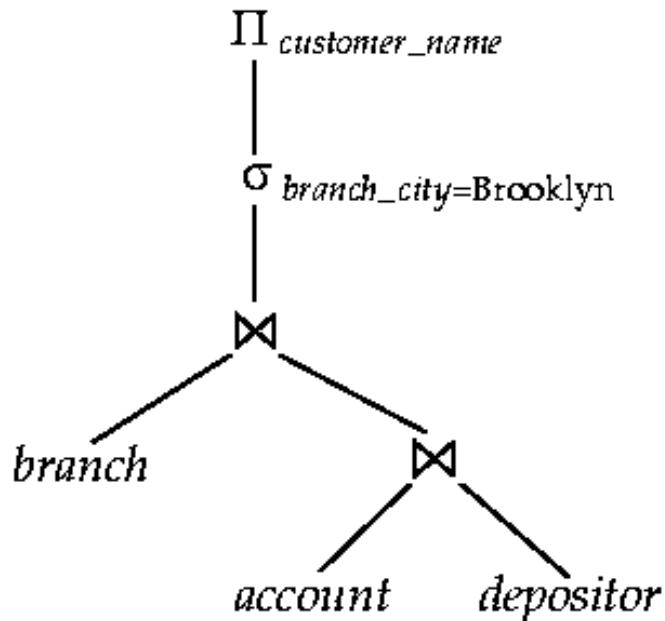
Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
 - Many factors contribute to time cost
 - ▶ *disk accesses, CPU, or even network communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
 - Number of seeks * average-seek-cost
 - Number of blocks read * average-block-read-cost
 - Number of blocks written * average-block-write-cost
 - ▶ Cost to write a block is greater than cost to read a block
 - data is read back after being written to ensure that the write was successful



Alternative Query Expressions (ex.)

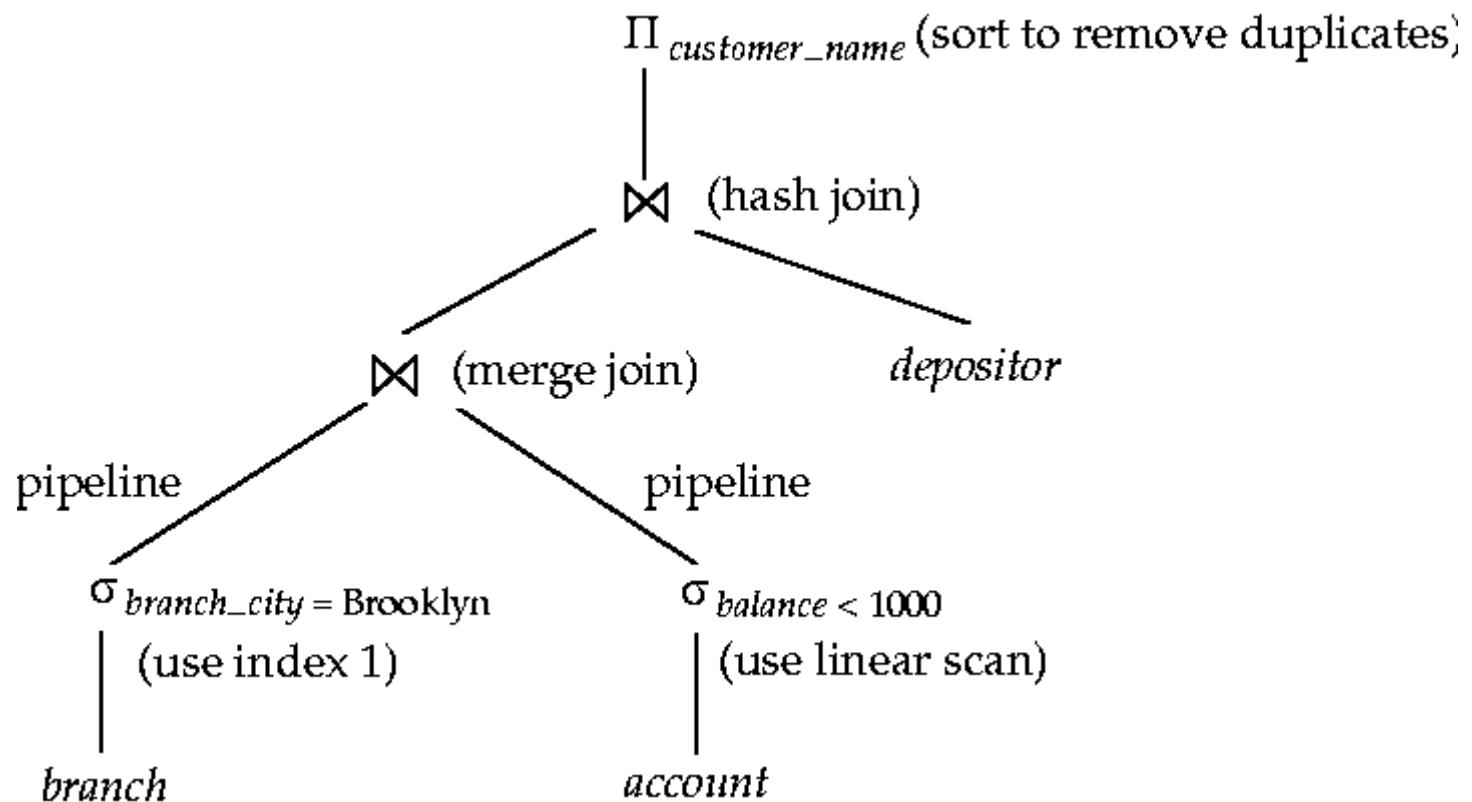
- Alternative ways of evaluating a given query
 - Equivalent expressions
 - Different algorithms for each operation





Query Evaluation Plan (example)

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.





Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
 - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
 1. Generate logically equivalent expressions using **equivalence rules**
 2. Annotate resultant expressions to get alternative query plans
 3. Choose the cheapest plan based on **estimated cost**
- Estimate of plan cost based on:
 - Statistical information about relations. e.g. -
 - ▶ number of tuples, number of distinct values for an attribute
 - Statistics estimation for intermediate results
 - ▶ to compute cost of complex expressions
 - Cost formulae for algorithms, computed using statistics



Transactions

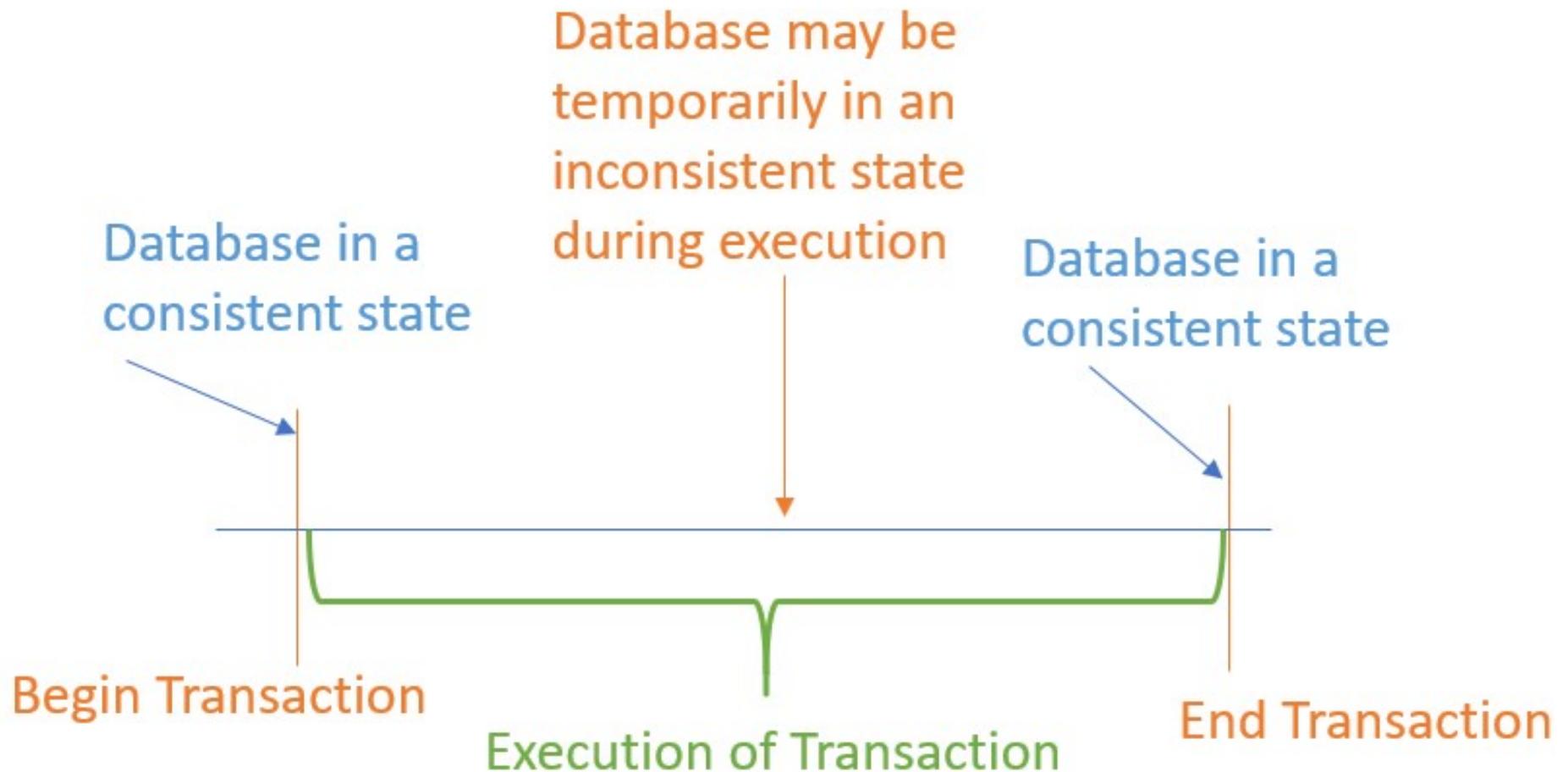


Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully (is committed), the database must be consistent.
- After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Multiple transactions can execute in parallel.
- Two main issues to deal with:
 - Failures of various kinds, such as hardware failures and system crashes
 - Concurrent execution of multiple transactions



Transaction Execution





ACID Properties of Transactions

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
 - That is, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.



Example of Fund Transfer (Transaction)

- Transaction to transfer \$50 from account A to account B:

1. **read(A)**
2. $A := A - 50$
3. **write(A)**
4. **read(B)**
5. $B := B + 50$
6. **write(B)**

$\text{SUM}(A + B)_{\text{BeforeTrans}}$



$= \text{SUM } (A + B)_{\text{AfterTrans}} ?$

- **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.



Example of Fund Transfer (Cont.)

1. **read(A)**
2. $A := A - 50$
3. **write(A)**
4. **read(B)**
5. $B := B + 50$
6. **write(B)**

1. **read(A)**
2. Display (A)

1. **read(A)**
2. Display (A)

- **Isolation requirement** — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).
 - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
 - However, executing multiple transactions concurrently has significant benefits, as we will see later.



Example of Fund Transfer (Cont.)

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist despite failures.



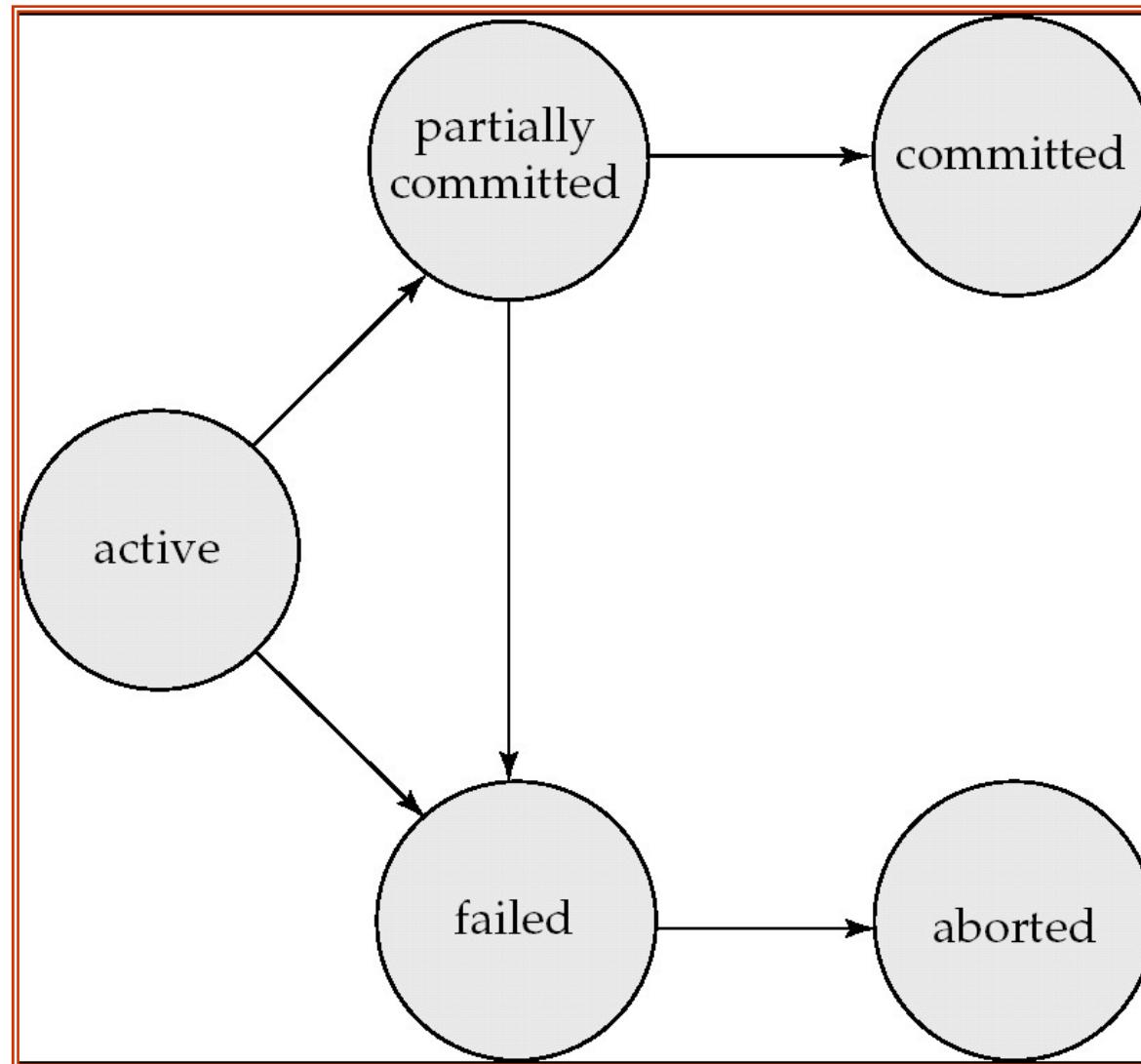


Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
Two options after it has been aborted:
 - restart the transaction; can be done only if no internal logical error
 - kill the transaction
- **Committed** – after successful completion.



Transaction State (Cont.)





Chapter 10: Big Data

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Motivation

- Very large volumes of data being collected
 - Driven by growth of web, social media, and more recently internet-of-things
 - Web logs were an early source of data
 - Analytics on web logs has great value for advertisements, web site structuring, what posts to show to a user, etc
- Big Data: differentiated from data handled by earlier generation databases
 - **Volume**: much larger amounts of data stored
 - **Velocity**: much higher rates of insertions
 - **Variety**: many types of data, beyond relational data



Querying Big Data

- Transaction processing systems that need very high scalability
 - Many applications willing to sacrifice ACID properties and other database features, if they can get very high scalability
- Query processing systems that
 - Need very high scalability, and
 - Need to support non-relation data



Big Data Storage Systems

- Distributed file systems
- Sharding across multiple databases
- Key-value storage systems
- Parallel and distributed databases



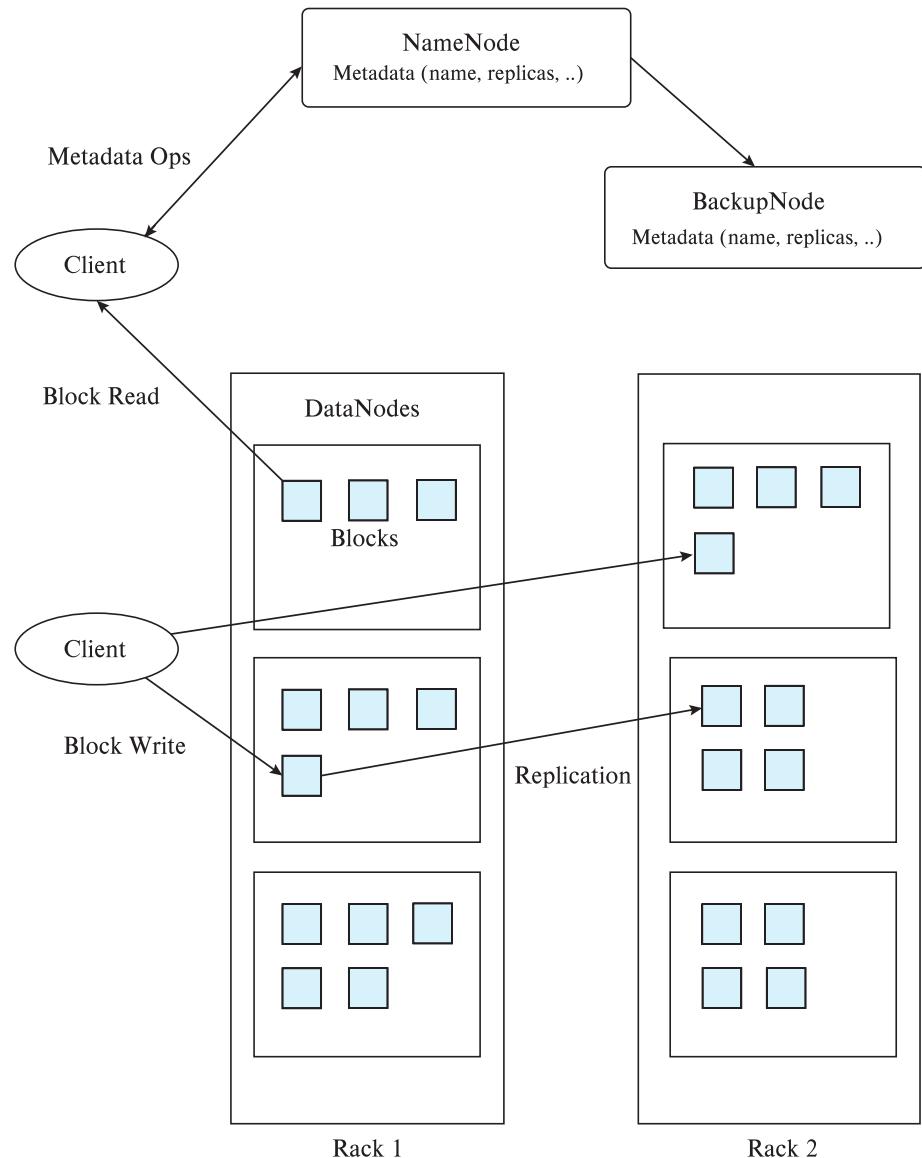
Distributed File Systems

- A distributed file system stores data across a large collection of machines, but provides single file-system view
- Highly scalable distributed file system for large data-intensive applications.
 - E.g., 10K nodes, 100 million files, 10 PB
- Provides redundant storage of massive amounts of data on cheap and unreliable computers
 - Files are replicated to handle hardware failure
 - Detect failures and recovers from them
- Examples:
 - Google File System (GFS)
 - Hadoop File System (HDFS)



Hadoop File System Architecture

- Single Namespace for entire cluster
- Files are broken up into blocks
 - Typically 64 MB block size
 - Each block replicated on multiple DataNodes
- Client
 - Finds location of blocks from NameNode
 - Accesses data directly from DataNode





Hadoop Distributed File System (HDFS)

- **NameNode**
 - Maps a filename to list of Block IDs
 - Maps each Block ID to DataNodes containing a replica of the block
- **DataNode**: Maps a Block ID to a physical location on disk
- Data Coherency
 - Write-once-read-many access model
 - Client can only append to existing files
- Distributed file systems good for millions of large files
 - But have very high overheads and poor performance with billions of smaller tuples



Sharding

- **Sharding:** partition data across multiple databases
- Partitioning usually done on some ***partitioning attributes*** (also known as ***partitioning keys*** or ***shard keys*** e.g. user ID
 - E.g., records with key values from 1 to 100,000 on database 1, records with key values from 100,001 to 200,000 on database 2, etc.
- Application must track which records are on which database and send queries/updates to that database
- Positives: scales well, easy to implement
- Drawbacks:
 - Not transparent: application has to deal with routing of queries, queries that span multiple databases
 - When a database is overloaded, moving part of its load out is not easy
 - Chance of failure more with more databases
 - need to keep replicas to ensure availability, which is more work for application



Parallel and Distributed Databases

- Parallel databases run multiple machines (cluster)
 - Developed in 1980s, well before Big Data
- Parallel databases were designed for smaller scale (10s to 100s of machines)
 - Did not provide easy scalability
- **Replication** used to ensure data availability despite machine failure
 - But typically restart query in event of failure
 - Restarts may be frequent at very large scale
 - Map-reduce systems (coming up next) can continue query execution, working around failures



Replication and Consistency

- **Availability** (system can run even if parts have failed) is essential for parallel/distributed databases
 - Via replication, so even if a node has failed, another copy is available
- **Consistency** is important for replicated data
 - All live replicas have same value, and each read sees latest version
 - Often implemented using majority protocols
 - E.g., have 3 replicas, reads/writes must access 2 replicas
 - Details in chapter 23
- **Network partitions** (network can break into two or more parts, each with active systems that can't talk to other parts)
- In presence of partitions, cannot guarantee both availability and consistency
 - Brewer's CAP “Theorem”



The MapReduce Paradigm

- Platform for reliable, scalable parallel computing
- Abstracts issues of distributed and parallel environment from programmer
 - Programmer provides core logic (via map() and reduce() functions)
 - System takes care of parallelization of computation, coordination, etc.
- Paradigm dates back many decades
 - But very large scale implementations running on clusters with 10^3 to 10^4 machines are more recent
 - Google Map Reduce, Hadoop, ..
- Data storage/access typically done using distributed file systems or key-value stores



MapReduce: Word Count Example

- Consider the problem of counting the number of occurrences of each word in a large collection of documents
- How would you do it in parallel?
- Solution:
 - Divide documents among workers
 - Each worker parses document to find all words, map function outputs (word, count) pairs
 - Partition (word, count) pairs across workers based on word
 - For each word at a worker, reduce function locally add up counts
- Given input: “One a penny, two a penny, hot cross buns.”
 - Records output by the map() function would be
 - (“One”, 1), (“a”, 1), (“penny”, 1), (“two”, 1), (“a”, 1), (“penny”, 1), (“hot”, 1), (“cross”, 1), (“buns”, 1).
 - Records output by reduce function would be
 - (“One”, 1), (“a”, 2), (“penny”, 2), (“two”, 1), (“hot”, 1), (“cross”, 1), (“buns”, 1)



Pseudo-code of Word Count

```
map(String record):
```

```
    for each word in record  
        emit(word, 1);
```

```
// First attribute of emit above is called reduce key
```

```
// In effect, group by is performed on reduce key to create a  
// list of values (all 1's in above code). This requires shuffle step  
// across machines.
```

```
// The reduce function is called on list of values in each group
```

```
reduce(String key, List value_list):
```

```
    String word = key
```

```
    int count = 0;
```

```
    for each value in value_list:
```

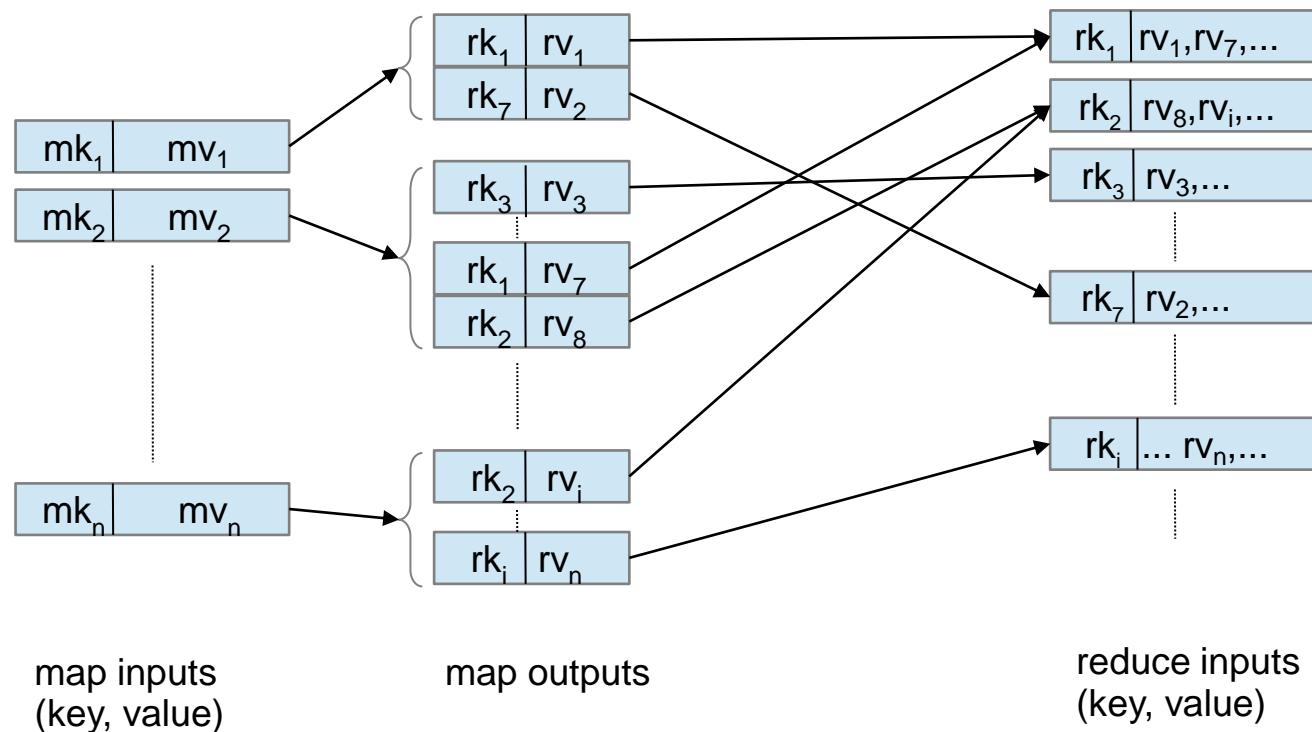
```
        count = count + value
```

```
    Output(word, count);
```



Schematic Flow of Keys and Values

- Flow of keys and values in a map reduce task





Map Reduce vs. Databases

- Map Reduce widely used for parallel processing
 - Google, Yahoo, and 100's of other companies
 - Example uses: compute PageRank, build keyword indices, do data analysis of web click logs,
 - Allows procedural code in map and reduce functions
 - Allows data of any type
- Many real-world uses of MapReduce cannot be expressed in SQL
- But many computations are much easier to express in SQL
 - Map Reduce is cumbersome for writing simple queries



Algebraic Operations

- Current generation execution engines
 - natively support algebraic operations such as joins, aggregation, etc. natively.
 - Allow users to create their own algebraic operators
 - Support trees of algebraic operators that can be executed on multiple nodes in parallel
- E.g. Apache Tez, Spark
 - Tez provides low level API; Hive on Tez compiles SQL to Tez
 - Spark provides more user-friendly API



Algebraic Operations in Spark

- **Resilient Distributed Dataset (RDD) abstraction**
 - Collection of records that can be stored across multiple machines
- RDDs can be created by applying algebraic operations on other RDDs
- RDDs can be lazily computed when needed
- Spark programs can be written in Java/Scala/R
 - Our examples are in Java
- Spark makes use of Java 8 Lambda expressions; the code

```
s -> Arrays.asList(s.split(" ")).iterator()
```

defines unnamed function that takes argument s and executes the expression `Arrays.asList(s.split(" ")).iterator()` on the argument
- Lambda functions are particularly convenient as arguments to map, reduce and other functions



Streaming Data and Applications

- **Streaming data** refers to data that arrives in a continuous fashion
 - Contrast to **data-at-rest**
- Applications include:
 - Stock market: stream of trades
 - e-commerce site: purchases, searches
 - Sensors: sensor readings
 - Internet of things
 - Network monitoring data
 - Social media: tweets and posts can be viewed as a stream
- Queries on streams can be very useful
 - Monitoring, alerts, automated triggering of actions



Querying Streaming Data

Approaches to querying streams:

- **Windowing:** Break up stream into windows, and queries are run on windows
 - Stream query languages support window operations
 - Windows may be based on time or tuples
 - Must figure out when all tuples in a window have been seen
 - Easy if stream totally ordered by timestamp
 - **Punctuations** specify that all future tuples have timestamp greater than some value
- **Continuous Queries:** Queries written e.g. in SQL, output partial results based on stream seen so far; query results updated continuously
 - Have some applications, but can lead to flood of updates



Querying Streaming Data (Cont.)

Approaches to querying streams (Cont.):

- **Algebraic operators on streams:**
 - Each operator consumes tuples from a stream and outputs tuples
 - Operators can be written e.g., in an imperative language
 - Operator may maintain state
- **Pattern matching:**
 - Queries specify patterns, system detects occurrences of patterns and triggers actions
 - **Complex Event Processing (CEP)** systems
 - E.g., Microsoft StreamInsight, Flink CEP, Oracle Event Processing



Stream Processing Architectures

- Many stream processing systems are purely in-memory, and do not persist data
- **Lambda architecture:** split stream into two, one output goes to stream processing system and the other to a database for storage
 - Easy to implement and widely used
 - But often leads to duplication of querying effort, once on streaming system and once in database



Stream Extensions to SQL

- SQL Window functions described in Section 5.5.2
- Streaming systems often support more window types
 - **Tumbling window**
 - E.g., hourly windows, windows don't overlap
 - **Hopping window**
 - E.g., hourly window computed every 20 minutes
 - **Sliding window**
 - Window of specified size (based on timestamp interval or number of tuples) around each incoming tuple
 - **Session window**
 - Groups tuples based on user sessions



Publish Subscribe Systems

- **Publish-subscribe (pub-sub)** systems provide convenient abstraction for processing streams
 - Tuples in a stream are published to a topic
 - Consumers subscribe to topic
- Parallel pub-sub systems allow tuples in a topic to be partitioned across multiple machines
- **Apache Kafka** is a popular parallel pub-sub system widely used to manage streaming data
- More details in book



Graph Data Model

- Graphs are a very general data model
- ER model of an enterprise can be viewed as a graph
 - Every entity is a node
 - Every binary relationship is an edge
 - Ternary and higher degree relationships can be modelled as binary relationships



Graph Data Model (Cont.)

- Graphs can be modelled as relations
 - $\text{node}(ID, \text{label}, \text{node_data})$
 - $\text{edge}(\text{fromID}, \text{toID}, \text{label}, \text{edge_data})$
- Above representation too simplistic
- Graph databases like Neo4J can provide a **graph view of relational schema**
 - Relations can be identified as representing either nodes or edges
- Query languages for graph databases make it
 - easy to express queries requiring edge traversal
 - allow efficient algorithms to be used for evaluation



End of Chapter 10



Chapter 11: Data Analytics

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Chapter 11: Data Analytics

- Overview
- Data Warehousing
- Online Analytical Processing
- Data Mining



Overview

- **Data analytics:** the processing of data to infer patterns, correlations, or models for prediction
- Primarily used to make business decisions
 - Per individual customer
 - E.g., what product to suggest for purchase
 - Across all customers
 - E.g., what products to manufacture/stock, in what quantity
- Critical for businesses today



Overview (Cont.)

- Common steps in data analytics
 - Gather data from multiple sources into one location
 - Data warehouses also integrated data into common schema
 - Data often needs to be **extracted** from source formats, **transformed** to common schema, and **loaded** into the data warehouse
 - Can be done as **ETL (extract-transform-load)**, or **ELT (extract-load-transform)**
 - Generate aggregates and reports summarizing data
 - Dashboards showing graphical charts/reports
 - **Online analytical processing (OLAP) systems** allow interactive querying
 - Statistical analysis using tools such as R/SAS/SPSS
 - Including extensions for parallel processing of big data
 - Build **predictive models** and use the models for decision making



Overview (Cont.)

- Predictive models are widely used today
 - E.g., use customer profile features (e.g. income, age, gender, education, employment) and past history of a customer to predict likelihood of default on loan
 - and use prediction to make loan decision
 - E.g., use past history of sales (by season) to predict future sales
 - And use it to decide what/how much to produce/stock
 - And to target customers
- Other examples of business decisions:
 - What items to stock?
 - What insurance premium to change?
 - To whom to send advertisements?



Overview (Cont.)

- **Machine learning** techniques are key to finding patterns in data and making predictions
- **Data mining** extends techniques developed by machine-learning communities to run them on very large datasets
- The term **business intelligence (BI)** is synonym for data analytics
- The term **decision support** focuses on reporting and aggregation

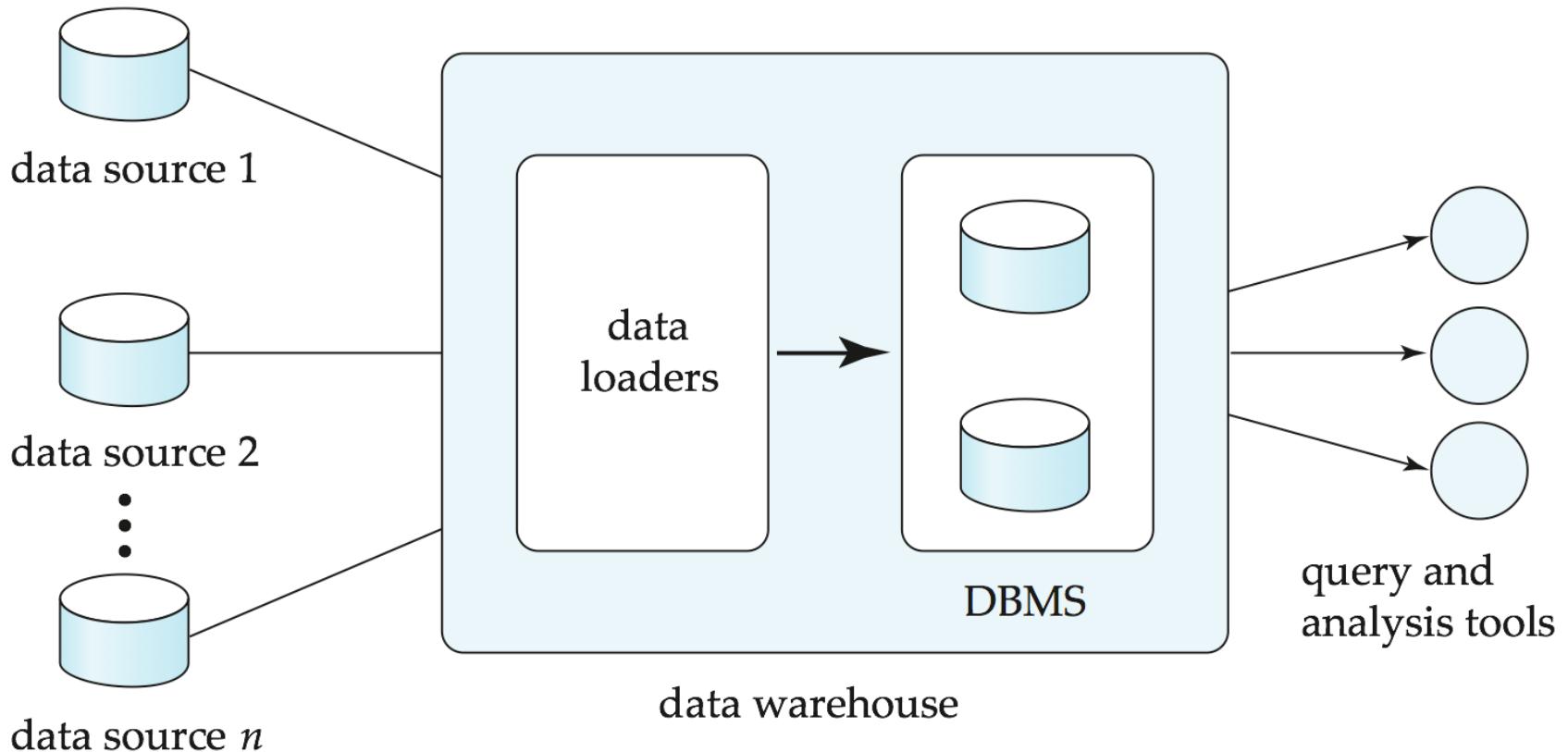


Data Warehousing

- Data sources often store only current data, not historical data
- Corporate decision making requires a unified view of all organizational data, including historical data
- A **data warehouse** is a repository (archive) of information gathered from multiple sources, stored under a unified schema, at a single site
 - Greatly simplifies querying, permits study of historical trends
 - Shifts decision support query load away from transaction processing systems



Data Warehousing





Design Issues

- *When and how to gather data*
 - **Source driven architecture:** data sources transmit new information to warehouse
 - either continuously or periodically (e.g., at night)
 - **Destination driven architecture:** warehouse periodically requests new information from data sources
 - **Synchronous vs asynchronous replication**
 - Keeping warehouse exactly synchronized with data sources (e.g., using two-phase commit) is often too expensive
 - Usually OK to have slightly out-of-date data at warehouse
 - Data/updates are periodically downloaded from online transaction processing (OLTP) systems.
- *What schema to use*
 - Schema integration



More Warehouse Design Issues

- **Data transformation and data cleansing**
 - E.g., correct mistakes in addresses (misspellings, zip code errors)
 - Merge address lists from different sources and **purge** duplicates
- *How to propagate updates*
 - Warehouse schema may be a (materialized) view of schema from data sources
 - View maintenance
- *What data to summarize*
 - Raw data may be too large to store on-line
 - Aggregate values (totals/subtotals) often suffice
 - Queries on raw data can often be transformed by query optimizer to use aggregate values



Data Analysis and OLAP

- **Online Analytical Processing (OLAP)**
 - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- We use the following relation to illustrate OLAP concepts
 - *sales (item_name, color, clothes_size, quantity)*

This is a simplified version of the *sales* fact table joined with the dimension tables, and many attributes removed (and some renamed)



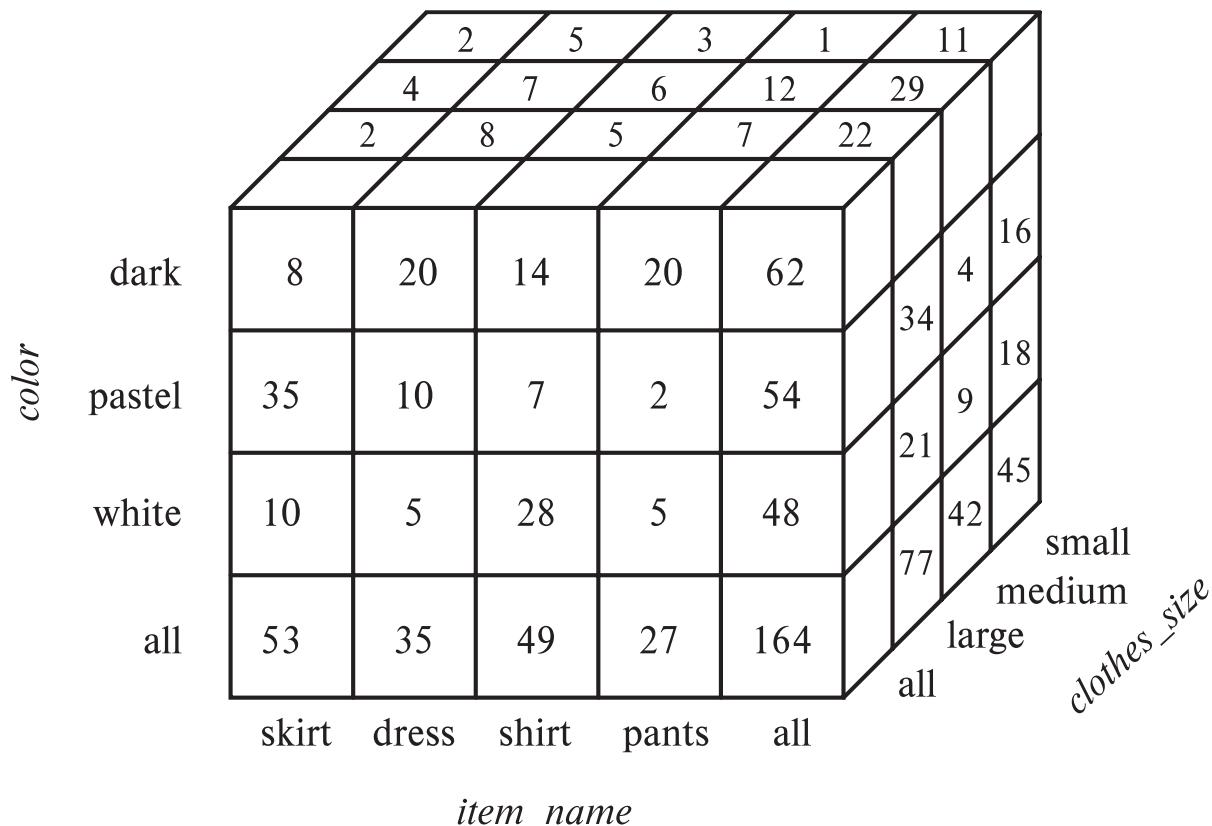
Example sales relation

<i>item_name</i>	<i>color</i>	<i>clothes_size</i>	<i>quantity</i>
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5
...
...



Data Cube

- A **data cube** is a multidimensional generalization of a cross-tab
- Can have n dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube





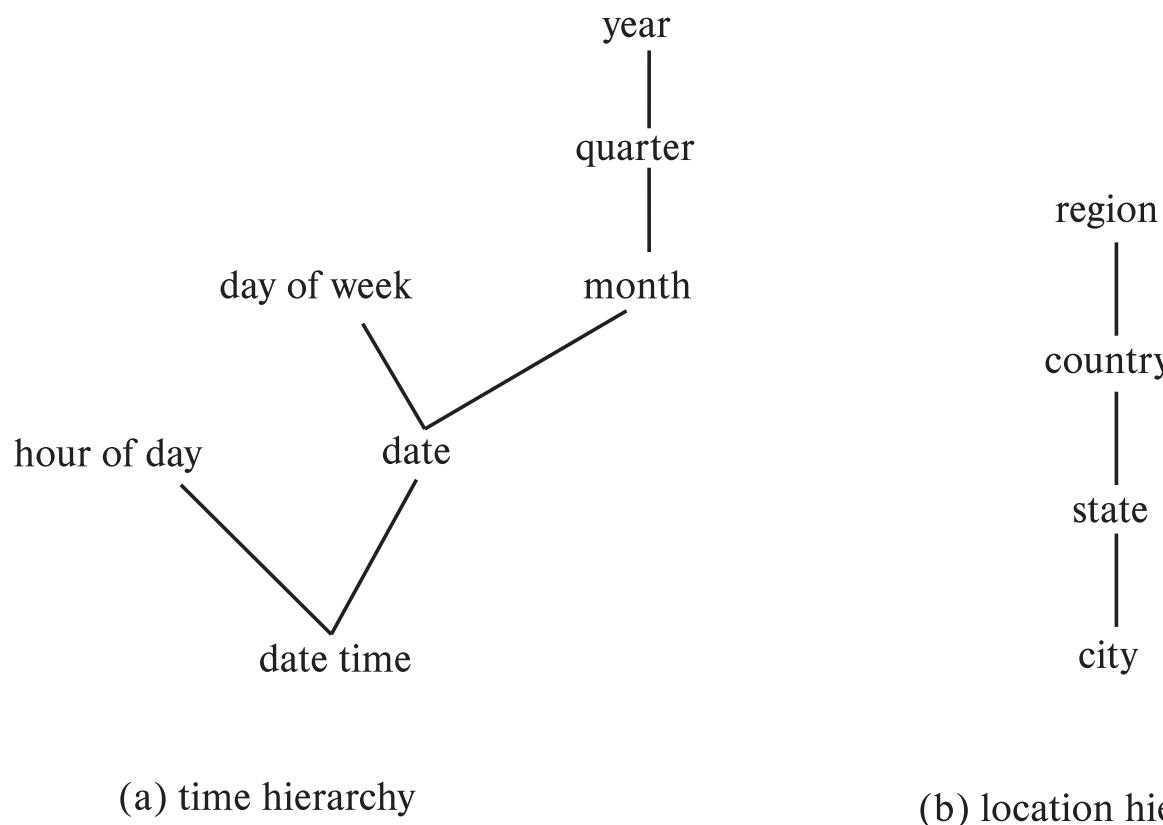
Online Analytical Processing Operations

- **Pivoting:** changing the dimensions used in a cross-tab
 - E.g., moving colors to column names
- **Slicing:** creating a cross-tab for fixed values only
 - E.g., fixing color to white and size to small
 - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.
- **Rollup:** moving from finer-granularity data to a coarser granularity
 - E.g., aggregating away an attribute
 - E.g., moving from aggregates by day to aggregates by month or year
- **Drill down:** The opposite operation - that of moving from coarser-granularity data to finer-granularity data



Hierarchies on Dimensions

- **Hierarchy** on dimension attributes: lets dimensions be viewed at different levels of detail
- E.g., the dimension *datetime* can be used to aggregate by hour of day, date, day of week, month, quarter or year



(a) time hierarchy

(b) location hierarchy



Reporting and Visualization

- **Reporting tools** help create formatted reports with tabular/graphical representation of data
 - E.g., SQL Server reporting services, Crystal Reports
- **Data visualization** tools help create interactive visualization of data
 - E.g., Tableau, FusionChart, plotly, Datawrapper, Google Charts, etc.
 - Frontend typically based on HTML+JavaScript

Acme Supply Company, Inc.
Quarterly Sales Report

Period: Jan. 1 to March 31, 2009

Region	Category	Sales	Subtotal
North	Computer Hardware	1,000,000	1,500,000
	Computer Software	500,000	
	All categories		
South	Computer Hardware	200,000	600,000
	Computer Software	400,000	
	All categories		
	Total Sales		2,100,000

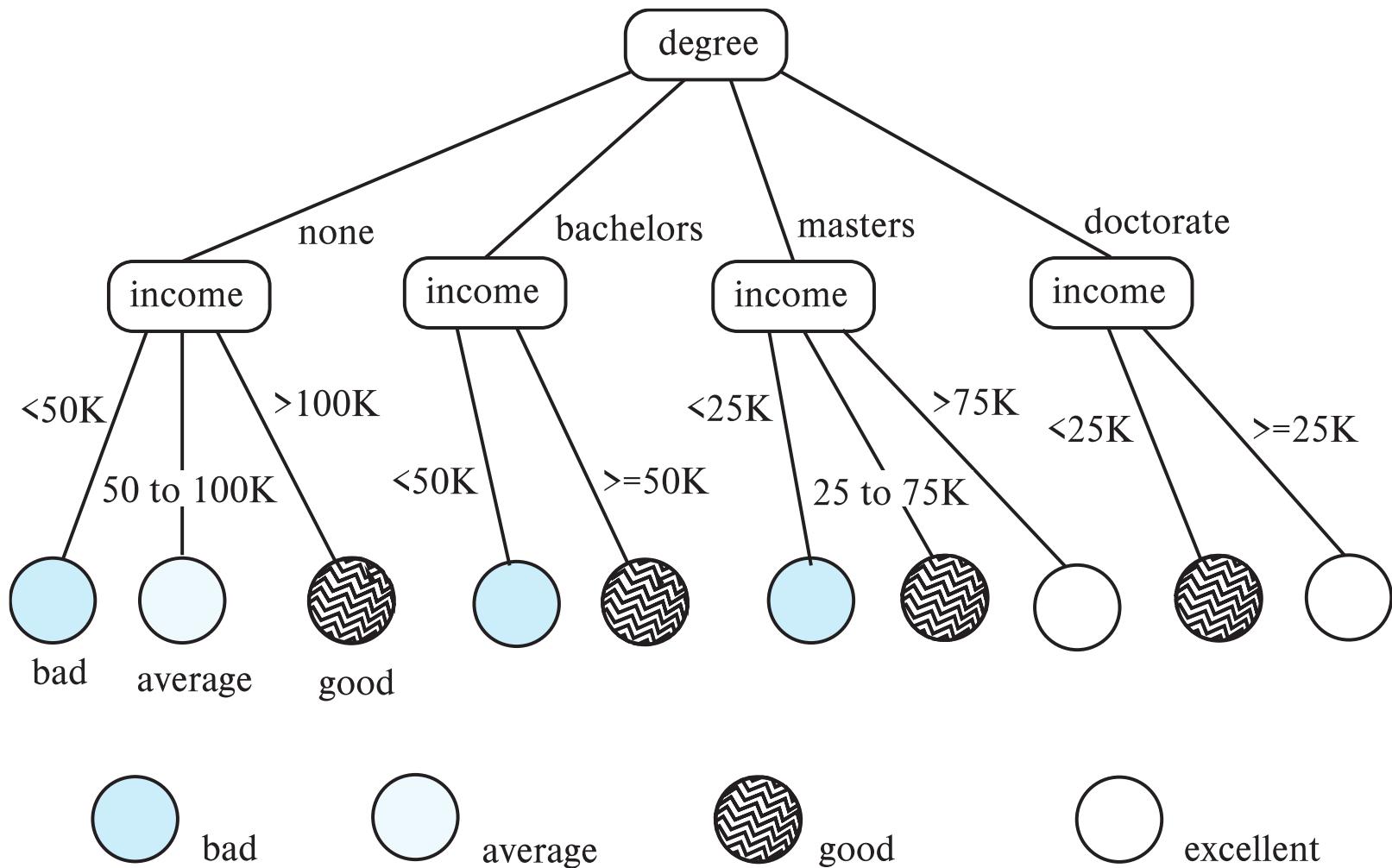


Data Mining

- **Data mining** is the process of semi-automatically analyzing large databases to find useful patterns
 - Similar goals to machine learning, but on very large volumes of data
- Part of the larger area of **knowledge discovery in databases (KDD)**
- Some types of knowledge can be represented as rules
- More generally, knowledge is discovered by applying machine learning techniques on past instances of data, to form a **model**
 - Model is then used to make predictions for new instances



Decision Tree Classifiers





Decision Trees

- Each internal node of the tree partitions the data into groups based on a **partitioning attribute**, and a **partitioning condition** for the node
- Leaf node:
 - all (or most) of the items at the node belong to the same class, or
 - all attributes have been considered, and no further partitioning is possible.
- Traverse tree from top to make a prediction
- Number of techniques for constructing decision tree classifiers
 - We omit details



Bayesian Classifiers

- Bayesian classifiers use **Bayes theorem**, which says

$$p(c_j | d) = \frac{p(d | c_j) p(c_j)}{p(d)}$$

where

$p(c_j | d)$ = probability of instance d being in class c_j ,

$p(d | c_j)$ = probability of generating instance d given class c_j ,

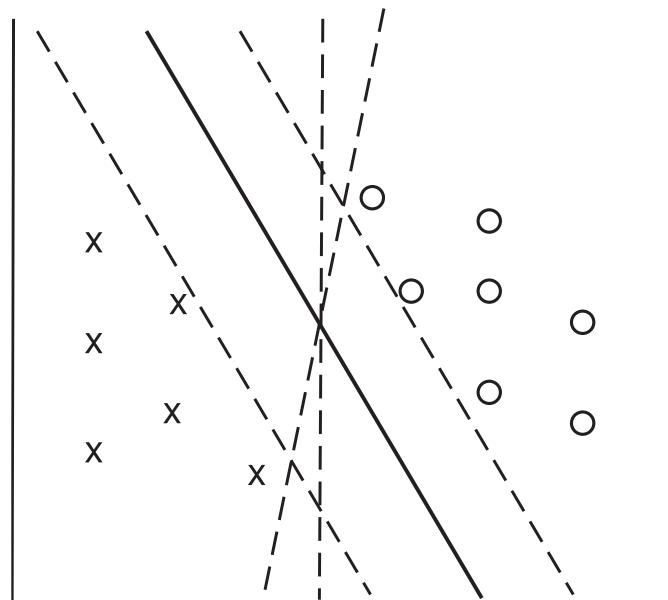
$p(c_j)$ = probability of occurrence of class c_j , and

$p(d)$ = probability of instance d occurring



Support Vector Machine Classifiers

- Simple 2-dimensional example:
 - Points are in two classes
 - Find a line (**maximum margin line**) s.t. line divides two classes, and distance from nearest point in either class is maximum





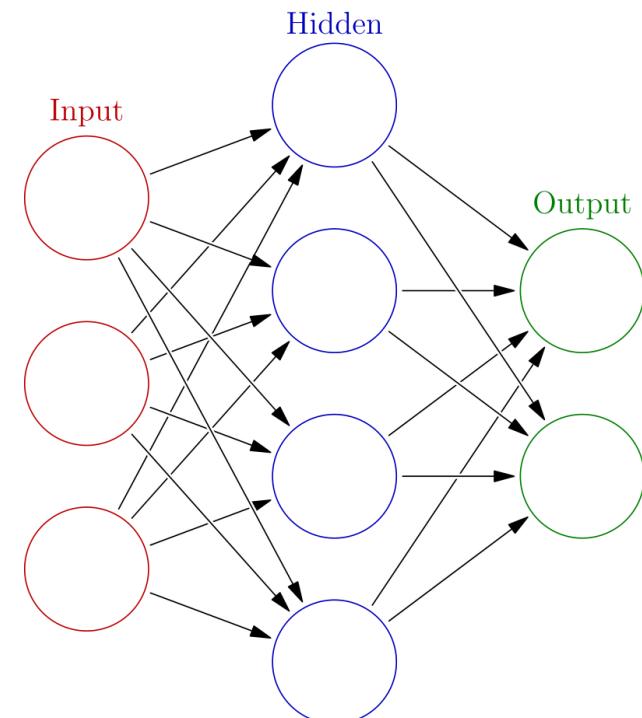
Support Vector Machine

- In n-dimensions points are divided by a plane, instead of a line
- SVMs can be used separators that are curve, not necessarily linear, by transforming points before classification
 - Transformation functions may be non-linear and are called kernel functions
 - Separator is a plane in the transformed space, but maps to curve in original space
- There may not be an exact planar separator for a given set of points
 - Choose plane that best separates points
- N-ary classification can be done by N binary classifications
 - In class i vs. not in class i .



Neural Network Classifiers

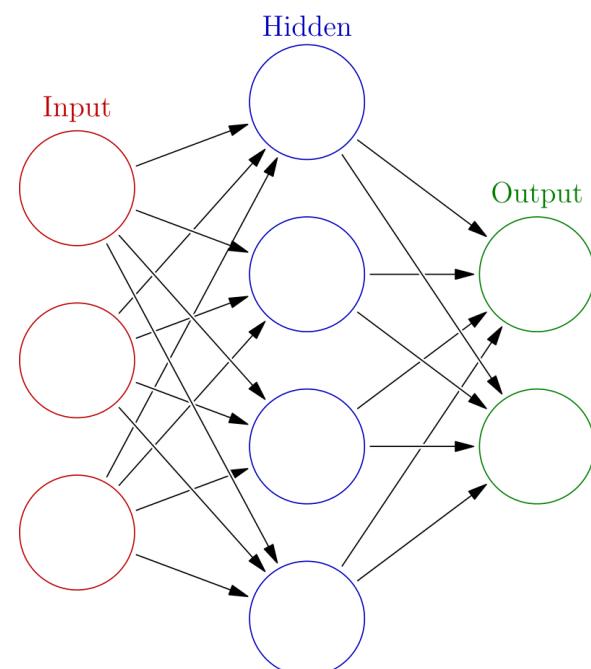
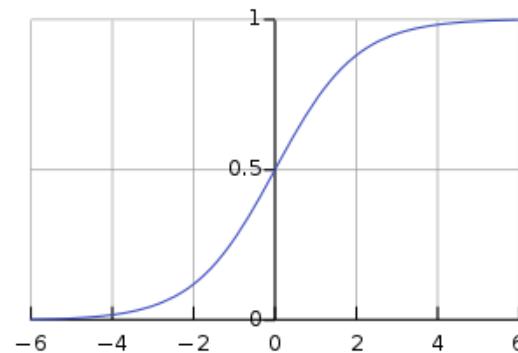
- Neural network has multiple layers
 - Each layer acts as input to next later
- First layer has input nodes, which are assigned values from input attributes
- Each node combines values of its inputs using some weight function to compute its value
 - Weights are associated with edges
- For classification, each output value indicates likelihood of the input instance belonging to that class
 - Pick class with maximum likelihood
- Weights of edges are key to classification
- Edge weights are learnt during training phase





Neural Network Classifiers

- Value of a node may be linear combination of inputs, or may be a non-linear function
 - E.g., sigmoid function
- **Backpropagation algorithm** works as follows
 - Weights are set randomly initially
 - Training instances are processed one at a time
 - Output is computed using current weights
 - If classification is wrong, weights are tweaked to get a higher score for the correct class





Neural Networks (Cont.)

- **Deep neural networks** have a large number of layers with large number of nodes in each layer
- **Deep learning** refers to training of deep neural network on very large numbers of training instances
- Each layer may be connected to previous layers in different ways
 - Convolutional networks used for image processing
 - More complex architectures used for text processing, and machine translation, speech recognition, etc.
- Neural networks are a large area in themselves
 - Further details beyond scope of this chapter



Regression

- Regression deals with the prediction of a value, rather than a class.
 - Given values for a set of variables, X_1, X_2, \dots, X_n , we wish to predict the value of a variable Y .
- One way is to infer coefficients $a_0, a_1, a_2, \dots, a_n$ such that
$$Y = a_0 + a_1 * X_1 + a_2 * X_2 + \dots + a_n * X_n$$
- Finding such a linear polynomial is called **linear regression**.
 - In general, the process of finding a curve that fits the data is also called **curve fitting**.
- The fit may only be approximate
 - because of noise in the data, or
 - because the relationship is not exactly a polynomial
- Regression aims to find coefficients that give the best possible fit.



Association Rules

- Retail shops are often interested in associations between different items that people buy.
 - Someone who buys bread is quite likely also to buy milk
 - A person who bought the book *Database System Concepts* is quite likely also to buy the book *Operating System Concepts*.
- Associations information can be used in several ways.
 - E.g. when a customer buys a particular book, an online shop may suggest associated books.
- **Association rules:**

bread \Rightarrow *milk* *DB-Concepts, OS-Concepts* \Rightarrow Networks

 - Left hand side: **antecedent**, right hand side: **consequent**
 - An association rule must have an associated **population**; the population consists of a set of **instances**
 - E.g. each transaction (sale) at a shop is an instance, and the set of all transactions is the population



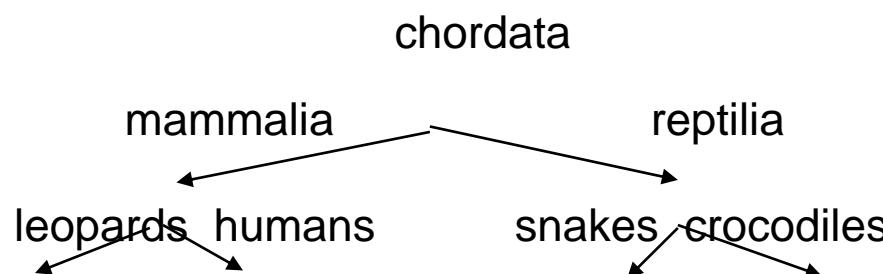
Association Rules (Cont.)

- Rules have an associated support, as well as an associated confidence.
- **Support** is a measure of what fraction of the population satisfies both the antecedent and the consequent of the rule.
 - E.g., suppose only 0.001 percent of all purchases include milk and screwdrivers. The support for the rule $milk \Rightarrow screwdrivers$ is low.
- **Confidence** is a measure of how often the consequent is true when the antecedent is true.
 - E.g., the rule $bread \Rightarrow milk$ has a confidence of 80 percent if 80 percent of the purchases that include bread also include milk.
- We omit further details, such as how to efficiently infer association rules



Clustering

- **Clustering:** Intuitively, finding clusters of points in the given data such that similar points lie in the same cluster
- Can be formalized using distance metrics in several ways
 - Group points into k sets (for a given k) such that the average distance of points from the centroid of their assigned group is minimized
 - Centroid: point defined by taking average of coordinates in each dimension.
 - Another metric: minimize average distance between every pair of points in a cluster
- **Hierarchical clustering:** example from biological classification
 - (the word classification here does not mean a prediction mechanism)





Other Types of Mining

- **Text mining:** application of data mining to textual documents
- **Sentiment analysis**
 - E.g., learn to predict if a user review is positive or negative about a product
- **Information extraction**
 - Create structured information from unstructured textual description or semi-structured data such as tabular displays
- **Entity recognition** and **disambiguation**
 - E.g., given text with name “Michael Jordan” does the name refer to the famous basketball player or the famous ML expert
- **Knowledge graph** (see Section 8.4)
 - Can be constructed by information extraction from different sources, such as Wikipedia

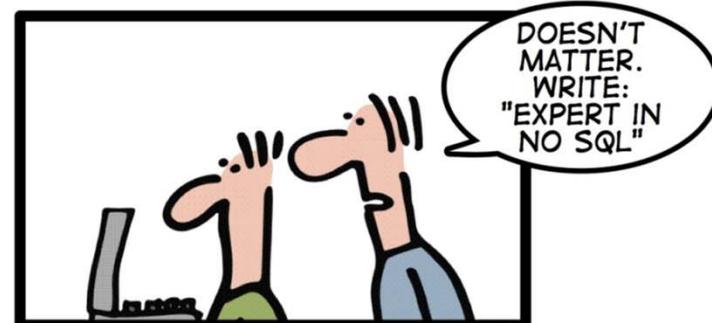


End of Chapter

Introduction to **NoSQL** Databases

Slides based on multiple sources

HOW TO WRITE A CV



Leverage the NoSQL boom

Outline

- SQL Databases
 - SQL Standard
 - SQL Characteristics
- NoSQL Databases
 - NoSQL Definition
 - General Characteristics
 - NoSQL Database Types
 - NoSQL Database Examples

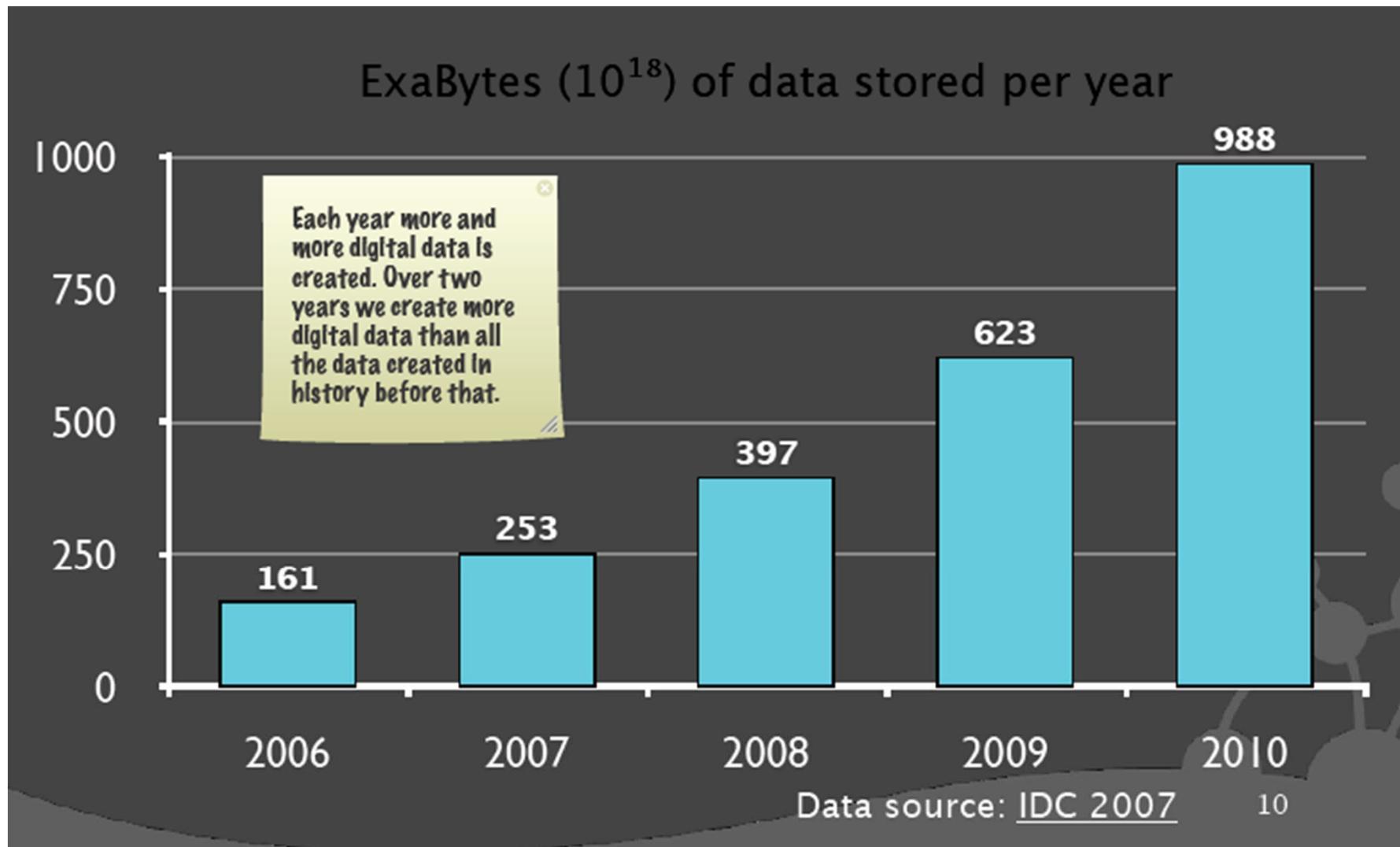
What is in the Name: NoSQL

- Is it ~~No to SQL~~ ?
- Not Only SQL

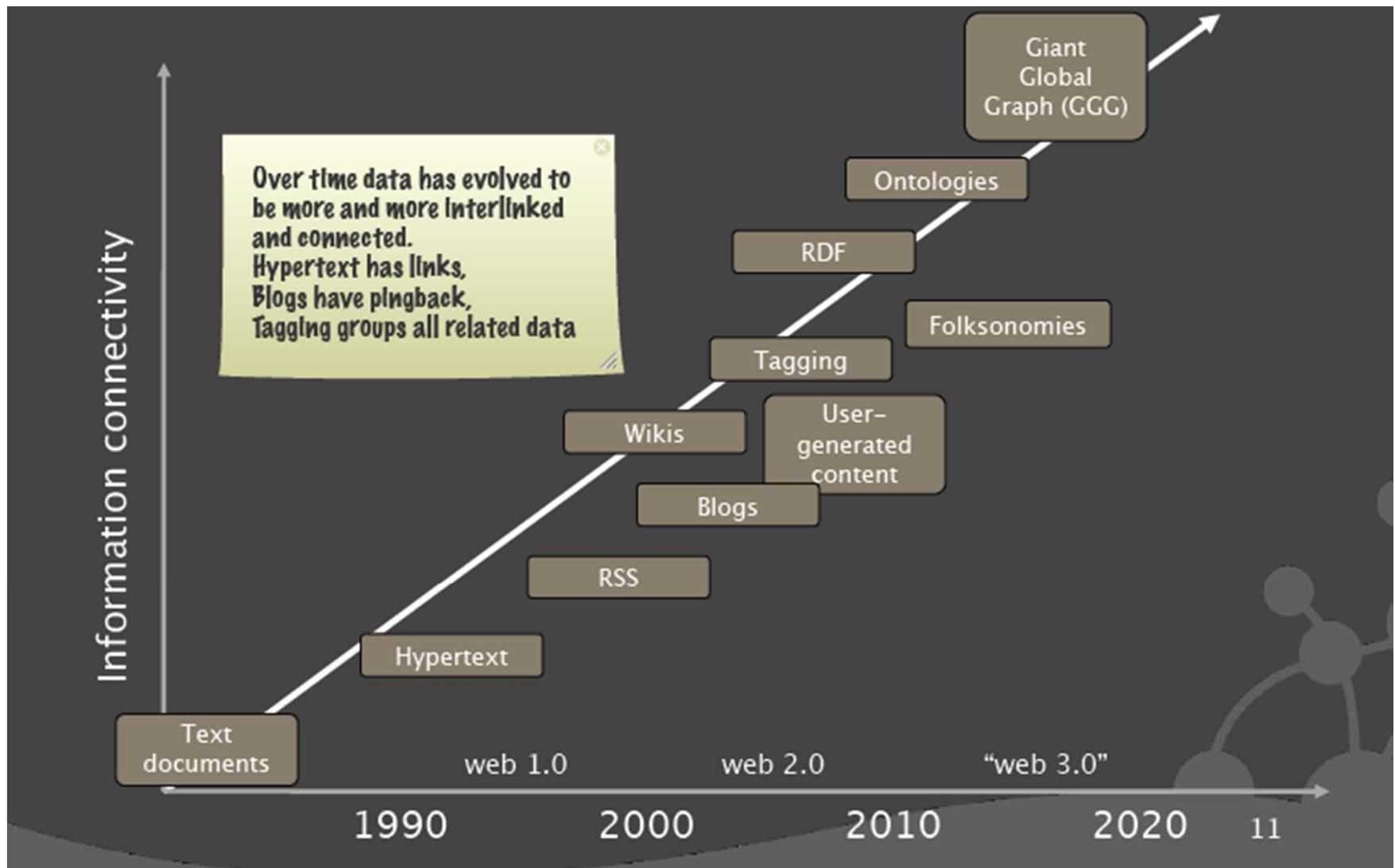
NOSQL - Defined by what it is Not

- “Any database that is not a Relational Database”
- The term was coined at a meetup with the creators behind some
- prominent emerging databases
- “Non-Relational Databases” might be more correct
- - But it’s a mouthful!
 - ... then there was a conference ...
 - ... and a mailing list ...
 - ... the name caught on ...
 - ... then there were more conferences ...
 - ... and here we are!

Why NoSQL: Trend 1 Data Size



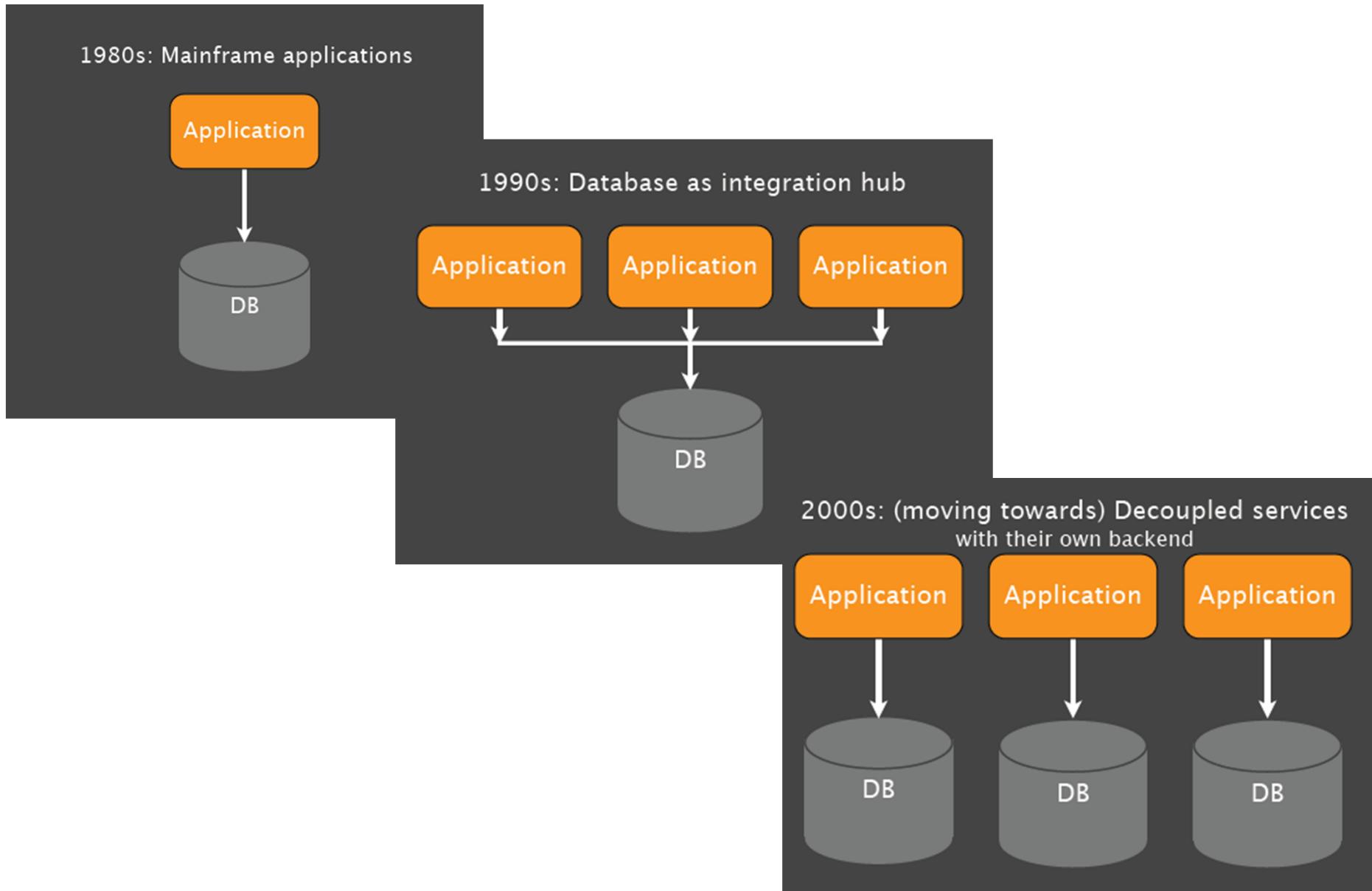
Why NoSQL: Trend 2 Connectedness



Why NoSQL: Trend 3 Semi-structure

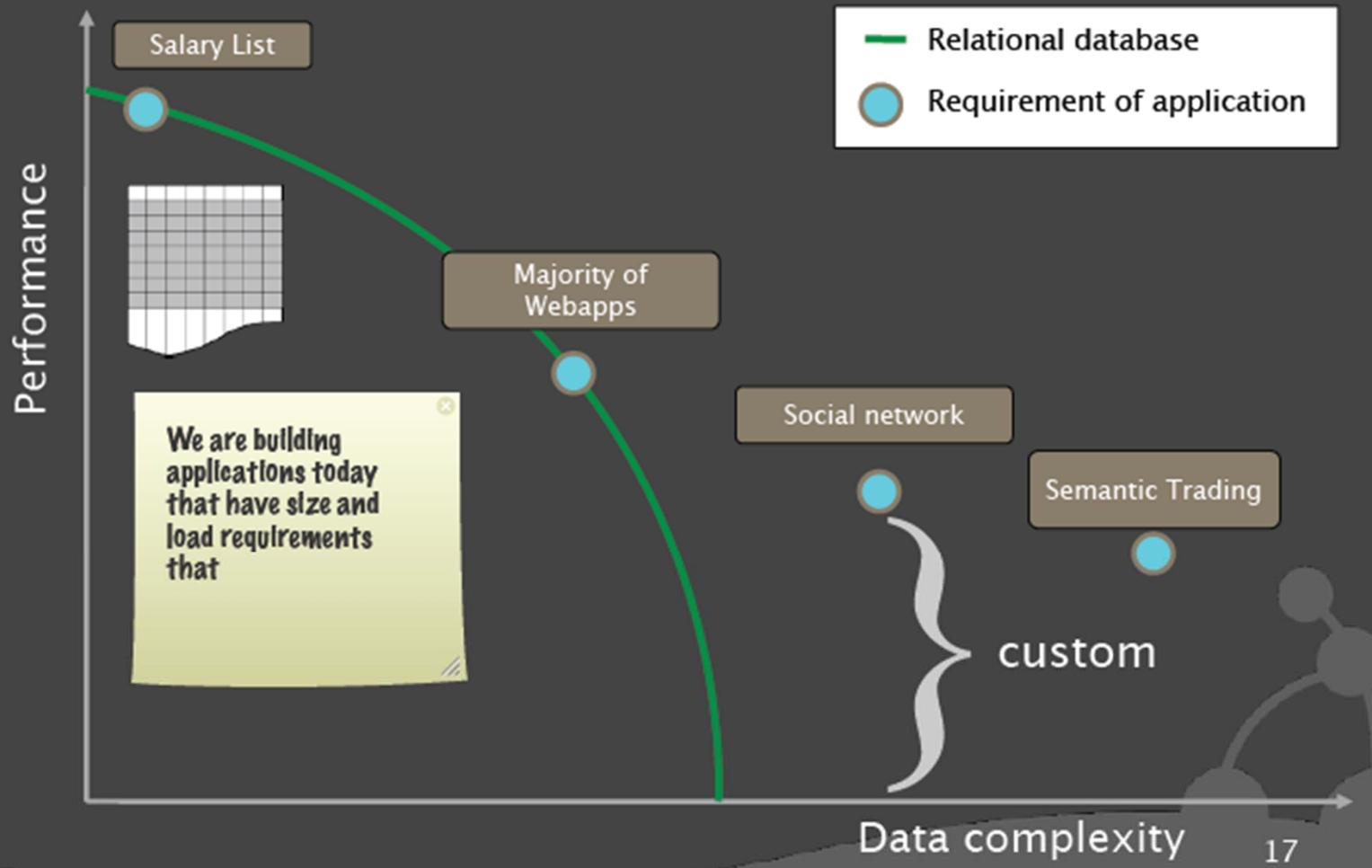
- Individualization of content
- In the salary lists of the 1970s, all elements had exactly one job
- In the salary lists of the 2000s, we need 5 job columns! Or 8? Or 15?
- All encompassing “entire world views”
- Store more data about each entity
- Trend accelerated by the decentralization of content generation that is the hallmark of the age of participation (“web 2.0”)

Why NoSQL: Trend 4 Architecture



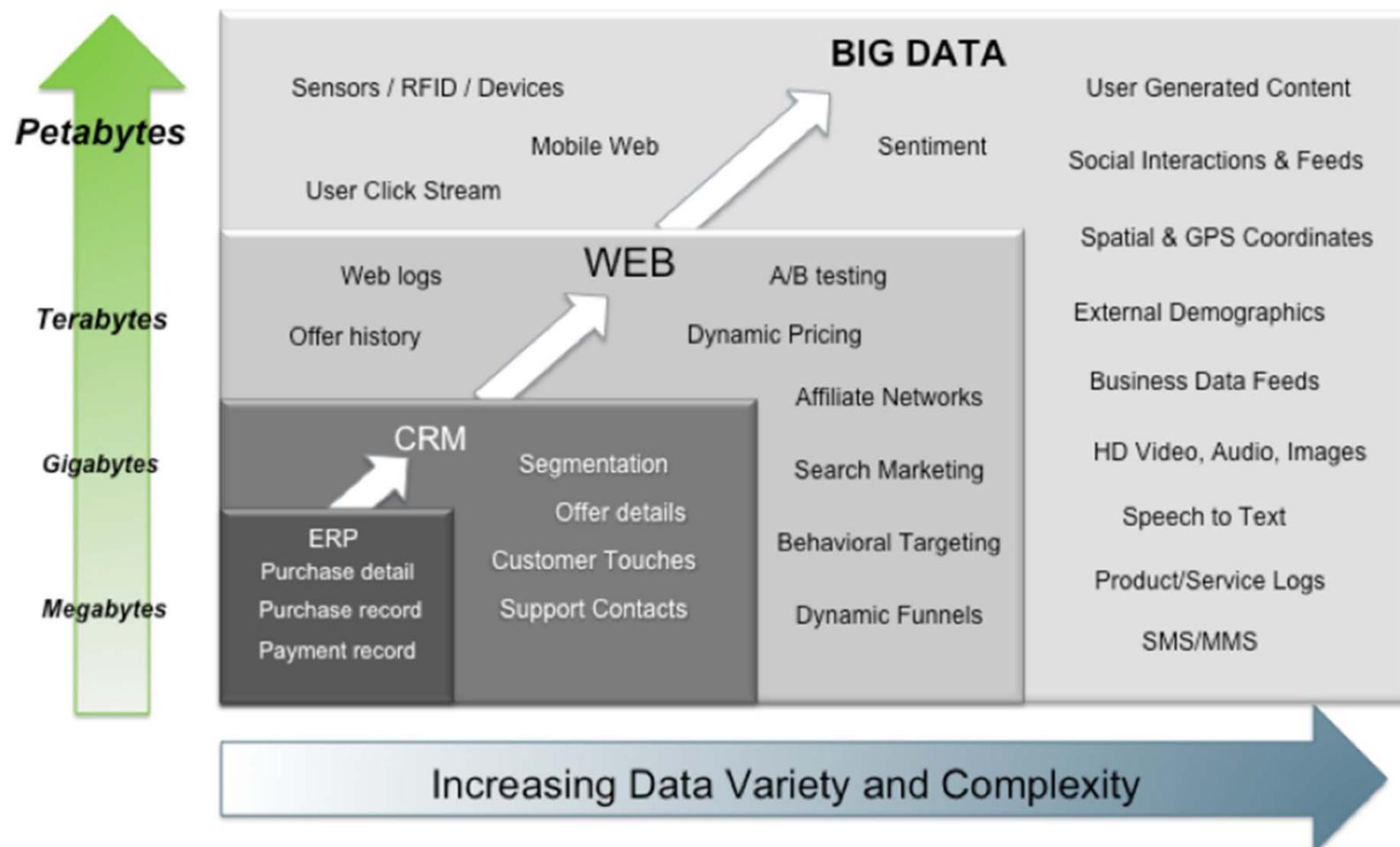
Limitations of RDBMS

RDBMS performance



Evolution of NoSQL DBs

Big Data = Transactions + Interactions + Observations



Source: Contents of above graphic created in partnership with Teradata, Inc.

Why NoSQL

- ACID doesn't scale well
- Web apps have different needs (than the apps that RDBMS were designed for)
 - Low and predictable response time (latency)
 - Scalability & elasticity (at low cost!)
 - High availability
 - Flexible schemas / semi-structured data
 - Geographic distribution (multiple datacenters)
- Web apps can (usually) do without
 - Transactions / strong consistency / integrity
 - Complex queries

NoSQL use cases

1. Massive data volumes
 - Massively distributed architecture required to store the data
 - Google, Amazon, Yahoo, Facebook – 10-100K servers
2. Extreme query workload
3. Impossible to efficiently do joins at that scale with an RDBMS
 - Schema evolution
 - Schema flexibility (migration) is not trivial at large scale
 - Schema changes can be gradually introduced with NoSQL

Dis / Advantages of NoSQL

- Advantages
 - Massive scalability
 - High availability
 - Lower cost (than competitive solutions at that scale)
 - (usually) predictable elasticity
 - Schema flexibility, sparse & semi-structured data
- Disadvantages
 - Limited query capabilities (so far)
 - Eventual consistency is not intuitive to program for
 - Makes client applications more complicated
 - No standardization
 - Portability might be an issue
 - Insufficient access control

SQL Characteristics

- Data stored in columns and tables
- Relationships represented by data
- Data Manipulation Language
- Data Definition Language
- Transactions
- Abstraction from physical layer

SQL Physical Layer Abstraction

- Applications specify what, not how
- Query optimization engine
- Physical layer can change without modifying applications
 - Create indexes to support queries
 - In Memory databases

Data Manipulation Language (DML)

- Data manipulated with Select, Insert, Update, & Delete statements
 - `Select T1.Column1, T2.Column2 ...
From Table1, Table2 ...
Where T1.Column1 = T2.Column1 ...`
- Data Aggregation
- Compound statements
- Functions and Procedures
- Explicit transaction control

Data Definition Language

- Schema defined at the start
- Create Table (Column1 Datatype1, Column2 Datatype 2, ...)
- Constraints to define and enforce relationships
 - Primary Key
 - Foreign Key
 - Etc.
- Triggers to respond to Insert, Update , & Delete
- Stored Modules
- Alter ...
- Drop ...
- Security and Access Control

Transactions – ACID Properties

- Atomic – All of the work in a transaction completes (commit) or none of it completes
- Consistent – A transaction transforms the database from one consistent state to another consistent state. Consistency is defined in terms of constraints.
- Isolated – The results of any changes made during a transaction are not visible until the transaction has committed.
- Durable – The results of a committed transaction survive failures

Brewer's CAP Theorem

(E. Brewer, N. Lynch)

A distributed system can support only two of the following characteristics:

- Consistency
- Availability
- Partition tolerance

.

Consistency

- all nodes see the same data at the same time
- client perceives that a set of operations has occurred all at once
- More like Atomic in ACID transaction properties

Availability

- Node failures do not prevent survivors from continuing to operate
- Every operation must terminate in an intended response

Partition Tolerance

- the system continues to operate despite arbitrary message loss
- Operations will complete, even if individual components are unavailable

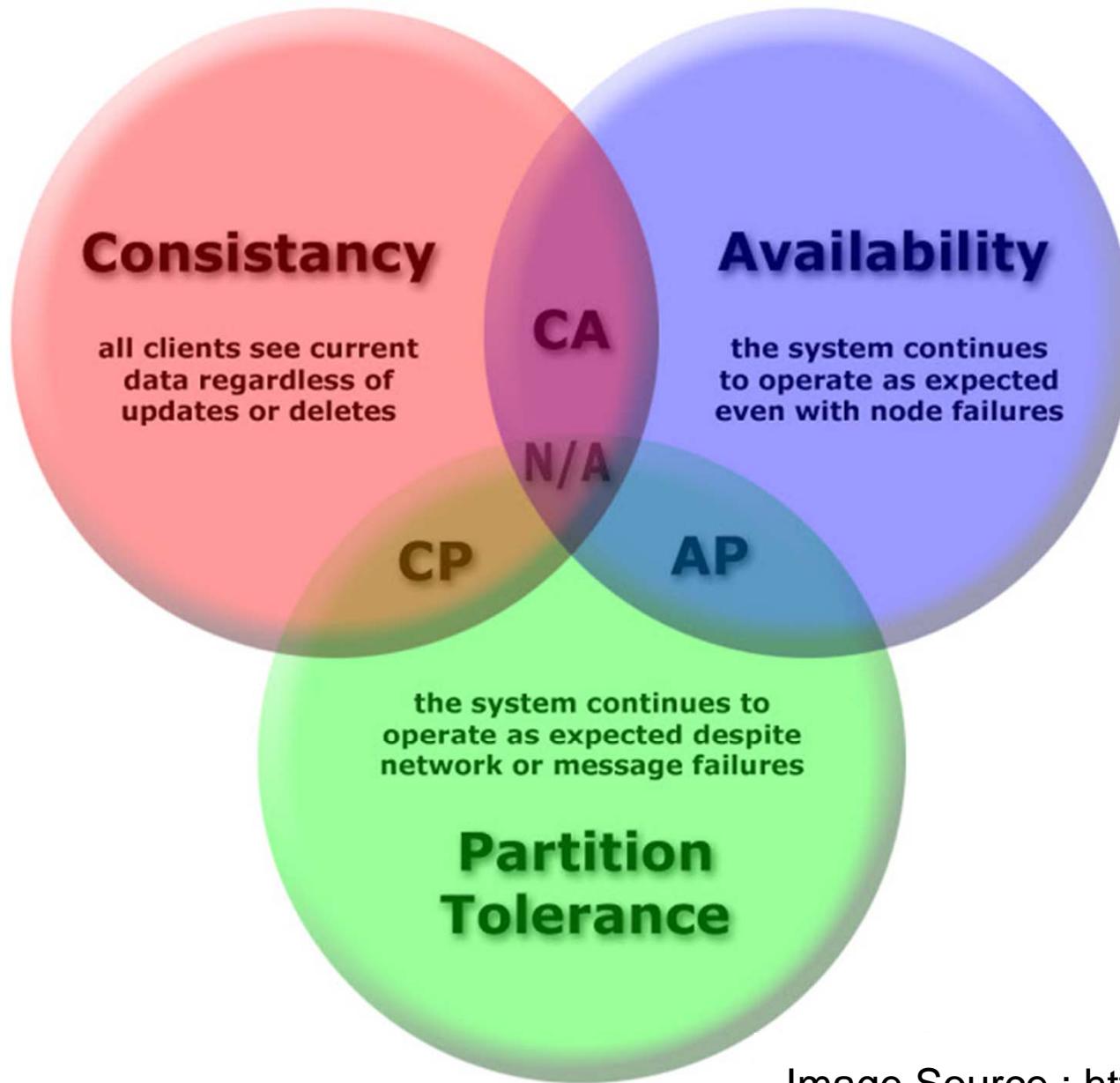


Image Source : <http://blog.nosqltips.com>

BASE Transactions

- Acronym contrived to be the opposite of ACID
 - Basically Available,
 - Soft state,
 - Eventually Consistent
- Characteristics
 - Weak consistency – stale data OK
 - Availability first
 - Best effort
 - Approximate answers OK
 - Aggressive (optimistic)
 - Simpler and faster

BASE Model

Focuses on Partition tolerance and availability and throws consistency out the window

- Basic Availability :
 - This constraint states that the system does guarantee the availability of the data as regards CAP Theorem
- Soft State :
 - The state of the system could change over time and called as ‘soft’ state
- Eventual Consistency :
 - The system will eventually become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

NoSQL Database Types

Discussing NoSQL databases is complicated because there are a variety of types:

- Column Store – Each storage block contains data from only one column
- Document Store – stores documents made up of tagged elements
- Key-Value Store – Hash table of keys

Other Non-SQL Databases

- XML Databases
- Graph Databases
- Codasyl Databases
- Object Oriented Databases
- Etc...

NoSQL Example: Column Store

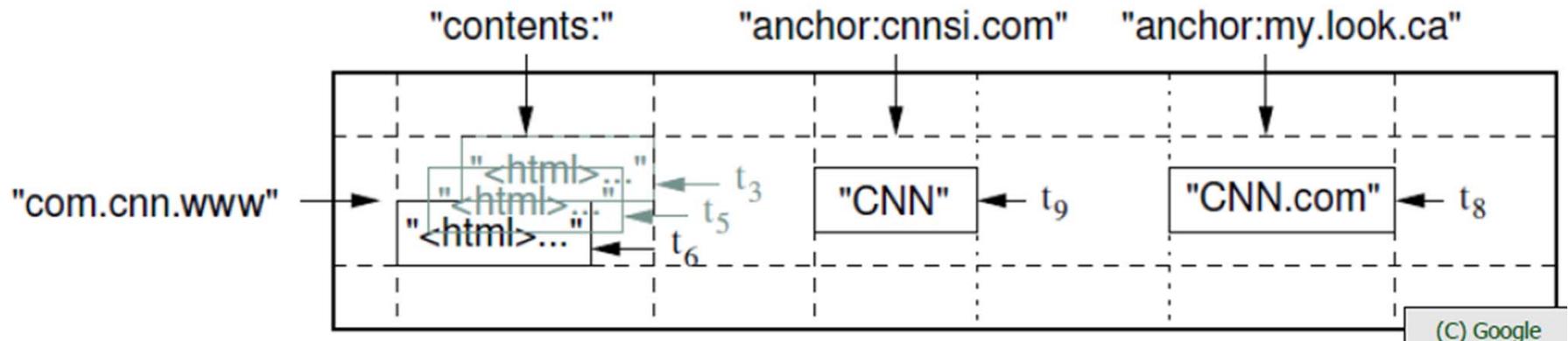
- Each storage block contains data from only one column
- Example: Hadoop/Hbase
 - <http://hadoop.apache.org/>
 - Yahoo, Facebook
- Example: Ingres VectorWise
 - Column Store integrated with an SQL database
 - <http://www.ingres.com/products/vectorwise>

Column Store Comments

- More efficient than row (or document) store if:
 - Multiple row/record/documents are inserted at the same time so updates of column blocks can be aggregated
 - Retrievals access only some of the columns in a row/record/document

Ex. Column Store Google BigTable

- Google, ~2006
- Sparse, distributed, persistent multidimensional sorted map
- $(row, column, timestamp)$ dimensions, value is *string*
- Key features
 - Hybrid row/column store
 - Single master (stand-by replica)
 - Versioning



Google BigTable (Data Model)

- *Row*
 - Keys are arbitrary strings
 - Data is sorted by row key
- *Tablet*
 - Row range is dynamically partitioned into tablets (sequence of rows)
 - Range scans are very efficient
 - Row keys should be chosen to improve *locality* of data access
- *Column, Column Family*
 - Column keys are arbitrary strings, unlimited number of columns
 - Column keys can be grouped into families
 - Data in a CF is stored and compressed together (Locality Groups)
 - Access control on the CF level

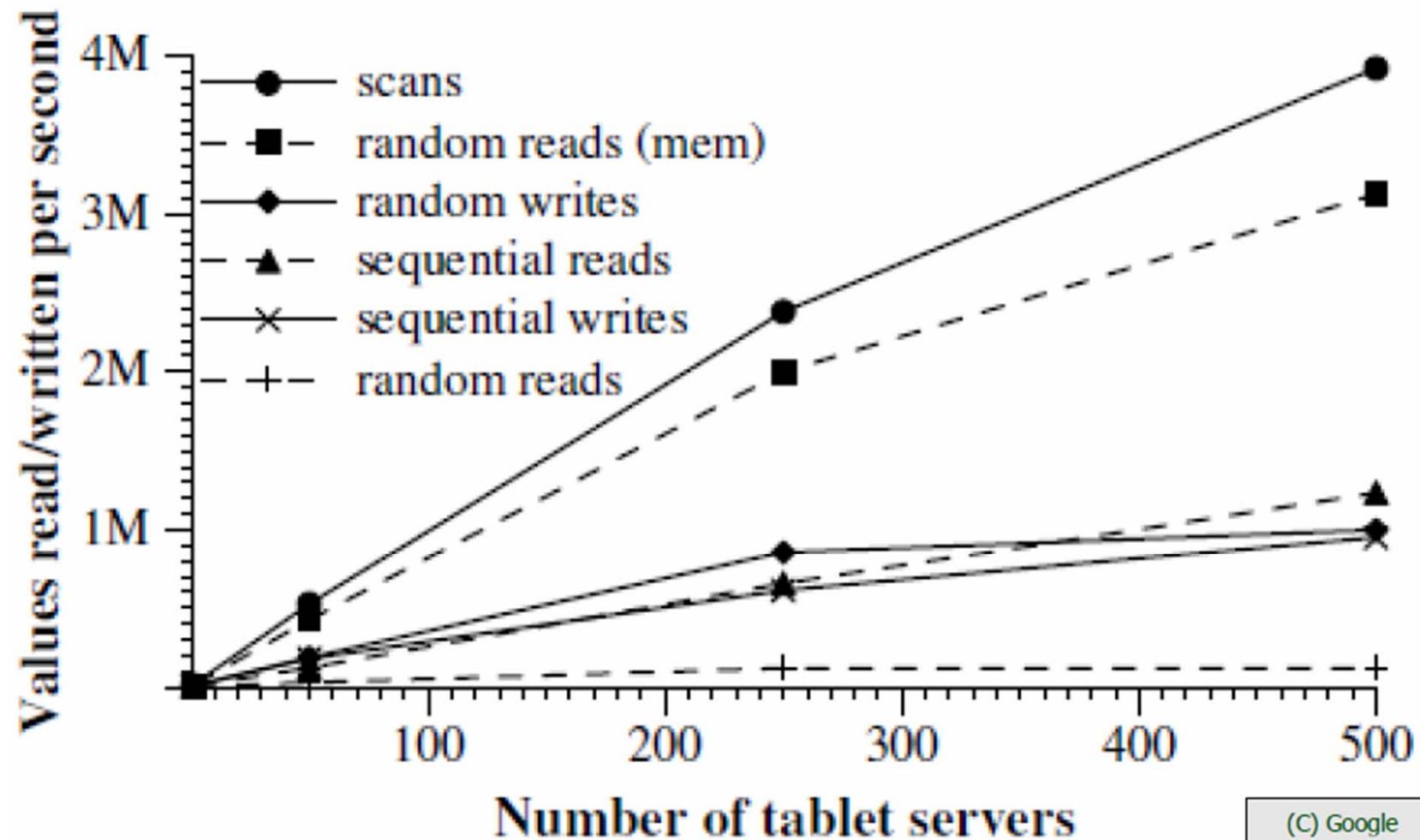
Google BigTable (Data Model^{Cont.})

- *Timestamps*
 - Each cell has multiple versions
 - Can be manually assigned
- *Versioning*
 - Automated garbage collection
 - Retain *last N* versions
 - Retain versions *newer than TS*
- *Architecture*
 - Data stored on *GFS*
 - Relies on *Chubby* (distributed lock service, Paxos)
 - 1 *Master server*
 - thousands of *Tablet servers*

Google BigTable (Architecture)

- *Master server*
 - Assign tablets to Tablet Servers
 - Balance TS load
 - Garbage collection
 - Schema management
 - Client data does **not** move through the MS (directly through TS)
 - Tablet location **not** handled by MS
- *Tablet server (many)*
 - thousands of tablets per TS
 - Manages Read / Write / Split of its tablets

Google BigTable (Performance)

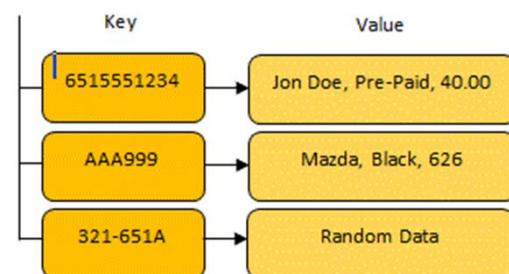


NoSQL Examples: Key-Value Store

- Hash tables of Keys
- Values stored with Keys
- Fast access to small data values
- Example – Project-Voldemort
 - <http://www.project-voldemort.com/>
 - Linkedin
- Example – MemCacheDB
 - <http://memcached.org/>
 - Backend storage is Berkeley-DB

Key-Value Store

- Store items as alpha-numeric identifiers(keys) and associated values in simple, standalone tables (Hash table)
- The values can be simple text strings or more complex list and sets.
- Query can usually only be performed against keys and it limited to exact matches
- Primary Use :
 - Manage user profile or session or retrieving product names
 - Amazon core services use Dynamo as distributed data store



Example Key Value Store: Dynamo

- Amazon, ~2007
- P2P key-value store
 - Object versioning
 - Consistent hashing
 - Gossip – membership & failure detection
 - Quorum reads
- Requirements & assumptions
 - *Simple query model* (unique keys, blobs, no schema, no multi-access)
 - *Scale out* (elasticity)
 - *Eventual consistency* (improved availability)
 - *Decentralized & symmetric* (P2P), *heterogeneous* (load distribution)
 - *Low latency / high throughput*
 - *Internal service* (no security model)

NoSQL Example: Document Store

- Example: CouchDB
 - <http://couchdb.apache.org/>
 - BBC
- Example: MongoDB
 - <http://www.mongodb.org/>
 - Foursquare, Shutterfly
- JSON – JavaScript Object Notation

Document Store Example: CouchDB

- Schema-free, document oriented database
 - Documents stored in JSON format (XML in old versions)
 - B-tree storage engine
 - MVCC model, no locking
 - no joins, no PK/FK (UUIDs are auto assigned)
 - Implemented in Erlang
 - 1st version in C++, 2nd in Erlang and 500 times more scalable
(source: “Erlang Programming” by Cesarini & Thompson)
 - Replication (incremental)
- Documents
 - UUID, version
 - Old versions retained

CouchDB JSON Example

```
{  
    "_id": "guid goes here",  
    "_rev": "314159",  
  
    "type": "abstract",  
  
    "author": "Keith W. Hare"  
  
    "title": "SQL Standard and NoSQL Databases",  
  
    "body": "NoSQL databases (either no-SQL or Not Only SQL)  
            are currently a hot topic in some parts of  
            computing.",  
    "creation_timestamp": "2011/05/10 13:30:00 +0004"  
}
```

CouchDB JSON Tags

- "_id"
 - GUID – Global Unique Identifier
 - Passed in or generated by CouchDB
- "_rev"
 - Revision number
 - Versioning mechanism
- "type", "author", "title", etc.
 - Arbitrary tags
 - Schema-less
 - Could be validated after the fact by user-written routine

MapReduce on CouchDB

Map Reduce Views

Docs

```
{ "user" : "Chris",
  "points" : 3 }
{ "user" : "Joe",
  "points" : 10 }
{ "user" : "Alice",
  "points" : 5 }
{ "user" : "Mary",
  "points" : 9 }
{ "user" : "Bob",
  "points" : 7 }
```

Map

```
function(doc) {
  if (doc.user && doc.points) {
    emit(doc.user, doc.points);
  }
}
```

```
{ "key" : "Alice", "value" : 5 }
{ "key" : "Bob", "value" : 7 }
{ "key" : "Chris", "value" : 3 }
{ "key" : "Joe", "value" : 10 }
{ "key" : "Mary", "value" : 9 }
```

Reduce

```
function(keys, values, rereduce) {
  return sum(values);
}
```

Alice ... Chris: 15
Everyone: 34

MapReduce

- Technique for indexing and searching large data volumes
- Two Phases, Map and Reduce
 - Map
 - Extract sets of Key-Value pairs from underlying data
 - Potentially in Parallel on multiple machines
 - Reduce
 - Merge and sort sets of Key-Value pairs
 - Results may be useful for other searches

MapReduce

- Map Reduce techniques differ across products
- Implemented by application developers, not by underlying software

Map Reduce Patent

Google granted US Patent 7,650,331, January 2010
System and method for efficient large-scale data processing

A large-scale data processing system and method includes one or more application-independent map modules configured to read input data and to apply at least one **application-specific map operation** to the input data to produce intermediate data values, wherein the map operation is automatically parallelized across multiple processors in the parallel processing environment. A plurality of intermediate data structures are used to store the intermediate data values. One or more application-independent reduce modules are configured to retrieve the intermediate data values and to apply at least one **application-specific reduce operation** to the intermediate data values to provide output data.

NoSQL “Definition”

From www.nosql-database.org:

Next Generation Databases mostly addressing some of the points: being **non-relational, distributed, open-source** and **horizontal scalable**. The original intention has been **modern web-scale databases**.

The movement began early 2009 and is growing rapidly. Often more characteristics apply as: **schema-free, easy replication support, simple API, eventually consistent / BASE** (not ACID), a **huge data amount**, and more.

NoSQL Distinguishing Characteristics

- Large data volumes
 - Google’s “big data”
- Scalable replication and distribution
 - Potentially thousands of machines
 - Potentially distributed around the world
- Queries need to return answers quickly
- Mostly query, few updates
- Asynchronous Inserts & Updates
- Schema-less
- ACID transaction properties are not needed – BASE
- CAP Theorem
- Mostly Open Source development

Storing and Modifying Data

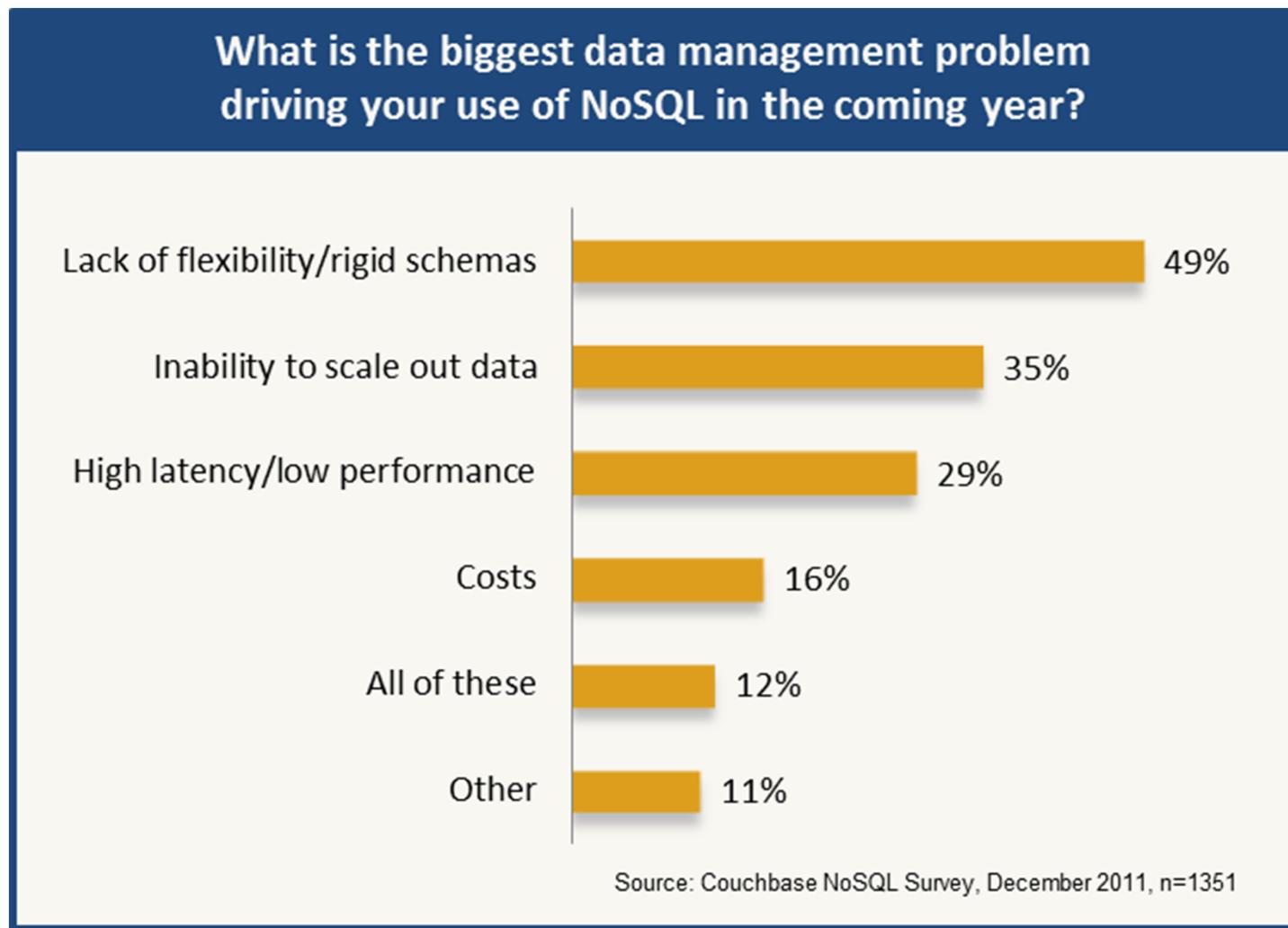
- Syntax varies
 - HTML
 - Java Script
 - Etc.
- Asynchronous – Inserts and updates do not wait for confirmation
- Versioned
- Optimistic Concurrency

Retrieving Data

- Syntax Varies
 - No set-based query language
 - Procedural program languages such as Java, C, etc.
- Application specifies retrieval path
- No query optimizer
- Quick answer is important
- May not be a single “right” answer

Adoption of NoSQL Database

- Couchbase survey was conducted in the 2012.



NoSQL Summary

- NoSQL databases reject:
 - Overhead of ACID transactions
 - “Complexity” of SQL
 - Burden of up-front schema design
 - Declarative query expression
- Programmer responsible for
 - Step-by-step procedural language
 - Navigating access path

Databases Landscape with respect to Availability, Consistency, Partition Tolerance

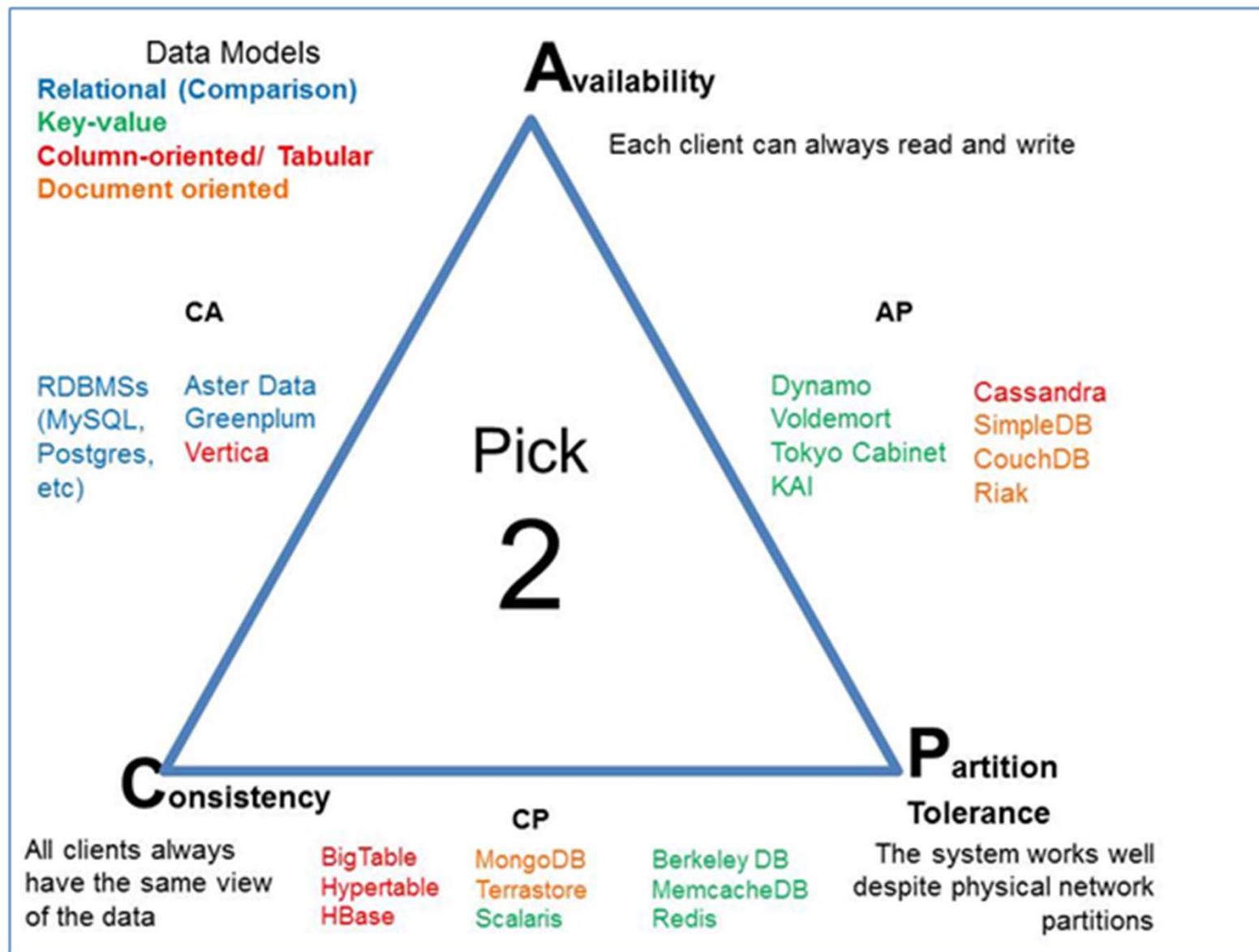


Image Source : <http://blog.flux7.com/blogs/nosql/cap-theorem-why-does-it-matter>

Database Ranking

283 systems in ranking, November 2015

Rank			DBMS	Database Model	Score		
Nov 2015	Oct 2015	Nov 2014			Nov 2015	Oct 2015	Nov 2014
1.	1.	1.	Oracle	Relational DBMS	1480.95	+13.99	+28.82
2.	2.	2.	MySQL	Relational DBMS	1286.84	+7.88	+7.77
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1122.33	-0.90	-97.87
4.	4.	↑ 5.	MongoDB +	Document store	304.61	+11.34	+59.87
5.	5.	↓ 4.	PostgreSQL	Relational DBMS	285.69	+3.56	+28.33
6.	6.	6.	DB2	Relational DBMS	202.52	-4.28	-3.71
7.	7.	7.	Microsoft Access	Relational DBMS	140.96	-0.87	+2.12
8.	8.	↑ 9.	Cassandra +	Wide column store	132.92	+3.91	+40.93
9.	9.	↓ 8.	SQLite	Relational DBMS	103.45	+0.78	+8.17
10.	10.	↑ 11.	Redis +	Key-value store	102.41	+3.61	+20.06

Image captured at : <http://db-engines.com/en/ranking>

NOTE: Most of the major Relational DB Vendors have included NoSQL components to their solutions to stay ahead of the competition.

Web References

- “NoSQL -- Your Ultimate Guide to the Non - Relational Universe!”
<http://nosql-database.org/links.html>
- “NoSQL (RDBMS)”
<http://en.wikipedia.org/wiki/NoSQL>
- PODC Keynote, July 19, 2000. *Towards Robust. Distributed Systems.* Dr. Eric A. Brewer. Professor, UC Berkeley. Co-Founder & Chief Scientist, Inktomi .
www.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
- “Brewer's CAP Theorem” posted by Julian Browne, January 11, 2009.
<http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>

Web References

- “Exploring CouchDB: A document-oriented database for Web applications”, Joe Lennon, Software developer, Core International.
<http://www.ibm.com/developerworksopensource/library/os-couchdb/index.html>
- “Graph Databases, NOSQL and Neo4j” Posted by Peter Neubauer on May 12, 2010 at:
<http://www.infoq.com/articles/graph-nosql-neo4j>
- “Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase comparison”, Kristóf Kovács.
<http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis>
- “Distinguishing Two Major Types of Column-Stores” Posted by Daniel Abadi on March 29, 2010
http://dbmsmusings.blogspot.com/2010/03/distinguishing-two-major-types-of_29.html

Web References

- “MapReduce: Simplified Data Processing on Large Clusters”, Jeffrey Dean and Sanjay Ghemawat, December 2004.
<http://labs.google.com/papers/mapreduce.html>
- “Scalable SQL”, ACM Queue, Michael Rys, April 19, 2011
<http://queue.acm.org/detail.cfm?id=1971597>
- “a practical guide to noSQL”, Posted by Denise Miura on March 17, 2011 at
<http://blogs.marklogic.com/2011/03/17/a-practical-guide-to-nosql/>

Books

- “CouchDB *The Definitive Guide*”, J. Chris Anderson, Jan Lehnardt and Noah Slater. O'Reilly Media Inc., Sebastopol, CA, USA. 2010
- “Hadoop *The Definitive Guide*”, Tom White. O'Reilly Media Inc., Sebastopol, CA, USA. 2011
- “MongoDB *The Definitive Guide*”, Kristina Chodorow and Michael Dirolf. O'Reilly Media Inc., Sebastopol, CA, USA. 2010