Unit 1

Relational Database Modeling

Syllabus (1/2)

- 1.1 Relational model basics
  - 1.1.1 Attributes
  - 1.1.2 Domains
  - 1.1.3 Schemas
  - 1.1.4 Keys
  - 1.1.5 Tuples
- 1.2 Relational algebra
  - 1.2.1 Set operations on relations
  - 1.2.2 Combining operations to form queries
  - 1.2.3 Naming and renaming

Syllabus (2/2)

1.3 Relational database design
- 1.3.1 Functional dependencies
- 1.3.2 First normal form
- 1.3.3 Second normal form
- 1.3.4 Third normal form
- 1.3.5 Other normal forms

1.4 SQL basics
- 1.4.1 Defining a relation schema
- 1.4.2 Database modifications
- 1.4.3 Simple queries
- 1.4.4 Subqueries
- 1.4.5 Aggregation operators
- 1.4.6 Grouping
- 1.4.7 Having clauses
- 1.4.8 Transactions

1.1 Relational model basics

- Dr. E. F. Codd proposed the **relational model** for database systems in 1970.
  - It is the basis for the relational database management system (RDBMS).
  - The relational model consists of the following:
    - Collection of objects or relations
    - Set of operators to act on the relations
    - Data integrity for accuracy and consistency
- The relational model uses a collection of tables to represent both data and the relationships among those data.
- Each table has multiple columns, and each column has a unique name.
  - Tables are also known as relations.

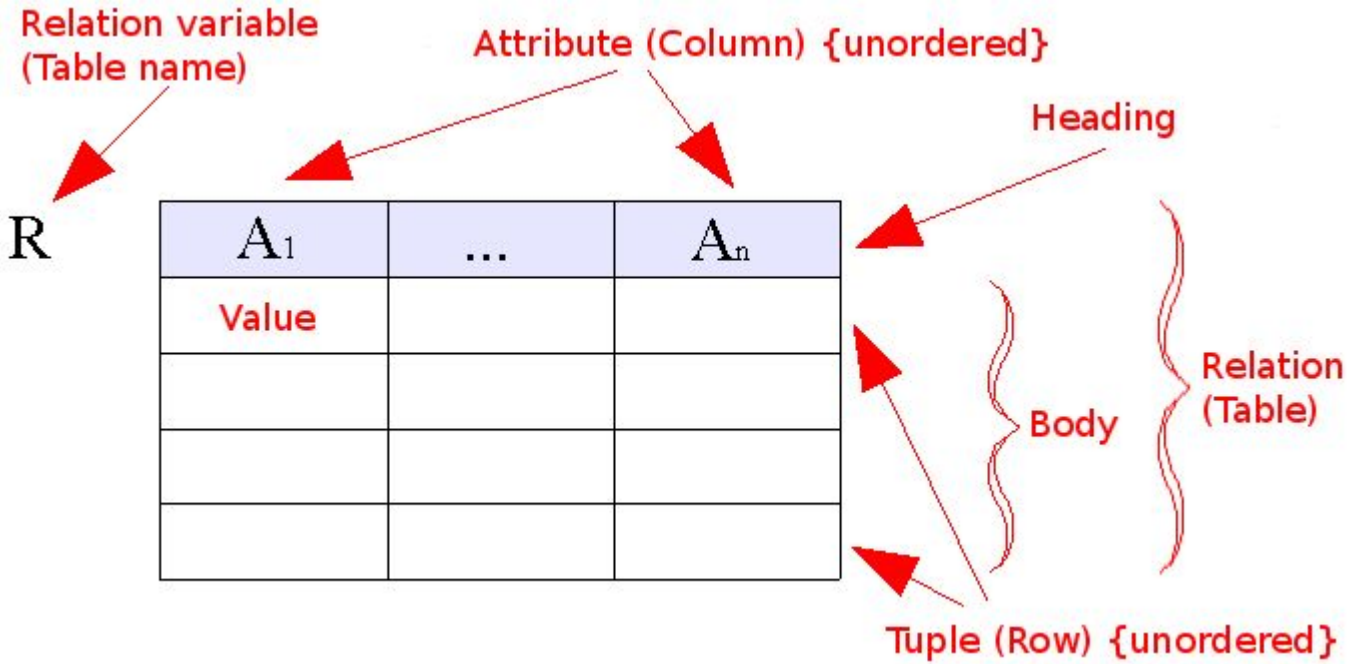
1.1 Relational model basics

- The relational model is an example of a record-based model.
  - Record-based models are so named because the database is structured in fixed-format records of several types.
  - Each table contains records of a particular type.
  - Each record type defines a fixed number of fields, or attributes.
  - The columns of the table correspond to the attributes of the record type.
- The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model

1.1.1. Attributes

- A relation consists of a **heading** and a **body**.
- A heading is a set of attributes.
- An **attribute** is an ordered pair of attribute name and type name.
- An attribute value is a specific valid value for the type of the attribute.
    - This can be either a scalar value or a more complex type.
- In the relational model the term relation is used to refer to a table, while the term tuple is used to refer to a row. Similarly, the term attribute refers to a column of a table.

Structure of a relation



1.1.2 Domains

- For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute.
- We require that, for all relation *r*, the domains of all attributes of *r* be atomic.A domain is atomic if elements of the domain are considered to be indivisible units.
- The important issue is not what the domain itself is, but rather how we use domain elements in our database.
    - Suppose that a phone number attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code and a local number, we would be treating it as a nonatomic value. If we treat each phone number as a single indivisible unit, then the attribute phone number would have an atomic domain.

1.1.3 Schemas

- A **relation schema**, which is the logical design of the database, consists of a list of attributes and their corresponding domains.
- The concept of a **relation instance** corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time; similarly the contents of a relation instance may change with time as the relationis updated. In contrast, the schema of a relation does not generally change.

schema notation

- Consider the *department* relation.

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

- The schema for that relation is

```
department(deptname,building,budget)
```

Standard Data Types

Data type	Description
CHARACTER(n)	Character string. Fixed-length n
VARCHAR(n) or CHARACTER VARYING(n)	Character string. Variable length. Maximum length n
BINARY(n)	Binary string. Fixed-length n
BOOLEAN	Stores TRUE or FALSE values
VARBINARY(n) or BINARY VARYING(n)	Binary string. Variable length. Maximum length n
INTEGER(p)	Integer numerical (no decimal). Precision p
SMALLINT	Integer numerical (no decimal). Precision 5
INTEGER	Integer numerical (no decimal). Precision 10
BIGINT	Integer numerical (no decimal). Precision 19
DECIMAL(p,s)	Exact numerical, precision p, scale s.
NUMERIC(p,s)	Exact numerical, precision p, scale s. (Same as DECIMAL)
FLOAT(p)	Approximate numerical, mantissa precision p. A floating number in base 10 exponential notation.
REAL	Approximate numerical, mantissa precision 7
FLOAT	Approximate numerical, mantissa precision 16
DOUBLE PRECISION	Approximate numerical, mantissa precision 16
DATE	Stores year, month, and day values
TIME	Stores hour, minute, and second values
TIMESTAMP	Stores year, month, day, hour, minute, and second values
INTERVAL	Composed of a number of integer fields, representing a period of time
ARRAY	A set-length and ordered collection of elements
MULTISET	A variable-length and unordered collection of elements
XML	Stores XML data

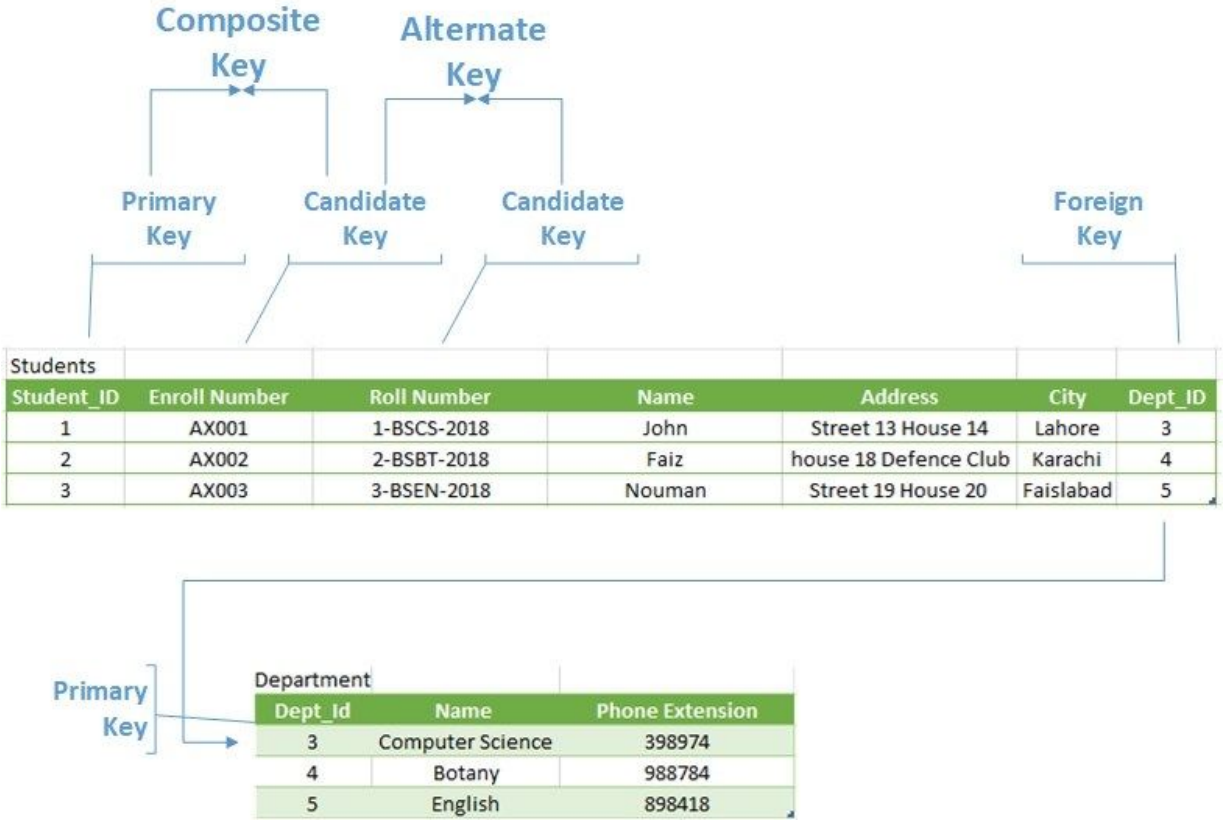
1.1.4 Keys

- In relational model, no two tuples in a relation are allowed to have exactly the same value for all attributes.
- Formally, let *R* denote the set of attributes in the schema of relation *r*.
- A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation
  - We say that a subset *K* of *R* is a superkey for *r*. If *K* is a superkey, then so is any superset of *K*
- Minimal superkeys are called **candidate keys**
  - It is possible that several distinct sets of attributes could serve as a candidate key.
- We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation.

Foreign Key and Referential Integrity

- A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time.
- A relation, say r_1, may include among its attributes the primary key of another relation, say r_2. This attribute is called a **foreign key** from r_1, referencing r_2.
- A **referential integrity constraint** requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation

Keys in relations



Students

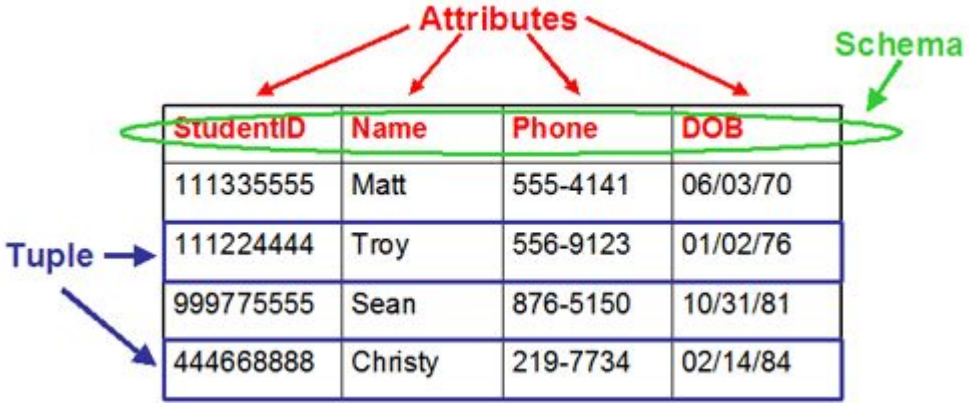
Student_ID	Enroll Number	Roll Number	Name	Address	City	Dept_ID
1	AX001	1-BSCS-2018	John	Street 13 House 14	Lahore	3
2	AX002	2-BSBT-2018	Faiz	house 18 Defence Club	Karachi	4
3	AX003	3-BSEN-2018	Nouman	Street 19 House 20	Faislabad	5

Department

Dept_Id	Name	Phone Extension
3	Computer Science	398974
4	Botany	988784
5	English	898418

1.1.5 Tuples

- In general, a row in a table represents a relationship among a set of values.
- In mathematical terminology, a **tuple** is simply a sequence (or list) of values.
- A relationship between *n* values is represented mathematically by an *n-tuple* of values, i.e., a tuple with *n* values, which corresponds to a row in a table.



1.2 Relational algebra

- A **query language** is a language in which a user requests information from the database.
- Query languages can be categorized as either **procedural** or **non procedural**.
  - In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result.
  - In a non procedural language, the user describes the desired information without giving a specific procedure for obtaining that information.
- The **relational algebra** is procedural, whereas the tuple relational calculus and domain relational calculus are non procedural.
- The relational algebra consists of a set of operations that take one or two relations as input and produce a new relation as their result.

Relational Operations

- Basic relational operations are:
  - Unary
    - selection σ (sigma)
    - projection π (pi)
  - Unary extended
    - Rename ρ (rho)
    - Duplicate elimination δ (delta)
    - Ordering τ (tau)
    - Aggregation γ (gamma)
  - Binary (set theory)
    - Union ∪
    - Intersection ∩
    - Difference -
  - Binary
    - Cartesian product ×
    - Join ⋈
    - Natural join *
    - Outer join
      - Right ⟖
      - Left ⟕
      - Full ⟗
    - Division ÷

Relational unary operations (1/2)

department

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

$\sigma_{budget >= 90000}$ (department)

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

$\pi_{dept\_name, budget}$ (department)

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

Relational unary operations (2/2)

$\rho_{\text{dept-name->office}}$ (department)

office	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Packard	50000
Music	Packard	80000
Physics	Watson	70000

$\tau_{\text{building, budget}\downarrow}$ (department)

office	*building*	*budget*
Music	Packard	80000
History	Packard	50000
Finance	Painter	120000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Biology	Watson	90000
Physics	Watson	70000

$\delta_{\text{building}}(\pi_{\text{building}}$ (department))

building
Watson
Taylor
Painter
Packard
Watson

$_{\text{building}}\gamma_{\text{average (budget)}}$ (department

building	*average(budget)*
Packard	65000
Painter	120000
Taylor	92500
Watson	80000

1.2.1 Set operations on relations

department

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

instructor

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

Π_{dept_name} (department)
$\cup$
Π_{dept_name} (instructor)

dept_name
Biology
Comp. Sci.
Elec. Eng.
Finance
History
Music
Physics

Π_{dept_name} (department)
$\cap$
Π_{dept_name} (instructor)

dept_name
Comp. Sci.
Elec. Eng.
Finance
History
Physics

Π_{dept_name} (department)
-
Π_{dept_name} (instructor)

dept_name
Biology
Music

1.2.2 Combining operations to form queries

$\Pi_{\text{dept\_name, building}}(\text{department}) \times \Pi_{\text{dept\_name, salary}}(\text{instructor})$

dept_name	*building*	*dept_name*	*salary*
Biology	Watson	Physics	95000
Biology	Watson	Finance	90000
Biology	Watson	History	60000
Biology	Watson	Comp. Sci.	75000
Biology	Watson	Elec. Eng.	80000
Biology	Watson	Comp. Sci.	65000
Biology	Watson	History	62000
Biology	Watson	Comp. Sci.	92000
Biology	Watson	Physics	87000
Biology	Watson	Finance	80000
Comp. Sci.	Taylor	Physics	95000
Comp. Sci.	Taylor	Finance	90000
...	...	...	...

instructor.dept_name

$\Pi_{\text{dept\_name, building}}(\text{department}) \bowtie_{\text{department.dept\_name} =}$

$\Pi_{\text{dept\_name, salary}}(\text{instructor})$

dept_name	*building*	*dept_name*	*salary*
Comp. Sci.	Taylor	Comp. Sci.	75000
Comp. Sci.	Taylor	Comp. Sci.	65000
Comp. Sci.	Taylor	Comp. Sci.	92000
Elec. Eng.	Taylor	Elec. Eng.	80000
Finance	Painter	Finance	90000
History	Packard	History	60000
History	Packard	History	62000
Physics	Watson	Physics	87000
Physics	Watson	Physics	95000

Relational binary operations (natural and outer joins)

department * instructor

dept_name	*building*	*budget*	*ID*	*name*	*salary*
Comp. Sci.	Taylor	100000	45565	Katz	75000
Comp. Sci.	Taylor	100000	10101	Srinivasan	65000
Comp. Sci.	Taylor	100000	83821	Brandt	92000
Elec. Eng.	Taylor	85000	98345	Kim	80000
Finance	Painter	120000	12121	Wu	90000
History	Packard	50000	32343	El Said	60000
History	Packard	50000	58583	Califieri	62000
Physics	Watson	70000	33456	Gold	87000
Physics	Watson	70000	22222	Einstein	95000

dept_name	*building*	*dept_name*	*salary*
Biology	Watson	null	null
Comp. Sci.	Taylor	Comp. Sci.	75000
Comp. Sci.	Taylor	Comp. Sci.	65000
Comp. Sci.	Taylor	Comp. Sci.	92000
Elec. Eng.	Taylor	Elec. Eng.	80000
Finance	Painter	Finance	90000
History	Packard	History	60000
History	Packard	History	62000
Music	Packard	null	null
Physics	Watson	Physics	87000
Physics	Watson	Physics	95000

$$\Pi_{\text{dept\_name, building}}(\text{department}) \bowtie_{\text{department.dept\_name = instructor.dept\_name}} \Pi_{\text{dept\_name, salary}}(\text{instructor})$$

1.2.3 Naming and renaming

- Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them.
  - Assume that a relational-algebra expression *E* has arity *n*.
  - Then, the expression $x(A_1,A_2,...,A_n)(E)$ returns the result of expression *E* under the name *x*, and with the attributes renamed to $A_1,A_2,...,A_n$

$$\pi_{instructor.salary}(\sigma_{instructor.salary<d.salary}(instructor \times \rho_d(instructor)))$$

1.3 Relational database design

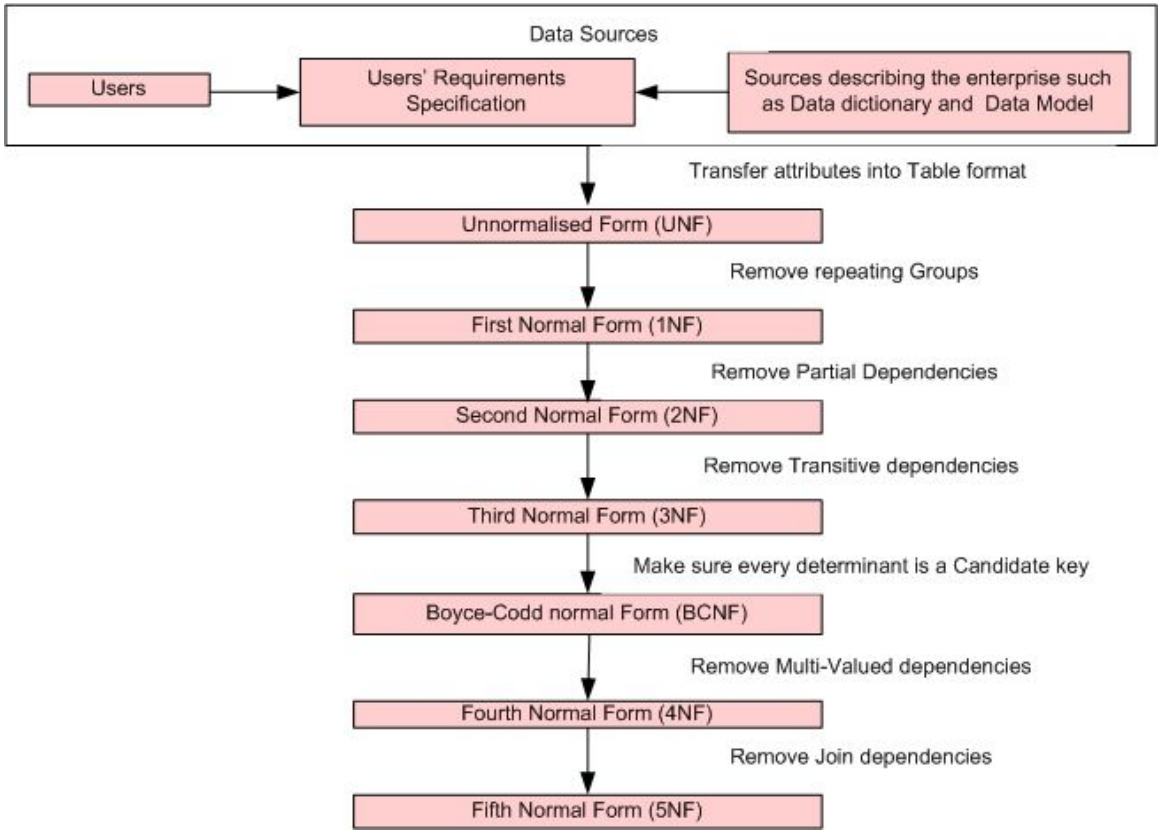
- In general, the goal of relational database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily.
- A real-world database has a large number of schemas and an even larger number of attributes.
  - The number of tuples can be in the millions or higher.
  - Discovering repetition would be costly.

1.3.1 Functional dependencies

- A method for designing a relational database is to use a process commonly known as **normalization**. The approach is to design schemas that are in an appropriate **normal form**.
- In a specification of functional requirements, users describe the kinds of operations (or transactions) that will be performed on the data.
- Therefore, we need to allow the database designer to specify rules such as “each specific value for deptname corresponds to at most one budget”
- This rules are specified as **functional dependencies**

  deptname→budget

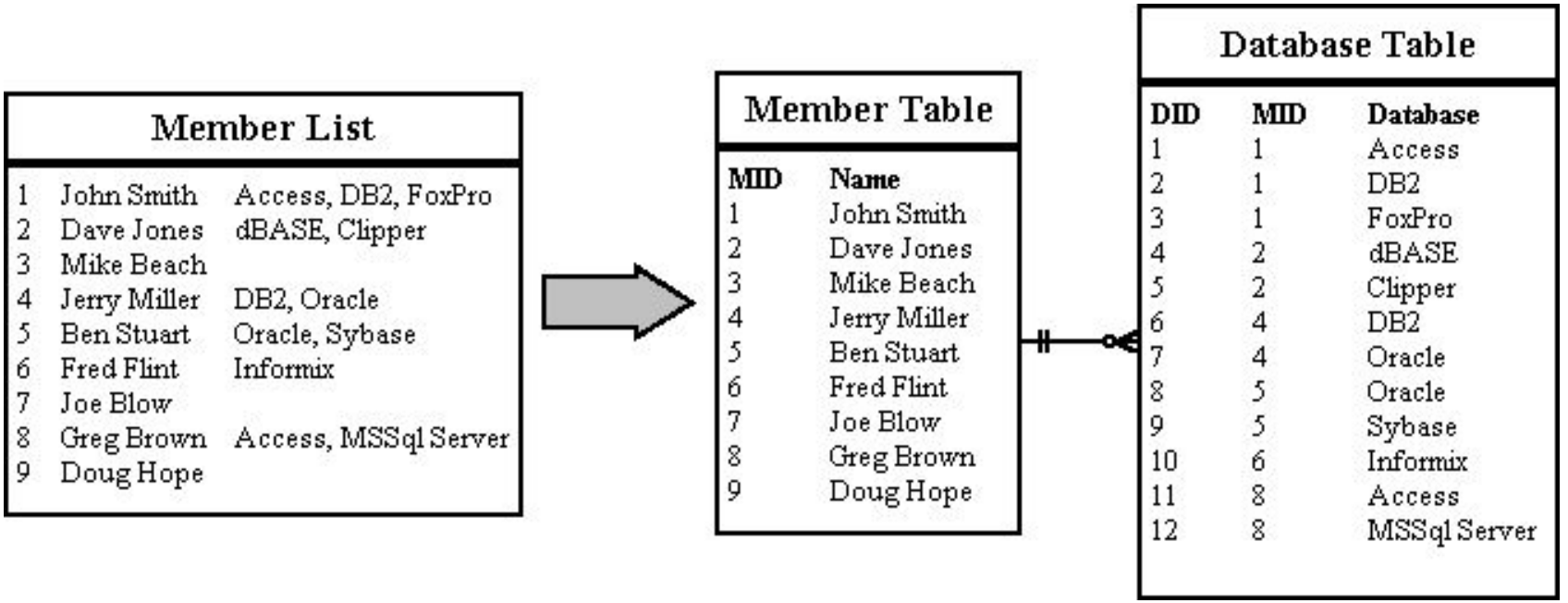
Normalization process



1.3.2 First normal form

- In the relational model, we formalize the idea that attributes do not have any substructure. A domain is **atomic** if elements of the domain are considered to be indivisible units.
- We say that a relation schema *R* is in **first normal form** (1NF) if the domains of all attributes of *R* are atomic.
- The use of set-valued attributes can lead to designs with redundant storage of data, which in turn can result in inconsistencies.

Example



Member List

1	John Smith	Access, DB2, FoxPro
2	Dave Jones	dBASE, Clipper
3	Mike Beach	
4	Jerry Miller	DB2, Oracle
5	Ben Stuart	Oracle, Sybase
6	Fred Flint	Informix
7	Joe Blow	
8	Greg Brown	Access, MSSql Server
9	Doug Hope	

Member Table

MID	Name
1	John Smith
2	Dave Jones
3	Mike Beach
4	Jerry Miller
5	Ben Stuart
6	Fred Flint
7	Joe Blow
8	Greg Brown
9	Doug Hope

Database Table

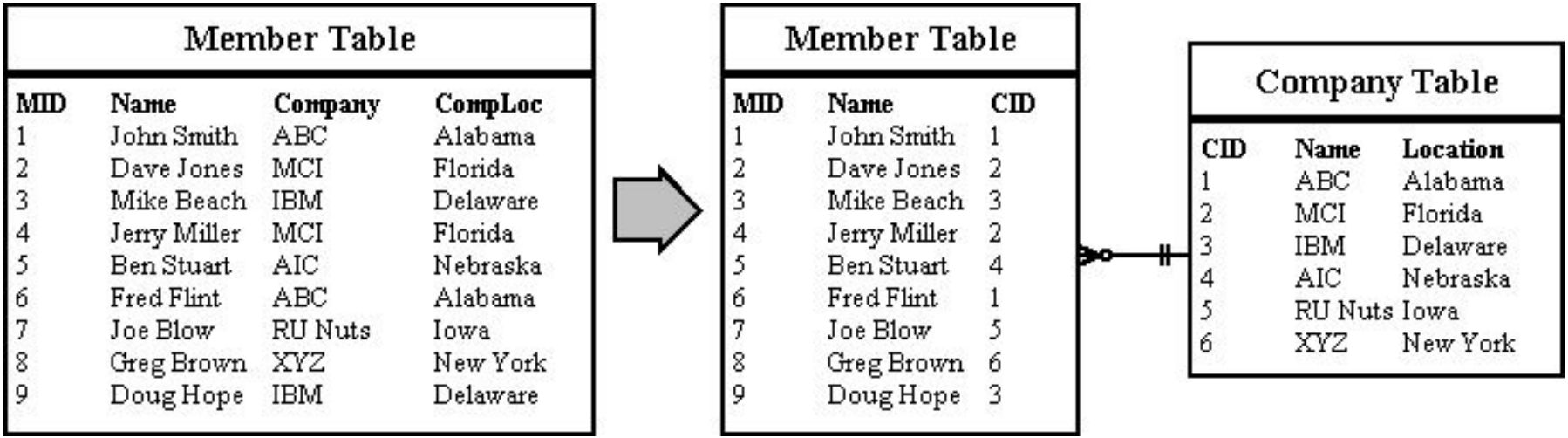
DID	MID	Database
1	1	Access
2	1	DB2
3	1	FoxPro
4	2	dBASE
5	2	Clipper
6	4	DB2
7	4	Oracle
8	5	Oracle
9	5	Sybase
10	6	Informix
11	8	Access
12	8	MSSql Server

1.3.3 Second normal form

- Some of the most commonly used types of real-world constraints can be represented formally as keys (superkeys, candidate keys and primary keys), or as functional dependencies.
- Using the functional-dependency notation, we say that K is a superkey of r(R) if the functional dependency K→R holds on r(R).
- Functional dependencies allow us to express constraints that we cannot express with superkeys. For example, consider the schema:

  instdept(ID,name,salary,deptname,building,budget)
- in which the functional dependency *deptname→budget* holds because for each *department* (identified by *deptname*) there is a unique *budget* amount.

Example



Member Table

MID	Name	Company	CompLoc
1	John Smith	ABC	Alabama
2	Dave Jones	MCI	Florida
3	Mike Beach	IBM	Delaware
4	Jerry Miller	MCI	Florida
5	Ben Stuart	AIC	Nebraska
6	Fred Flint	ABC	Alabama
7	Joe Blow	RU Nuts	Iowa
8	Greg Brown	XYZ	New York
9	Doug Hope	IBM	Delaware

Member Table

MID	Name	CID
1	John Smith	1
2	Dave Jones	2
3	Mike Beach	3
4	Jerry Miller	2
5	Ben Stuart	4
6	Fred Flint	1
7	Joe Blow	5
8	Greg Brown	6
9	Doug Hope	3

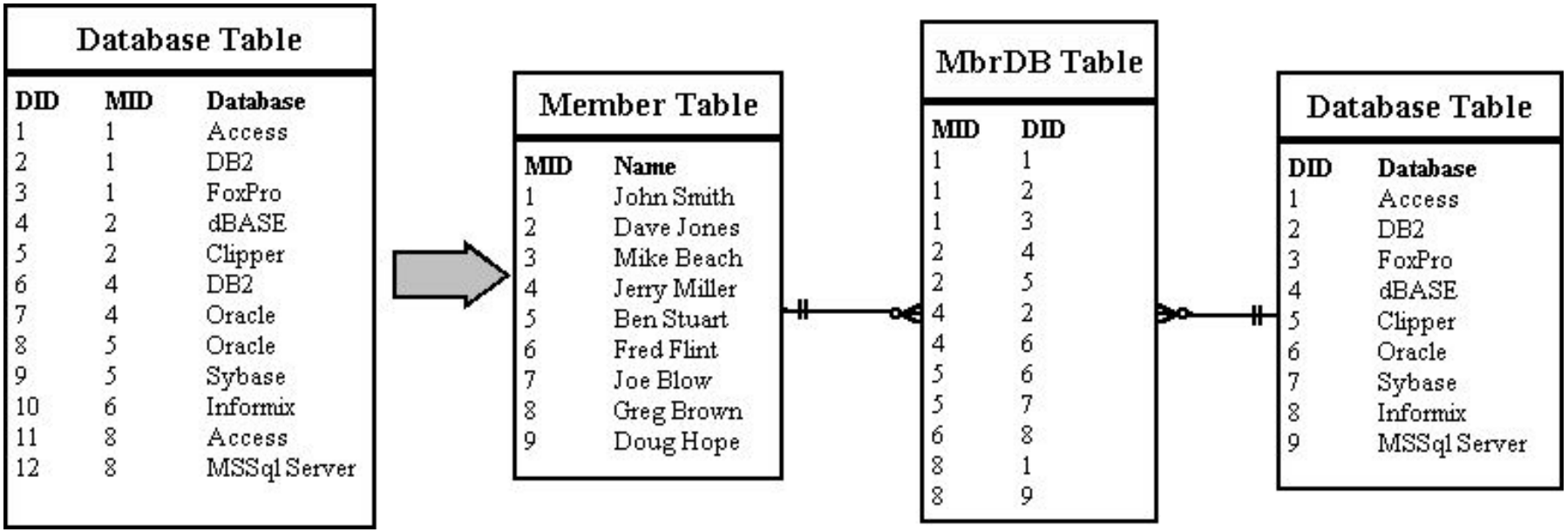
Company Table

CID	Name	Location
1	ABC	Alabama
2	MCI	Florida
3	IBM	Delaware
4	AIC	Nebraska
5	RU Nuts	Iowa
6	XYZ	New York

1.3.4 Third normal form

- A relation schema R is in third normal form with respect to a set F of functional dependencies if, for all functional dependencies of the form $x \rightarrow y$, where $x \subseteq R$ and $y \subseteq R$, at least one of the following holds:
  - $x \rightarrow y$ is a trivial functional dependency.
  - x is a superkey for R.
  - Each attribute A in $y - x$ is contained in a candidate key for R.

Example


Database Table
DID MID Database
1 1 Access
2 1 DB2
3 1 FoxPro
4 2 dBASE
5 2 Clipper
6 4 DB2
7 4 Oracle
8 5 Oracle
9 5 Sybase
10 6 Informix
11 8 Access
12 8 MSSql Server
Member Table
MID Name
1 John Smith
2 Dave Jones
3 Mike Beach
4 Jerry Miller
5 Ben Stuart
6 Fred Flint
7 Joe Blow
8 Greg Brown
9 Doug Hope
MbrDB Table
MID DID
1 1
1 2
1 3
2 4
2 5
4 2
4 6
5 6
5 7
6 8
8 1
8 9
Database Table
DID Database
1 Access
2 DB2
3 FoxPro
4 dBASE
5 Clipper
6 Oracle
7 Sybase
8 Informix
9 MSSql Server

1.3.5 Other normal forms

- One of the more desirable normal forms that we can obtain is Boyce–Codd normal form (BCNF). It eliminates all redundancy that can be discovered based on functional dependencies.
- A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies of the form $x \rightarrow y$, where $x \subseteq R$ and $y \subseteq R$, at least one of the following holds:
  - $x \rightarrow y$ is a trivial functional dependency (that is,$x \subseteq y$).
  - x is a superkey for schema R

1.4 SQL basics

- IBM developed the original version of SQL, originally called Sequel, as part of the System R project in the early 1970s.
- The Sequel language has evolved since then, and its name has changed to SQL (Structured Query Language).
- In 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) published an SQL standard, called SQL-86.
- ANSI published an extended standard for SQL, SQL-89, in 1989. The next version of the standard was SQL-92 standard, followed by SQL:1999, SQL:2003, SQL:2006, and most recently SQL:2008.

SQL issues

- The SQL language has several parts:
  - **Data-definition language(DDL).** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
  - **Data-manipulation language(DML).** The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
  - **Integrity**.The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
  - **View definition**.The SQL DDL includes commands for defining views.
  - **Transaction control**. SQL includes commands for specifying the beginning and ending of transactions.
  - **Authorization**.The SQL DDL includes commands for specifying access rights to relations and views.

1.4.1 Defining a relation schema

- The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:
  - The schema for each relation.
  - The types of values associated with each attribute.
  - The integrity constraints.
  - The set of indices to be maintained for each relation.
  - The security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

Create table DDL

- SQL define a relation by using the create table command:
  - **CREATE TABLE** [schema.]table
    (col_name **datatype** [**DEFAULT** expr][**column_constraint**],
    ...
    [**table_constraint**][,...]);
  - **column_constraint** -> **NOT NULL** | [**CONSTRAINT** name] **UNIQUE** | **PRIMARY KEY** | **CHECK** (condition) | **REFERENCES** table_ref[(col_ref)] [**ON** {**DELETE** | **UPDATE**} {**CASCADE** | **SET NULL** | **NO ACTION** | **SET DEFAULT**}]
  - **table_constraint** -> [**CONSTRAINT** name] **UNIQUE** (col_name[, col_name...]) | **PRIMARY KEY** (col_name[, col_name...]) | **CHECK** (condition) | **FOREIGN KEY** (col_name[, col_name...]) **REFERENCES** table_ref[(col_ref[, col_ref...])] [**ON** {**DELETE** | **UPDATE**} {**CASCADE** | **SET NULL** | **NO ACTION** | **SET DEFAULT**}]

Basic Types

- The SQL standard supports a variety of built-in types, including:
  - **char[*acter*](n):** A fixed-length character string with user-specified length n.
  - **varchar(n):** A variable-length character string with user-specified maximum length n.
  - **int[*eger*]:** An integer (a finite subset of the integers that is machine dependent).
  - **smallint**: A small integer (a machine-dependent subset of the integer type).
  - **numeric (*p,d)*:** A fixed-point number with user-specified precision. The number consists of *p* digits (plus a sign), and *d* of the *p* digits are to the right of the decimal point.
  - **real, double precision:** Floating-point and double-precision floating-point numbers with machine-dependent precision.
  - **float(n):** A floating-point number, with precision of at least *n* digits.

Constraints

- **DEFAULT**. Specify a default value for a column during an insert.
- **NOT NULL**. Ensures that null values are not permitted for the column.
- **UNIQUE**. Requires that every value in a column or set of columns (key) be unique.
- **PRIMARY KEY**. Creates a primary key for the table. Only one primary key can be created for each table.
- **CHECK**. Defines a condition that each row must satisfy
- **FOREIGN KEY**. Designates a column or combination of columns as a foreign key and establishes a relationship between a primary key or a unique key in the same table or a different table.
    - **ON DELETE | UPDATE CASCADE**: Deletes or updates the dependent rows in the child table when a row in the parent table is deleted
    - **ON DELETE | UPDATE SET NULL**: Converts dependent foreign key values to null
    - **ON DELETE | UPDATE SET DEFAULT**: Converts dependent foreign key values to default value on column
    - The default behavior is called the *restrict rule*, which disallows the update or deletion of referenced data.

Example

```
create table department (
deptname varchar(20),
building varchar(15),
budget numeric(12,2)
constraint chk_budg
check(budget > 0.0),
primary key(deptname)
);
```

```
create table instructor(
ID varchar(5),
name varchar(20) not null,
deptname varchar(20),
salary numeric(8,2) default
100.00,
primary key(ID),
unique(name),
foreign key(deptname) references
department on delete no action
on update cascade
);
```

1.4.2 Database modifications

- If necessary change the table structure for any of the following reasons:
  - Omitted a column.
  - Column definition needs to be changed.
  - Need to remove column
- Using the ALTER TABLE statement:
  - ```
    ALTER TABLE [schema.]table
    [ADD col_name col_constraint]
    [MODIFY col_name type col_constraint]
    [ADD table_constraint]
    [DROP PRIMARY KEY | UNIQUE | CONSTRAINT constraint_name [CASCADE]]
    ```
    ```

Drop a table

- When you dropping a table
  - All data and structure in the table are deleted.
  - Any pending transactions are committed.
  - All indexes are dropped.
  - All constraints are dropped

```
DROP TABLE table_name
```

Data Manipulation Language operations

- A DML statement is executed when:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- INSERT Statement Syntax

```
INSERT INTO table [(column [, column...])]
VALUES (value [, value...]);
```

- Insert a new row containing values for each column.
  - List values in the default order of the columns in the table.
  - Optionally, list the columns in the INSERT clause.
  - Enclose character and date values in single quotation marks

Example

```
INSERT INTO departments(department_id,
department_name, manager_id, location_id)
VALUES (70, 'Public Relations', 100, 1700);
```

- Inserting Rows with Null Value

```
INSERT INTO departments (department_id, department_name )
VALUES (30, 'Purchasing');

INSERT INTO departments
VALUES (100, 'Finance', NULL, NULL);
```

Changing Data in a Table

- Modify existing rows with the UPDATE statement:

```
UPDATE table
SET column = value [, column = value, ...]
[WHERE condition];
```

- Example:

```
UPDATE employees
SET department_id = 70
WHERE employee_id = 113;
```

Removing a Row from a Table

- Remove existing rows from a table by using the DELETE statement:

```
DELETE [FROM] table
[WHERE condition];
```

- Example

```
DELETE FROM departments
WHERE department_name = 'Finance';
```

- TRUNCATE Statement
- Removes all rows from a table, leaving the table empty and the table structure intact

```
TRUNCATE TABLE table_name;
```

1.4.3 Simple queries

- Basic SELECT Statement

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table;
```

- SELECT identifies the columns to be displayed.
- FROM identifies the table containing those columns
- Example:

```
SELECT department_id, location_id
FROM departments;
```

Writing SQL Statements

- SQL statements are not case sensitive.
- SQL statements can be on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines
- Indents are used to enhance readability.
- Semicolons (;) are required if you execute multiple SQL statements

Arithmetic Expressions

- Create expressions with number and date data by using arithmetic operators.
  - `*` Multiply
  - `/` Divide
  - `-` Subtract
  - `+` Add
- Example:

```
SELECT last_name, salary, salary + 300
FROM employees;
```

Defining a Column Alias

- A column alias:
  - Renames a column heading
  - Is useful with calculations
  - Immediately follows the column name (There can also be the optional AS keyword between the column name and alias.)
  - Requires double quotation marks if it contains spaces or special characters or if it is case sensitive
- Example:

```
SELECT last_name "Name" , salary*12 "Annual Salary"
FROM employees;
SELECT last_name AS name, commission_pct comm
FROM employees;
```

Duplicate Rows

- Use keyword **DISTINCT** to avoid duplicate rows:

```
SELECT DISTINCT department_id
FROM employees;
```

Limiting the Rows That Are Selected

- Restrict the rows that are returned by using the WHERE clause:
  - The WHERE clause follows the FROM clause.

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

- Example:

```
SELECT employee_id, last_name, job_id, department_id
FROM employees
WHERE department_id = 90 ;
```

Comparison Conditions

<	Less than
<=	Less than or equal to
>=	Greater than or equal to
>	Greater than
=	Equal to
<>	Not equal to
BETWEEN ...AND…	Between two values (inclusive)
IN(set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

Logical Conditions

NOT	Returns TRUE if the following condition is false
OR	Returns TRUE if either component condition is true
AND	Returns TRUE if both component conditions are true

Example:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary >=10000
AND job_id LIKE '%MAN%' ;
```

Using the ORDER BY Clause

- Sort retrieved rows with the ORDER BY clause:
  - ASC: ascending order, default
  - DESC: descending order
- The ORDER BY clause comes last in the SELECT statement
- Example:

```
SELECT last_name, job_id, department_id, hire_date
FROM employees
ORDER BY hire_date ;
SELECT last_name, department_id, salary
FROM employees
ORDER BY department_id, salary DESC;
```

1.4.4 Subqueries

- The subquery (inner query) executes once before the main query (outer query).
  - The result of the subquery is used by the main query.
- Syntax:

```
SELECT select_list
FROM table
WHERE expr operator
       (SELECT select_list
          FROM table);
```

Example

```
SELECT last_name, salary
FROM employees
WHERE salary >
       (SELECT salary
       FROM employees
       WHERE last_name = 'Abel');
```

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition.
- Use single-row operators with single-row subqueries, and use multiple-row operators with multiple-row subqueries.

1.4.5 Aggregation operators

- Functions that give results over some column
  - AVG
  - COUNT
  - MAX
  - MIN
  - SUM
- Functions AVG and SUM work only over numeric data
- Example:

```
SELECT AVG(salary), MAX(salary),
MIN(salary), SUM(salary)
FROM employees
WHERE job_id LIKE '%REP%';
```

Functions

- MIN and MAX work for numeric, character, and date data types

```
SELECT MIN(hire_date), MAX(hire_date)
FROM employees;
```

- COUNT(DISTINCT *expr*) returns the number of distinct non-null values of the *expr*

```
SELECT COUNT(DISTINCT department_id)
FROM employees;
```

1.4.6 Grouping

- Permits divide rows in a table into smaller groups by using the GROUP BY clause

```
SELECT column, group_function(column)
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[ORDER BY column];
```

- All columns in the SELECT list that are not in group functions must be in the GROUP BY clause
- The GROUP BY column does not have to be in the SELECT list

Example

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id ;
```

- Using the GROUP BY Clause on Multiple Columns

```
SELECT department_id dept_id, job_id, SUM(salary)
FROM employees
GROUP BY department_id, job_id;
```

1.4.7 Having clause

- Restrict Group Results with the HAVING Clause
  - 1. Rows are grouped.
    2. The group function is applied.
    3. Groups matching the HAVING clause are displayed
- Syntax:

```
SELECT column, group_function
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[HAVING group_condition]
[ORDER BY column]
```

Example

```
SELECT department_id, MAX(salary)
FROM employees
GROUP BY department_id
HAVING MAX(salary)>10000 ;

SELECT job_id, SUM(salary) PAYROLL
FROM employees
WHERE job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING SUM(salary) > 13000
ORDER BY SUM(salary);
```

1.4.8 Transactions

- A transaction consists of a sequence of query and/or update statements.
- A database transaction consists of one of the following:
  - DML statements that constitute one consistent change to the data
  - One DDL statement
  - One data control language (DCL) statement
- Begin when the first DML SQL statement is executed
- End with one of the following events:
  - A COMMIT or ROLLBACK statement is issued.
  - A DDL or DCL statement executes (automatic commit)

Commit and Rollback operations

- With the use of COMMIT and ROLLBACK statements, is possible:
  - Ensure data consistency
  - Preview data changes before making changes permanent
  - Group logically related operations
- Commit the changes:

```
DELETE FROM employees WHERE employee_id = 99999;
1 row deleted.
INSERT INTO departments VALUES (290, 'Corporate Tax',
NULL, 1700);
1 row created
COMMIT;
Commit complete
```

Rollback operation

- Discard all pending changes by using the ROLLBACK statement:
  - Data changes are undone.
  - Previous state of the data is restored.
  - Locks on the affected rows are released

```
DELETE FROM copy_emp;
20 rows deleted.
ROLLBACK ;
Rollback complete
```

Properties of a transaction

- A database transaction, by definition, must be **atomic**, **consistent**, **isolated** and **durable**. Database practitioners often refer to these properties of database transactions using the acronym ACID.
- Transactions provide an "all-or-nothing" proposition, stating that each work-unit performed in a database must either complete in its entirety or have no effect whatsoever. Further, the system must isolate each transaction from other transactions, results must conform to existing constraints in the database, and transactions that complete successfully must get written to durable storage.

Isolation in a DBMS

- Isolation determines how transaction integrity is visible to other users and systems.
- When attempting to maintain the highest level of isolation, a DBMS usually acquires locks on data which may result in a loss of concurrency
    - A lower isolation level increases the ability of many users to access the same data at the same time, but increases the number of concurrency effects (such as dirty reads or lost updates) users might encounter.
    - Conversely, a higher isolation level reduces the types of concurrency effects that users may encounter, but requires more system resources and increases the chances that one transaction will block another

Isolation levels

- The isolation levels defined by the ANSI/ISO SQL standard are:
  - **Serializable**. This is the highest isolation level. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.
  - **Repeatable reads.** Write skew is possible at this isolation level, a phenomenon where two writes are allowed to the same column(s) in a table by two different writers (who have previously read the columns they are updating), resulting in the column having data that is a mix of the two transactions.
  - **Read committed.** Guarantees that any data read is committed at the moment it is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read.
  - **Read uncommitted**. This is the lowest isolation level. In this level, dirty reads are allowed, so one transaction may see not-yet-committed changes made by other transactions.

Read phenomena (1/3)

- The ANSI/ISO standard SQL 92 refers to three different read phenomena:
  - **Dirty reads** (aka uncommitted dependency) occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.

Transaction 1

```
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 20 */



1;

/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 21 */
```

Transaction 2

```
/* Query 2 */
UPDATE users SET age = 21 WHERE id =

/* No commit here */



ROLLBACK; /* lock-based DIRTY READ */
```

Read phenomena (2/3)

- A **non-repeatable read** occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads.

Transaction 1

```
/* Query 1 */
SELECT * FROM users WHERE id = 1;
```

Transaction 2

```
/* Query 2 */
UPDATE users SET age = 21 WHERE id = 1;
COMMIT; /* in multiversion concurrency
control, or lock-based READ COMMITTED */
```

```
/* Query 1 */
SELECT * FROM users WHERE id = 1;
COMMIT; /* lock-based REPEATABLE READ */
```

- At the SERIALIZABLE and REPEATABLE READ isolation levels, the DBMS must return the old value for the second SELECT. At READ COMMITTED and READ UNCOMMITTED, the DBMS may return the updated value; this is a non-repeatable read.

Read phenomena (3/3)

- A phantom read occurs when, in the course of a transaction, new rows are added by another transaction to the records being read.

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;

/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
COMMIT;
```

Transaction 2

```
/* Query 2 */
INSERT INTO users(id,name,age) VALUES
(3,'Bob',27);

COMMIT;
```

- In REPEATABLE READ mode, the range would not be locked, allowing the record to be inserted

Isolation levels vs read phenomena

- The following table shows how a DBMS deals with different read phenomena:

Isolation level	Lost updates	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	don't occur	may occur	may occur	may occur
Read Committed	don't occur	don't occur	may occur	may occur
Repeatable Read	don't occur	don't occur	don't occur	may occur
Serializable	don't occur	don't occur	don't occur	don't occur

Unit 2

Semistructured Data-model Basics

Syllabus

2.1 The semistructured data-model

- 2.1.1 Semistructured data
- 2.1.2 XML
- 2.1.3 Document Type Definitions (DTD)
- 2.1.4 XML schema

2.2 Programming languages for XML

- 2.2.1 XPath
- 2.2.2 XQuery
- 2.2.3 Extensible Stylesheet Language

2.1 The semistructured data-model

- The semi-structured model is a database model where there is no separation between the data and the schema, and the amount of structure used depends on the purpose.
- The advantages of this model are the following:
  - It can represent the information of some data sources that cannot be constrained by schema.
  - It provides a flexible format for data exchange between different types of databases.
  - It can be helpful to view structured data as semi-structured (for browsing purposes).
  - The schema can easily be changed.
  - The data transfer format may be portable.

2.1.1 Semistructured data

- Semi-structured data is a form of structured data that does not conform with the formal structure of data models associated with data tables, but contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data.
- Advantages
  - Support for nested or hierarchical data often simplifies data models representing complex relationships between entities.
  - Support for lists of objects simplifies data models by avoiding messy translations of lists into a relational data model.
- Disadvantages
  - The traditional relational data model has a popular and ready-made query language, SQL.
  - Prone to "garbage in, garbage out"; by removing restraints from the data model, there is less fore-thought that is necessary to operate a data application.

2.1.2 XML

- Is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.
- The design goals of XML emphasize simplicity, generality, and usability across the Internet.
- It is a textual data format with strong support via Unicode for different human languages.

XML constructs (1/3)

- For the family of markup languages that includes HTML,SGML, and XML,the markup takes the form of **tags** enclosed in angle brackets,**<>**.
- Tag are used in pairs, with **<tag>** and **</tag>** delimiting the beginning and the end of the portion of the document to which the tag refers.
- An **element** is a logical document component that either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag.
    - The characters between the start-tag and end-tag, if any, are the element's content, and may contain markup, including other elements, which are called child elements.
    - An example is **<greeting>Hello, world!</greeting>**. Another is **<line-break />.**

XML constructs (2/3)

- An **attribute** is a markup construct consisting of a name–value pair that exists within a start-tag or empty-element tag.
  - An example is
  - An XML attribute can only have a single value and each attribute can appear at most once on each element.
- XML documents may begin with an **XML declaration** that describes some information about themselves.
  - An example is **<?xml version="1.0" encoding="iso-8859-1"?>**.
- Sometimes we need to store values containing tags without having the tags interpreted as XML tags. So that we can do so, XML allows this construct, CDATA section:
  - **<![CDATA[**<course>···</course>**]]>**

XML constructs (3/3)

- **Comments** may appear anywhere in a document outside other markup.
  - Comments begin with **<!--** and end with **-->**.
  - Comments cannot appear before the XML declaration.
- The XML specification defines five "predefined **entities**" representing special characters, and requires that all XML processors translates them. The entities can be explicitly declared in a DTD

`<`	represents	`"<"`
`>`	represents	`">"`
`&`	represents	`"&"`
`&apos;`	represents	`"'"`
`&quot;`	represents	`'"'`

Well-formedness in XML

- The XML specification defines an XML document as a well-formed text, meaning that it satisfies a list of syntax rules provided in the specification. Some key points in the fairly lengthy list include:
  - The document contains only properly encoded legal Unicode characters.
  - None of the special syntax characters such as **<** and **&** appear except when performing their markup-delineation roles.
  - The start-tag, end-tag, and empty-element tag that delimit elements are correctly nested, with none missing and none overlapping.
  - Tag names are case-sensitive; the start-tag and end-tag must match exactly.
  - Tag names cannot contain any of the characters `!"#$%&'()*+,/;<=>?@[\]^`{|}~` , nor a space character, and cannot begin with `"-"`, `"."`, or a numeric digit.
  - A single root element contains all the other elements.

XML Example

```
<bibliography>
    <paper pubid="wsa" role="publication">
        <authors>
            <author authorRef=”joyce” age=“45”>
                J. L. R. Colina </author>
        </authors>
        <fullPaper source="http://mysite.com/confusion"/>
        <title>Object Confusion in a Deviator System </title>
        <related papers="deviation101 x_deviators"/>
    </paper>
</bibliography>
```

Schemas and validation

- In addition to being well-formed, an XML document may be valid. This means that it contains a reference to a Document Type Definition (DTD), and that its elements and attributes are declared in that DTD and follow the grammatical rules for them that the DTD specifies.
  - It defines the document structure with a list of legal elements and attributes.
  - A DTD can be declared inline inside an XML document, or as an external reference.
  - The external subset may be referenced via a **public** identifier and/or a **system** identifier
- Example

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

- However, the DTD does not constraint types like integer or string. Instead, it constrains only the appearance of subelements and attributes within an element.

2.1.3 Document Type Definitions (DTD)

- In a DTD, XML elements are declared with the following syntax:

  `<!ELEMENT element-name (element-content)>`

  - Empty elements are declared with the category keyword **EMPTY**:

    `<!ELEMENT element-name EMPTY>`
  - Elements with only parsed character data are declared with **#PCDATA** inside parentheses:

    `<!ELEMENT element-name (#PCDATA)>`
  - Elements with one or more children are declared with the name of the children elements inside parentheses:

    `<!ELEMENT element-name (child1,child2,...)>`

Element occurrences

- Declaring Only One Occurrence of an Element

  ```
  <!ELEMENT element-name (child-name)>
  ```

- Declaring Minimum One Occurrence of an Element

  ```
  <!ELEMENT element-name (child-name+)>
  ```

- Declaring Zero or More Occurrences of an Element

  ```
  <!ELEMENT element-name (child-name*)>
  ```

- Declaring Zero or One Occurrences of an Element

  ```
  <!ELEMENT element-name (child-name?)>
  ```

Attributes

- An attribute declaration has the following syntax:

```
<!ATTLIST element-name attribute-name attribute-type attribute-value>
```

  - The attribute-type can be one of the following:

CDATA	*The value is character data*
(en1|en2|..)	*The value must be one from an enumerated list*
ID	*The value is a unique id*
IDREF	*The value is the id of another element*
IDREFS	*The value is a list of other ids*
NMTOKEN	*The value is a valid XML name*
NMTOKENS	*The value is a list of valid XML names*
ENTITY	*The value is an entity*
ENTITIES	*The value is a list of entities*
NOTATION	*The value is a name of a notation*

  - The attribute-value can be one of the following:

value	*The default value of the attribute*
#REQUIRED	*The attribute is required*
#IMPLIED	*The attribute is optional*
#FIXED value	*The attribute value is fixed*

DTD Example

```
<!ELEMENT author (#PCDATA)>
<!ATTLIST author
authorRef CDATA #REQUIRED
age CDATA #REQUIRED>
<!ELEMENT authors (author)>
<!ELEMENT fullPaper EMPTY>
<!ATTLIST fullPaper source CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT related EMPTY>
<!ATTLIST related papers CDATA #REQUIRED>
<!ELEMENT paper (authors,fullPaper?,title,related)>
<!ATTLIST paper
pubid CDATA #REQUIRED
role CDATA #REQUIRED>
<!ELEMENT bibliography (paper+)>
```

2.1.4 XML Schema

- An **XML Schema** describes the structure of an XML document.
- The XML Schema language is also referred to as **XML Schema Definition** (XSD).
- The purpose of an XML Schema is to define the legal building blocks of an XML document
  - the elements and attributes that can appear in a document
  - the number of (and order of) child elements
  - data types for elements and attributes
  - default and fixed values for elements and attributes

XML Schema validation

- A schema is an abstract collection of metadata, consisting of a set of schema components: chiefly element and attribute declarations and complex and simple type definitions.
  - When an instance document is validated against a schema (a process known as assessment), the schema to be used for validation can either be supplied as a parameter to the validation engine, or it can be referenced directly from the instance document using attribute **xsi:schemaLocation**

```
<?xml version="1.0"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://www.mysite.com myschema.xsd">
```

XML Schema structure (1/3)

- The main components of a schema are:
  - **Element declarations**, which define properties of elements. These include the element name and target namespace. An important property is the *type* of the element, which constrains what attributes and children the element can have.
  - `<xs:element name="xxx" type="schema-type" default="value" fixed="value"/>`
  - XSD provides a set of 19 primitive data types `(anyURI, base64Binary, boolean, date, dateTime, decimal, double, duration, float, hexBinary, gDay, gMonth, gMonthDay, gYear, gYearMonth, NOTATION, QName, string, time).`
  - **Attribute declarations**, which define properties of attributes. Again the properties include the attribute name and target namespace. The attribute type constrains the values that the attribute may take. An attribute declaration may also include a default value or a fixed value (which is then the only value the attribute may take.)
  - `<xs:attribute name="xxx" type="schema-type" default="value" fixed="value" use="required"/>`

XML Schema structure (2/3)

- **Restrictions** are used to define acceptable values for XML elements or attributes. Restrictions on XML elements are called facets.
  - **enumeration** Defines a list of acceptable values
  - **fractionDigits** Specifies the maximum number of decimal places allowed. Must be >= 0
  - **length** Specifies the exact number of characters or list items allowed. Must be >= 0
  - **maxExclusive** Specifies the upper bounds for numeric values (the value must be < than this value)
  - **maxInclusive** Specifies the upper bounds for numeric values (the value must be <= to this value)
  - **maxLength** Specifies the maximum number of characters or list items allowed. Must be >= 0
  - **minExclusive** Specifies the lower bounds for numeric values (the value must be > this value)
  - **minInclusive** Specifies the lower bounds for numeric values (the value must be >=l to this value)
  - **minLength** Specifies the minimum number of characters or list items allowed. Must be >= 0
  - **pattern** Defines the exact sequence of characters that are acceptable
  - **totalDigits** Specifies the exact number of digits allowed. Must be > 0
  - **whiteSpace** Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

Examples

```
The only acceptable values are: Audi, Golf, BMW
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="Golf"/>
      <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
The only acceptable value is THREE of the
LOWERCASE OR UPPERCASE letters from a to z
<xs:element name="initials">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern
value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```
The acceptable value is zero or more
occurrences of lowercase letters from a to z
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-z])*"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
There must be exactly eight characters in a row
and those characters must be lowercase or
uppercase letters from a to z, or a number from
0 to 9
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-zA-Z0-9]{8}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

XML Schema structure (3/3)

- **Complex types** describe the permitted content of an element, including its element and text children and its attributes. A complex type definition consists of a set of attribute uses and a content model.
  - There are four kinds of complex elements:
    - empty elements
    - elements that contain only other elements
    - elements that contain only text
    - elements that contain both other elements and text
  - Example
  - ```
    <xs:element name="employee">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="firstname" type="xs:string"/>
          <xs:element name="lastname" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    ```
    ```

Indicators

- There are several indicators:
  - Order indicators:
    - `<all>` specifies that the child elements can appear in any order, and that each child element must occur only once
    - `<choice>` specifies that either one child element or another can occur
    - `<sequence>` specifies that the child elements must appear in a specific order
  - Occurrence indicators:
    - `maxOccurs`, specifies the maximum number of times an element can occur
    - `minOccurs`, specifies the minimum number of times an element can occur

```
<xs:element name="persons">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="person"
maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element
name="full_name" type="xs:string"/>
            <xs:element
name="child_name" type="xs:string"
        minOccurs="0" maxOccurs="5"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

XML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
   <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- definition of simple elements -->
<xs:element name="orderperson" type="xs:string"/>
<xs:element name="name" type="xs:string"/>
<xs:element name="address" type="xs:string"/>
<xs:element name="city" type="xs:string"/>
<xs:element name="country" type="xs:string"/>
<xs:element name="title" type="xs:string"/>
<xs:element name="note" type="xs:string"/>
<xs:element name="quantity" type="xs:positiveInteger"/>
<xs:element name="price" type="xs:decimal"/>

<!-- definition of attributes -->
<xs:attribute name="orderid" type="xs:string"/>

<!-- definition of complex elements -->
<xs:element name="shipto">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="address"/>
      <xs:element ref="city"/>
      <xs:element ref="country"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="item">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="title"/>
      <xs:element ref="note" minOccurs="0"/>
      <xs:element ref="quantity"/>
      <xs:element ref="price"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="orderperson"/>
      <xs:element ref="shipto"/>
      <xs:element ref="item" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute ref="orderid" use="required"/>
  </xs:complexType>
</xs:element> </xs:schema>
```

Same Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:simpleType name="stringtype">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="inttype">
  <xs:restriction base="xs:positiveInteger"/>
</xs:simpleType>
<xs:simpleType name="dectype">
  <xs:restriction base="xs:decimal"/>
</xs:simpleType>
<xs:simpleType name="orderidtype">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{6}"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:complexType name="shiptotype">
  <xs:sequence>
    <xs:element name="name" type="stringtype"/>
    <xs:element name="address" type="stringtype"/>
    <xs:element name="city" type="stringtype"/>
    <xs:element name="country" type="stringtype"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="itemtype">
  <xs:sequence>
    <xs:element name="title" type="stringtype"/>
    <xs:element name="note" type="stringtype"
minOccurs="0"/>
    <xs:element name="quantity" type="inttype"/>
    <xs:element name="price" type="dectype"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="shiporder type">
  <xs:sequence>
    <xs:element name="orderperson" type="stringtype"/>
    <xs:element name="shipto" type="shiptotype"/>
    <xs:element name="item" maxOccurs="unbounded"
type="itemtype"/>
  </xs:sequence>
  <xs:attribute name="orderid" type="orderidtype"
use="required"/>
</xs:complexType>
<xs:element name="shiporder" type="shiporder type"/>
</xs:schema>
```

2.2 Programming languages for XML

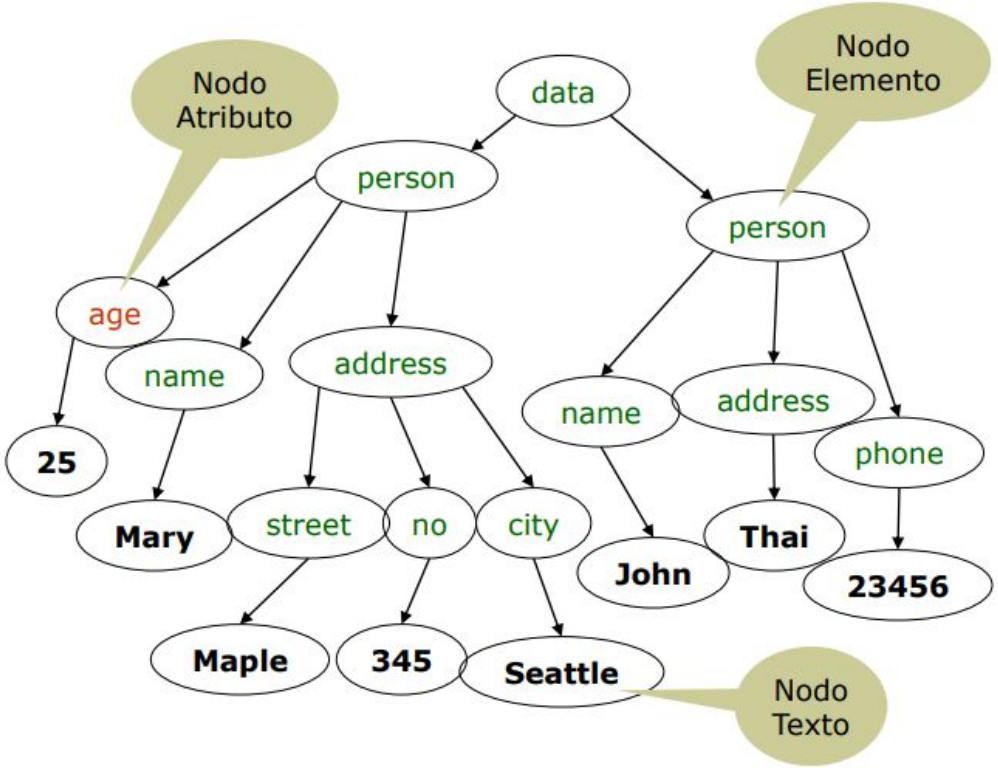
- Tools for querying and transformation of XML data are essential to extract information from large bodies of XML data, and to convert data between different representations (schemas) in XML.
- **XPath** is a language for path expressions and is actually a building block for XQuery.
- **XQuery** is the standard language for querying XML data. It is modeled after SQL but is significantly different, since it has to deal with nested XML data. XQuery also incorporates XPath expressions.
- **XSLT** language is designed for transforming XML. However, it is used primarily in document-formatting applications, rather in data-management applications.

Tree Model of XML

- A tree model of XML data is used in all these languages. An XMLdocument is modeled as a tree, with nodes corresponding to elements and attributes.
- **Element nodes** can have **child nodes**, which can be subelements or **attributes** of the element.
- Correspondingly, each node (whether attribute or element), other than the **root** element, has a **parent** node, which is an element.
- The order of elements and attributes in the XML document is modeled by the ordering of children of nodes of the tree.
- The terms **parent**, **child**, **ancestor**, **descendant**, and **siblings** are used in the tree model of XML data.

Example of a tree model

```
<data>
  <person age="25" >
    <name> Mary </name>
    <address>
      <street> Maple </street>
      <no> 345 </no>
      <city> Seattle </city>
    </address>
  </person>
  <person>
    <name> John </name>
    <address>Thailand</address>
    <phone> 23456  </phone>
  </person>
</data>
```

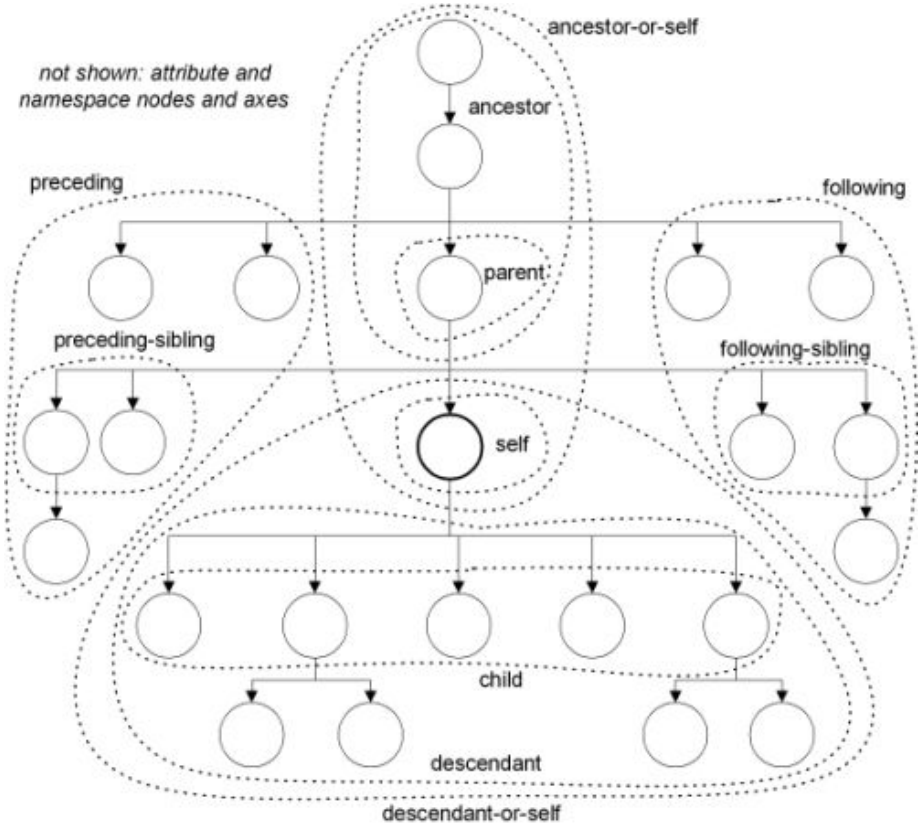


2.2.1 XPath

- XPath stands for XML Path Language
- XPath uses "path like" syntax to identify and navigate nodes in an XML document
- These path expressions look very much like the path expressions you use with traditional computer file systems.
- XPath contains over 200 built-in functions
- XPath is a major element in the XSLT standard
- XPath is a W3C recommendation

Relationship of Nodes

- Parent
  - Each element and attribute has one parent.
- Children
  - Element nodes may have zero, one or more children
- Siblings
  - Nodes that have the same parent.
- Ancestors
  - A node's parent, parent's parent, etc.
- Descendants
  - A node's children, children's children, etc.



Selecting Nodes

- XPath uses path expressions to select nodes in an XML document.
- The node is selected by following a path or steps.
- The most useful path expressions are listed below:

Expression	Description
`nodename`	Selects all nodes with the name "nodename"
`/`	Selects from the root node
`//`	Selects nodes in the document from the current node that match the selection no matter where they are
`.`	Selects the current node
`..`	Selects the parent of the current node
`@`	Selects attributes

Predicates

- Predicates are used to find a specific node or a node that contains a specific value.
- Predicates are always embedded in square brackets [ ].
- XPath wildcards can be used to select unknown XML nodes:

`*`	Matches any element node
`@*`	Matches any attribute node
`node()`	Matches any node of any kind
`comment()`	Matches any XML comment node
`text()`	Matches a node of type text
`processing-instruction()`	Matches XML processing instructions

- By using the `|` operator in an XPath expression is possible select several paths.

Functions (1/4)

- XSLT 2.0, XPath 2.0, and XQuery 1.0, share the same functions library (shown only some of them)

`number(arg)`	Returns the numeric value of the argument. The argument could be a *boolean*, *string*, or *node-set*
`abs(num)`	Returns the absolute value of the argument
`ceiling(num)`	Returns the smallest integer that is greater than the number argument
`floor(num)`	Returns the largest integer that is not greater than the number argument
`round(num)`	Rounds the number argument to the nearest integer
`string(arg)`	Returns the string value of the argument. The argument could be a *number*, *boolean*, or *node-set*
`compare(comp1,comp2)`	Returns -1 if *comp1* is less than *comp2*, 0 if *comp1* is equal to *comp2*, or 1 if *comp1* is greater than *comp2*
`concat(string,string,...)`	Returns the concatenation of the strings

Functions (2/4)

`string-join((string,string,...),sep)` Returns a string created by concatenating the string arguments and using the *sep* argument as the separator

`substring(string,start,len)`

`substring(string,start)` Returns the substring from the start position to the specified length. Index of the first character is 1. If *length* is omitted it returns the substring from the start position to the end

`string-length(string)`

`string-length()` Returns the length of the specified string. If there is no string argument it returns the length of the string value of the current node

`normalize-space()` Removes leading and trailing spaces from the specified string, and replaces all internal sequences of white space with one and returns the result. If there is no string argument it does the same on the current node

`upper-case(string)` Converts the string argument to upper-case

`lower-case(string)` Converts the string argument to lower-case

Functions (3/4)

`translate(string1,string2,string3)`	Converts *string1* by replacing the characters in *string2* with the characters in *string3*
`contains(string1,string2)`	Returns true if *string1* contains *string2*, otherwise it returns *false*
`starts-with(string1,string2)`	Returns true if *string1* starts with *string2*, otherwise it returns *false*
`ends-with(string1,string2)`	Returns true if *string1* ends with *string2*, otherwise it returns *false*
`substring-before(string1,string2)`	Returns the start of *string1* before *string2* occurs in it
`substring-after(string1,string2)`	Returns the remainder of *string1* after *string2* occurs in it
`matches(string,pattern)`	Returns *true* if the string argument matches the pattern, otherwise, it returns *false*
`replace(string,pattern,replace)`	Returns a string that is created by replacing the given pattern with the replace argument
`count((item,item,...))`	Returns the count of nodes
`avg((arg,arg,...))`	Returns the average of the argument values

Functions (4/4)

`max((arg,arg,...))`	Returns the argument that is greater than the others
`min((arg,arg,...))`	Returns the argument that is less than the others
`sum(arg,arg,...)`	Returns the sum of the numeric value of each node in the specified *node-set*
`id((string,string,...),node)`	Returns a sequence of element nodes that have an *ID value* equal to the value of one or more of the values specified in the string argument
`idref((string,string,...),node)`	Returns a sequence of element or attribute nodes that have an *IDREF value* equal to the value of one or more of the values specified in the string argument
`position()`	Returns the index position of the node that is currently being processed
`last()`	Returns the number of items in the processed node list
`current-dateTime()`	Returns the current dateTime (with timezone)
`current-date()`	Returns the current date (with timezone)
`current-time()`	Returns the current time (with timezone)

Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book>
  <title lang="en">Harry Potter</title>
  <price>29.99</price>
</book>
<book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
</book>
</bookstore>
```

`bookstore`
Selects all nodes with the name "bookstore"

`/bookstore`
Selects the root element bookstore
Note: If the path starts with a slash (/) it always represents an absolute path to an element!

`bookstore/book`
Selects all book elements that are children of bookstore

`//book`
Selects all book elements no matter where they are in the document

`bookstore//book`
Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element

`//@lang`
Selects all attributes that are named lang

Examples

`/bookstore/book[1]`
Selects the first book element that is the child of the bookstore element.

`/bookstore/book[last()]`
Selects the last book element that is the child of the bookstore element

`/bookstore/book[last()-1]`
Selects the last but one book element that is the child of the bookstore element

`/bookstore/book[position()<3]`
Selects the first two book elements that are children of the bookstore element

`//title[@lang]`
Selects all the title elements that have an attribute named lang

Examples

`//title[@lang='en']`
Selects all the title elements that have a "lang" attribute with a value of "en"

`/bookstore/book[price>35.00]`
Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00

`/bookstore/book[price>35.00]/title`
Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

`/bookstore/*`
Selects all the child element nodes of the bookstore element

Examples

`//*`
Selects all elements in the document

`//title[@*]`
Selects all title elements which have at least one attribute of any kind

`//book/title | //book/price`
Selects all the title AND price elements of all book elements

`//title | //price`
Selects all the title AND price elements in the document

`/bookstore/book/title | //price`
Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

2.2.2 XQuery

- XQuery is a language for finding and extracting elements and attributes from XML documents.
  - XQuery for XML is like SQL for databases
  - XQuery is built on XPath expressions
  - XQuery is supported by all major databases
  - XQuery is a W3C Recommendation
- XQuery can be used to:
  - Extract information to use in a Web Service
  - Generate summary reports
  - Transform XML data to XHTML
  - Search Web documents for relevant information

XQuery processing

- XQuery uses path expressions to navigate through elements in an XML document.
- XQuery uses predicates to limit the extracted data from XML documents.
- Example. The following predicate is used to select all the *book* elements under the *bookstore* element that have a *price* element with a value that is less than 30:

  ```
  doc("books.xml")/bookstore/book[price<30]
  ```
- *books.xml* is the file to be used, and the *doc()* function open it.

FLWOR Expressions

- FLWOR (pronounced "flower") is an acronym for "For, Let, Where, Order by, Return".
  For - selects a sequence of nodes
  Let - binds a sequence to a variable
  Where - filters the nodes
  Order by - sorts the nodes
  Return - what to return (gets evaluated once for every node)
- Same previous example with XQuery

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
return $x/title
```

XQuery Basic Syntax Rules

- Some basic syntax rules:
  - XQuery is case-sensitive
  - XQuery elements, attributes, and variables must be valid XML names
  - An XQuery string value can be in single or double quotes
  - An XQuery variable is defined with a `$` followed by a name, e.g. `$bookstore`
  - XQuery comments are delimited by `(:` and `:)`, e.g. `(: XQuery Comment :)`
- XQuery Conditional Expressions
  - "If-Then-Else" expressions are allowed in XQuery.

```
for $x in doc("books.xml")/bookstore/book
return if ($x/@category="CHILDREN")
then <child>{data($x/title)}</child>
else <adult>{data($x/title)}</adult>
```

XQuery Comparisons

- In XQuery there are two ways of comparing values.
  1. General comparisons: =, !=, <, <=, >, >=
  2. Value comparisons: eq, ne, lt, le, gt, ge
- The difference between both ways are:
  - The following expression returns true if any *q* attributes have a value greater than 10:
    `$bookstore//book/@q > 10`
  - The following expression returns true if there is only one *q* attribute returned by the expression, and its value is greater than 10. If more than one q is returned, an error occurs:
    `$bookstore//book/@q gt 10`

XQuery Selecting and Filtering

- The *for* Clause
  - The *for* clause binds a variable to each item returned by the in expression.
  - The *for* clause results in iteration.
  - There can be multiple for clauses in the same FLWOR expression.
  - To loop a specific number of times in a *for* clause, you may use the ***to*** keyword.

```
for $x in (1 to 5)
return <test>{$x}</test>
```

  - The ***at*** keyword can be used to count the iteration

```
for $x at $i in doc("books.xml")/bookstore/book/title
return <book>{$i}. {data($x)}</book>
```

  - It is also allowed with more than one in expression in the *for* clause. Use comma to separate each in expression

```
for $x in (10,20), $y in (100,200)
return <test>x={$x} and y={$y}</test>
```

FLWOR expressions

- The *let* Clause
  - The *let* clause allows variable assignments and it avoids repeating the same expression many times. The *let* clause does not result in iteration.

```
let $x := (1 to 5)
return <test>{$x}</test>
```

- The *where* Clause
  - The *where* clause is used to specify one or more criteria for the result

```
where $x/price>30 and $x/price<100
```

- The *order by* Clause
  - The *order by* clause is used to specify the sort order of the result.

```
for $x in doc("books.xml")/bookstore/book
order by $x/@category, $x/title
return $x/title
```

Generating results

- The *return* Clause
  - The *return* clause specifies what is to be returned

```
<html>
<body>
<h1>Bookstore</h1>
<ul>
{
for $x in doc("books.xml")/bookstore/book
order by $x/title
return <li>{data($x/title)}. Category: {data($x/@category)}</li>
}
</ul>
</body>
</html>
```

2.2.3 Extensible Stylesheet Language

- XSL (eXtensible Stylesheet Language) is a styling language for XML.
- XSLT stands for XSL Transformations.
- XSLT is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X)HTML element.

Declaration

- The correct way to declare an XSL style sheet according to the W3C XSLT Recommendation is:
  <**xsl:stylesheet** version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
- Link the XSL Style Sheet to the XML Document
  Add the XSL style sheet reference to your XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="myxslt.xsl"?>
```

XSLT tags (1/3)

- An XSL style sheet consists of one or more set of rules that are called templates.
- A template contains rules to apply when a specified node is matched.
- The `<xsl:template>` element is used to build templates.
  - The **match** attribute is used to associate a template with an XML element.
  - The value of the **match** attribute is an XPath expression
  - `<xsl:template match="`*XPath*`">`
- The `<xsl:value-of>` element can be used to extract the value of an XML element and add it to the output stream of the transformation
  - `<xsl:value-of select="`*Xpath*`"/>`
- The XSL `<xsl:for-each>` element can be used to select every XML element of a specified node-set
  - `<xsl:for-each select="`*XPath*`">`

XSLT tags (2/3)

- The `<xsl:sort>` element is used to sort the output.
  - The **select** attribute indicates what XML element to sort on.
  - `<xsl:sort select="`*element*`"/>`
- The `<xsl:if>` element is used to put a conditional test against the content of the XML file.
  - The value of the required **test** attribute contains the expression to be evaluated
  - ```
    <xsl:if test="expression">
      ...some output if the expression is true...
    </xsl:if>
    ```
    ```

XSLT tags (3/3)

- The `<xsl:choose>` element is used in conjunction with `<xsl:when>` and `<xsl:otherwise>` to express multiple conditional tests.
  - ```
    <xsl:choose>
      <xsl:when test="expression">
        ... some output ...
      </xsl:when>
      <xsl:otherwise>
        ... some output ....
      </xsl:otherwise>
    </xsl:choose>
    ```
- The `&lt;xsl:apply-templates&gt;` element applies a template to the current element or to the current element's child nodes.
  - `<xsl:template match="XPath">`
    ```

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
href="cdcatalog.xsl"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
.
.
</catalog>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
  <body>
  <h2>My CD Collection</h2>
  <table border="1">
    <tr bgcolor="#9acd32">
      <th>Title</th>
      <th>Artist</th>
    </tr>
    <xsl:for-each select="catalog/cd">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <xsl:choose>
        <xsl:when test="price &gt; 10">
          <td bgcolor="#ff00ff">
          <xsl:value-of select="artist"/></td>
        </xsl:when>
        <xsl:otherwise>
          <td><xsl:value-of select="artist"/></td>
        </xsl:otherwise>
      </xsl:choose>
    </tr>
    </xsl:for-each>
  </table>
  </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

Unit 3

Database Systems Implementation

Syllabus (1/2)

- 3.1 Index structures
  - 3.1.1 Index-Structure Basics
  - 3.1.2 B-trees
  - 3.1.3 Hash tables
  - 3.1.4 Multidimensional indexes
- 3.2 Query execution
  - 3.2.1 Scanning
  - 3.2.2 Hashing
  - 3.2.3 Sorting
  - 3.2.4 Indexing

Syllabus (2/2)

3.3 Query optimization
- 3.3.1 Algebraic laws for improving query plan
- 3.3.2 Estimating the cost of operations
- 3.3.3 Cost-based plan selection
- 3.3.4 Order of joins

3.4 Concurrency control
- 3.4.1 Serial and Serializable schedules
- 3.4.2 Enforcing serializability by locks
- 3.4.3 Locking systems with several lock modes

3.5 Transaction management
- 3.5.1 Serializability and Recoverability
- 3.5.2 Deadlocks

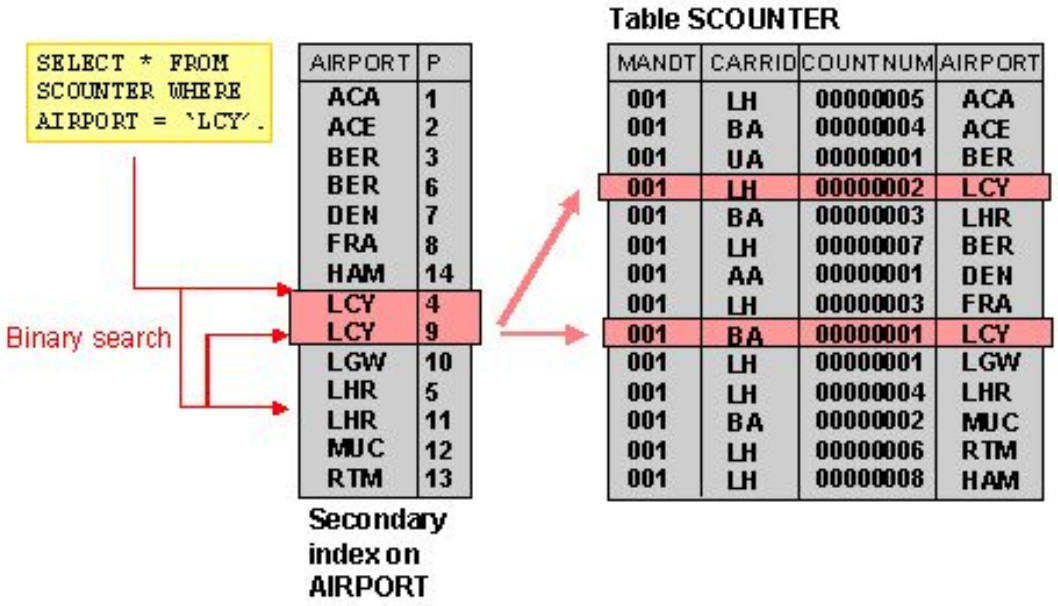
3.1 Index structures

- Indexes are additional auxiliary access structures, which are used to speed up the retrieval of records in response to certain search conditions.
- The index structures are additional files on disk that provide secondary access paths, which provide alternative ways to access the records without affecting the physical placement of records in the primary data file on disk.
- They enable efficient access to records based on the indexing fields that are used to construct the index.
- Basically, any field of the file can be used to create an index, and multiple indexes on different fields—as well as indexes on multiple fields—can be constructed on the same file.

3.1.1 Index-Structure Basics

- An index access structure is usually defined on a single field of a file, called an indexing field (or indexing attribute).
- The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value.
- The values in the index are ordered so that we can do a binary search on the index.
    - If both the data file and the index file are ordered, and since the index file is typically much smaller than the data file, searching the index using a binary search is a better option.

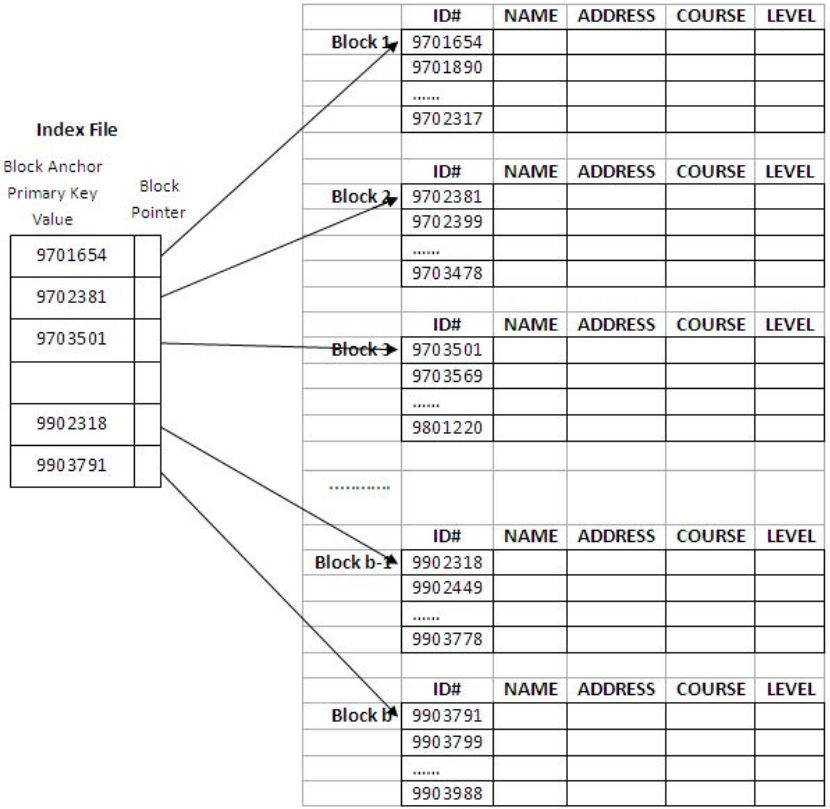
Searching in index

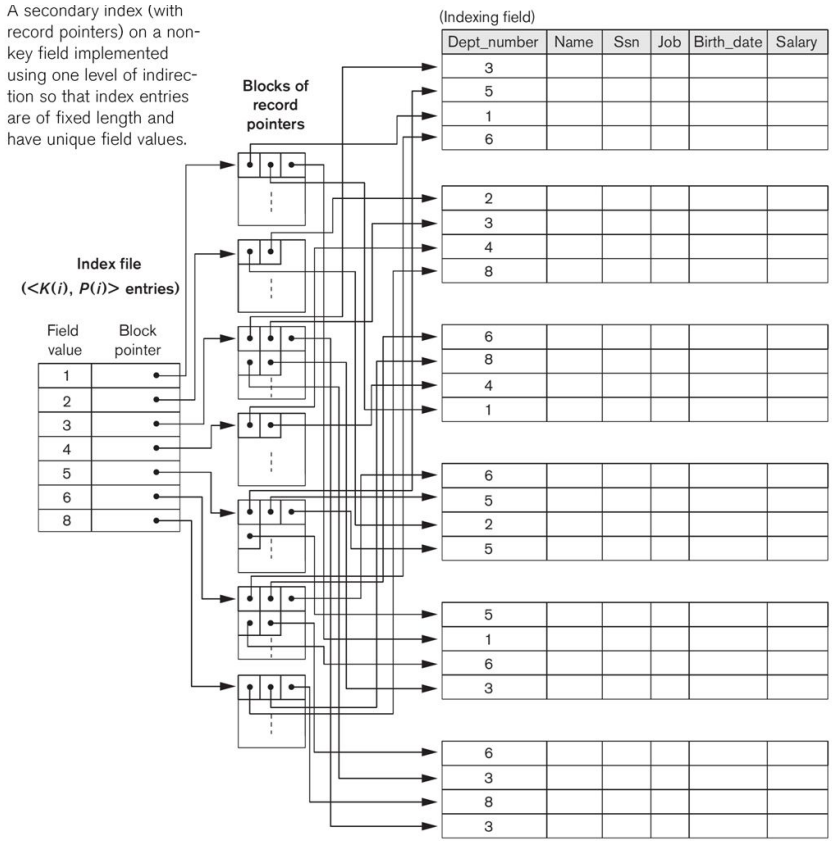


Types of indexes

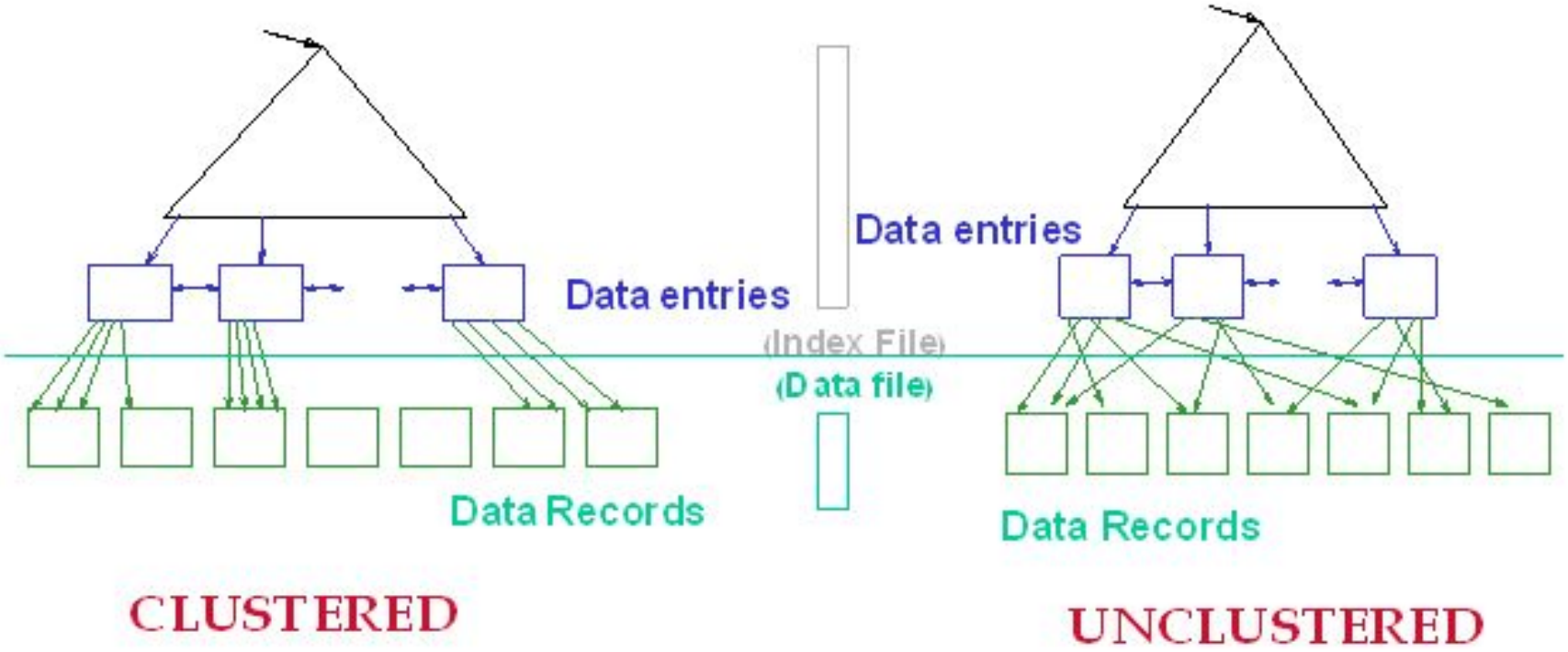
- There are several types of ordered indexes.
  - A **primary index** is specified on the ordering key field of an ordered file of records.
  - If the ordering field is not a key field—that is, if numerous records in the file can have the same value for the ordering field— another type of index, called a clustering index, can be used.
  - A third type of index, called a **secondary index**, can be specified on any nonordering field of a file.

Primary vs. Secondary Indexes



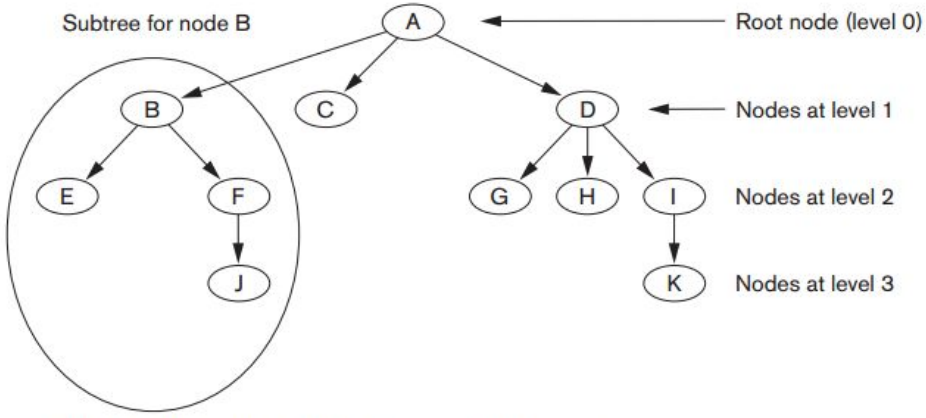


Clustered vs. unclustered Indexes



3.1.2 B-trees

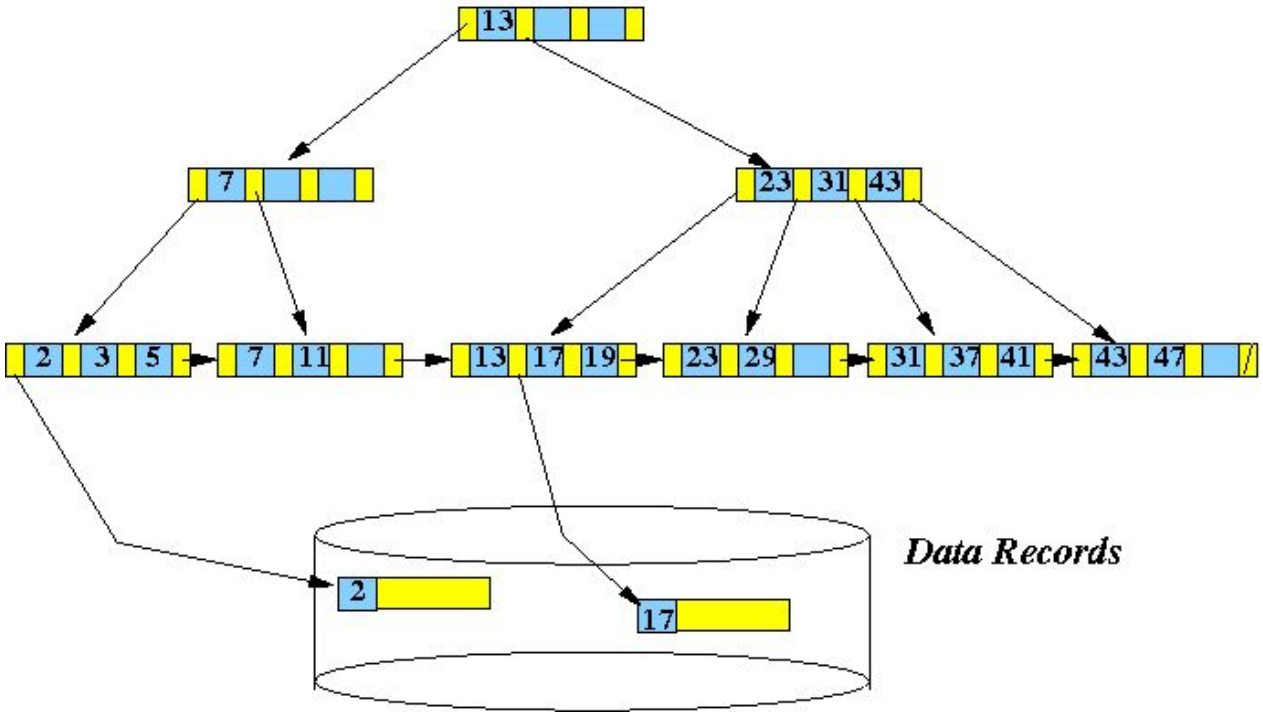
- A tree is formed of nodes.
- Each node in the tree, except for a special node called the root, has one parent node and zero or more child nodes.
- A node that does not have any child nodes is called a leaf node; a non leaf node is called an internal node.
- The root node has no parent.



B-tree structure

- The B-tree has additional constraints that ensure that the tree is always balanced
  - The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints
  - When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split.
- Each internal node of a B-tree contains a number of keys.
  - The keys act as separation values which divide its subtrees.
- A B-tree is kept balanced by requiring that all leaf nodes be at the same depth

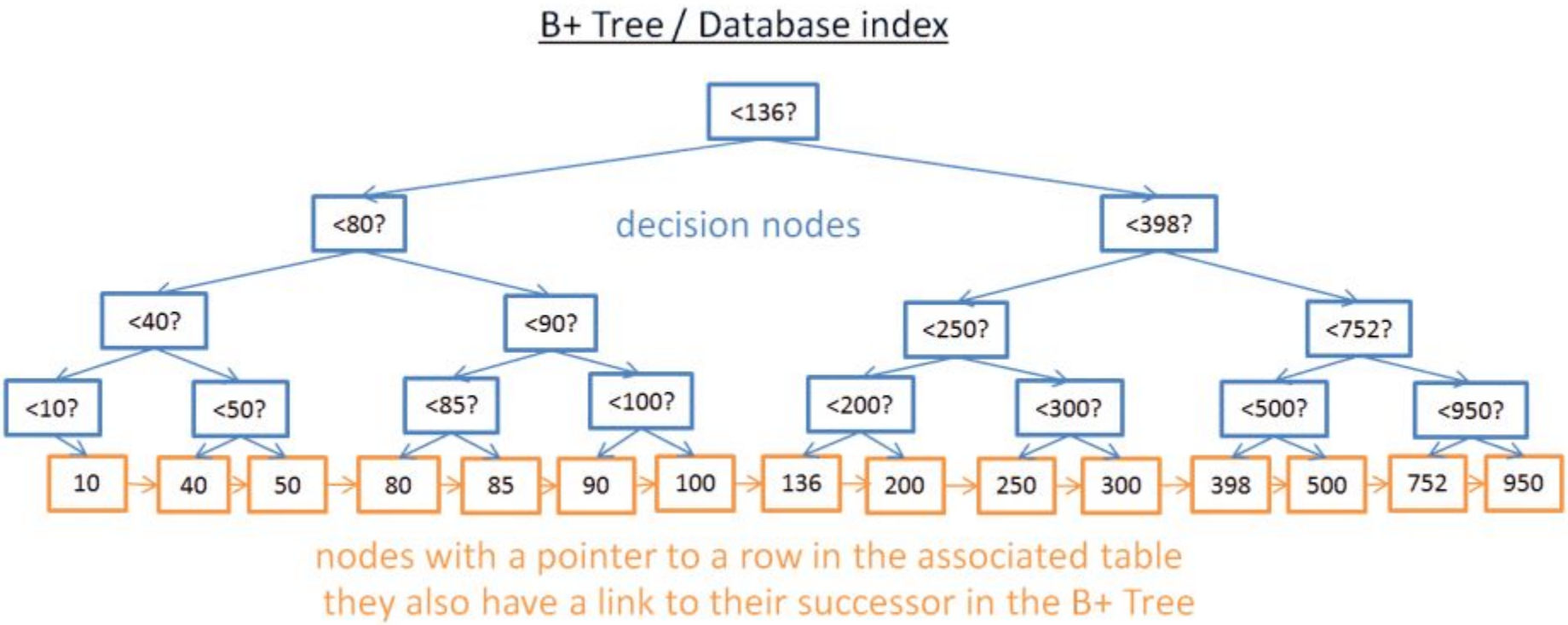
Example

- 

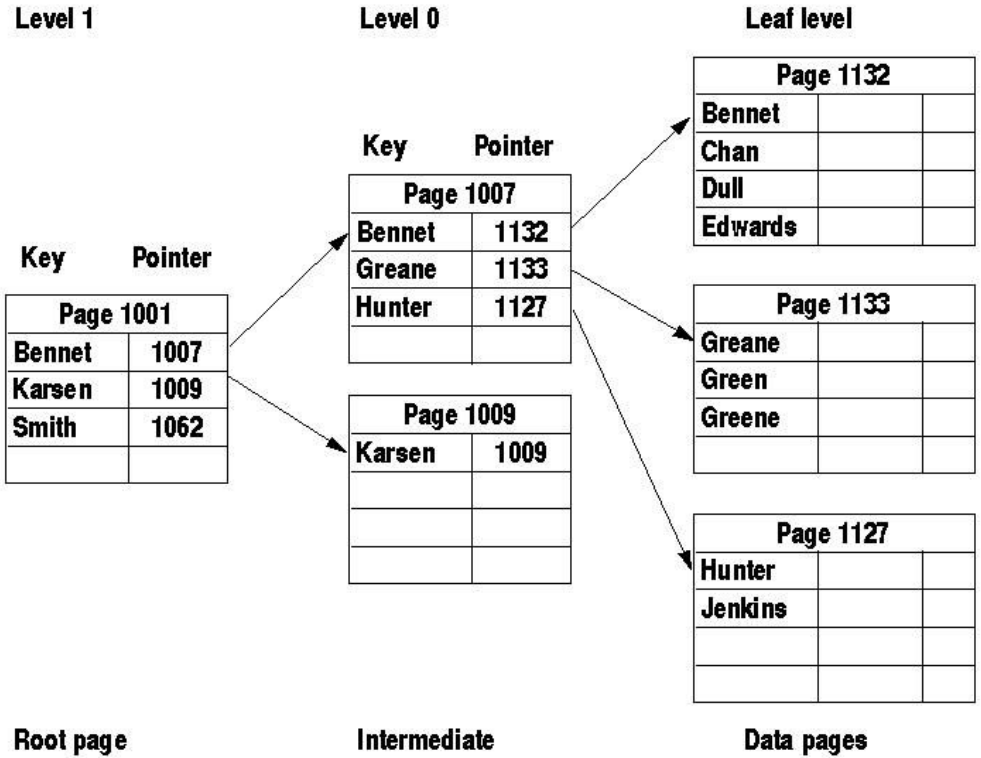
Variants

- A B-tree stores keys in its internal nodes but need not store those keys in the records at the leaves. The general class includes variations such as the B+ tree and the B* tree.
  - In the B+ tree, copies of the keys are stored in the internal nodes; the keys and records are stored in leaves; in addition, a leaf node may include a pointer to the next leaf node to speed sequential access.
  - The B* tree balances more neighboring internal nodes to keep the internal nodes more densely packed. This variant ensures non-root nodes are at least 2/3 full instead of 1/2

B+ Example


B+ Tree / Database index
<136?
<80?
decision nodes
<398?
<40?
<90?
<250?
<752?
<10?
<50?
<85?
<100?
<200?
<300?
<500?
<950?
10
40
50
80
85
90
100
136
200
250
300
398
500
752
950
nodes with a pointer to a row in the associated table
they also have a link to their successor in the B+ Tree

B-Tree index in Databases



Operational cost in primary index

Example: Primary Index

Block Size: 512 Bytes **Record Size: 35 Bytes** **nRecords: 6200**

bFactor: ⌊ 512 / 35 ⌋ = 14 rec/block **nBlocks: ⌈ 6200 / 14 ⌉ = 443 blocks**

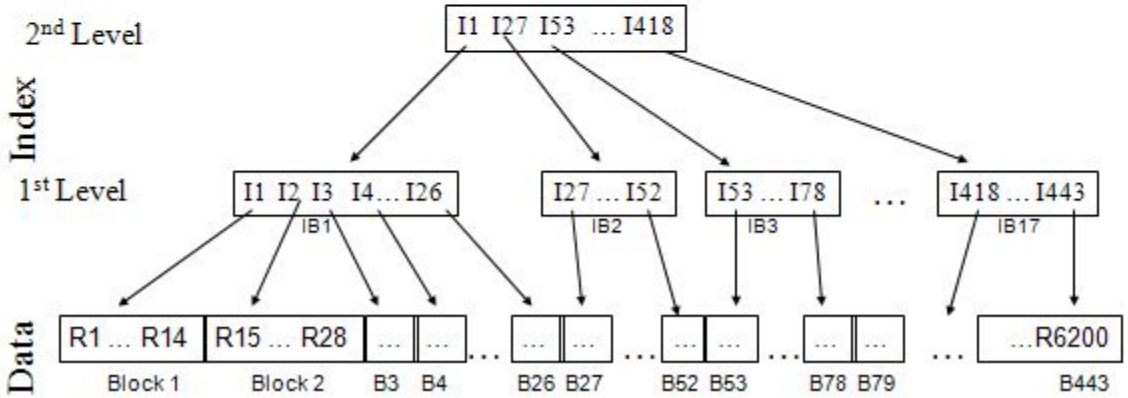
Index Entry Size: 19 Bytes **Fan Out: ⌊ 512 / 19 ⌋ = 26** index entry/block

First Index Level: ⌈ 443 / 26 ⌉ = ⌈17.03846 ⌉ = 18 Blocks

Second Index Level: ⌈ 18 / 26 ⌉ = 1 Block

Total Number of Levels: 2 Levels (check with Log_{26} 443)

Total Size of Primary Index: (18 + 1) = 19 Blocks



Operational cost in secondary index

Example: Secondary Index

Block Size: 512 Bytes **Record Size: 35 Bytes** **nRecords: 6200**

bFactor: ⌊ 512 / 35 ⌋ = 14 rec/block **nBlocks: ⌈ 6200 / 14 ⌉ = 443 blocks**

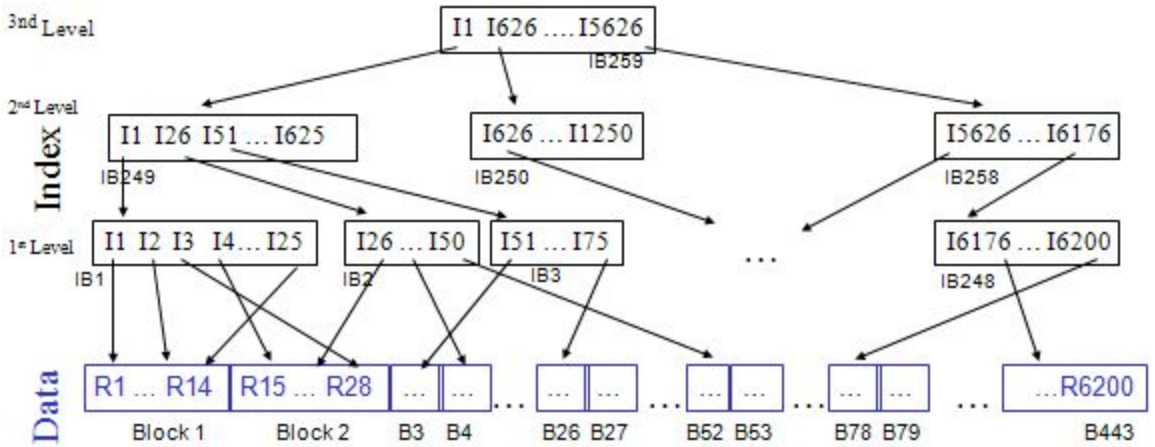
Index Entry Size: 20 Bytes **Fan Out: ⌊ 512 / 20 ⌋ = 25** index entry/block

First Index Level: ⌈ 6200 / 25 ⌉ = 248 Blocks

Second Index Level: ⌈ 248 / 25 ⌉ = 10 Blocks

Third Index Level: ⌈ 10 / 25 ⌉ = 1 Block **Total Size: 259 Blocks**

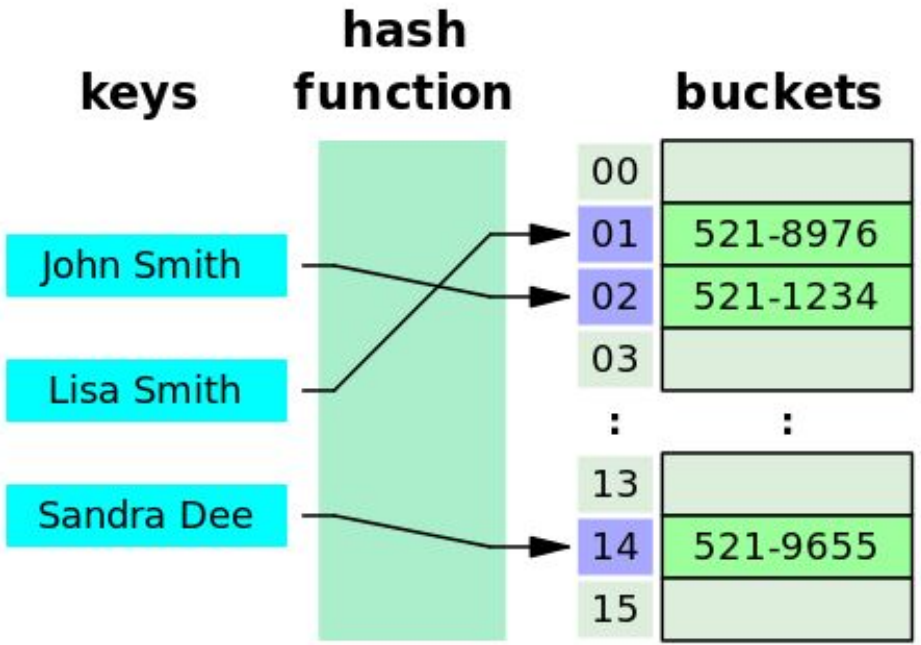
Total Number of Levels: 3 Levels (check with Log_{25} 6200)



3.1.3 Hash tables

- Is a data structure that implements an associative array abstract data type, a structure that can map keys to values.
- A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.
- the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash collisions where the hash function generates the same index for more than one key. Such collisions must be accommodated in some way.

Functionality of Hash table



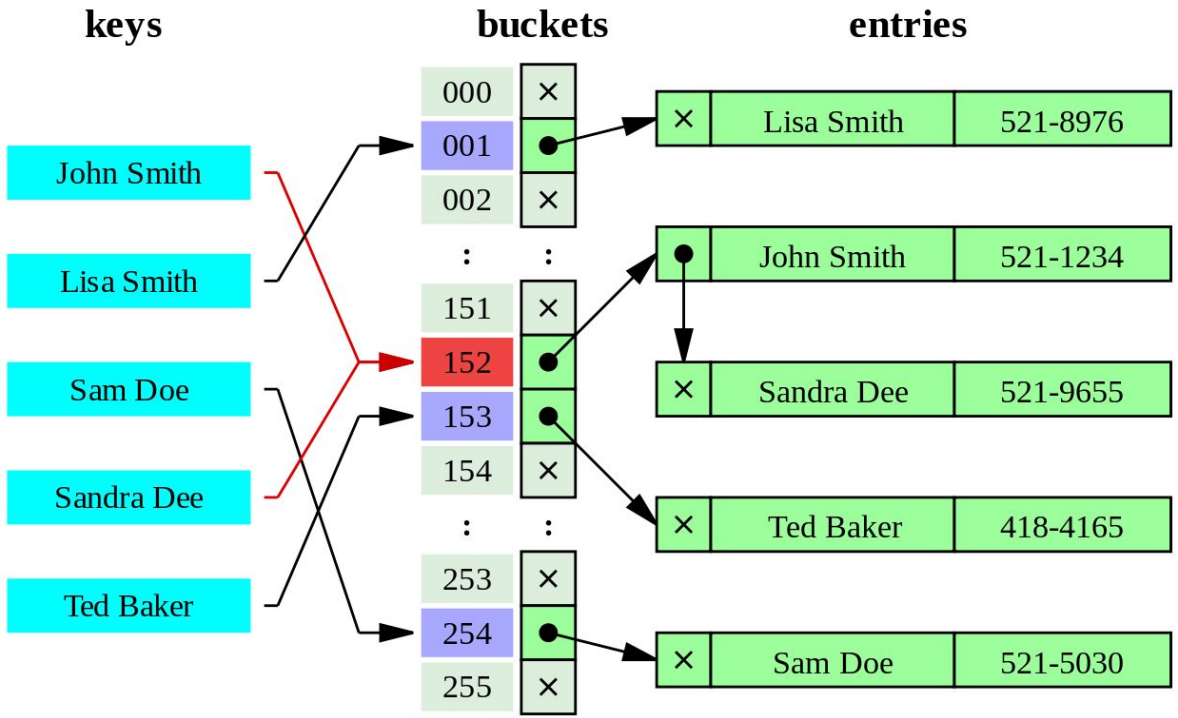
Advantages of hash tables

- In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table.
- In many situations, hash tables turn out to be on average more efficient than search trees or any other table lookup structure.

Hash function

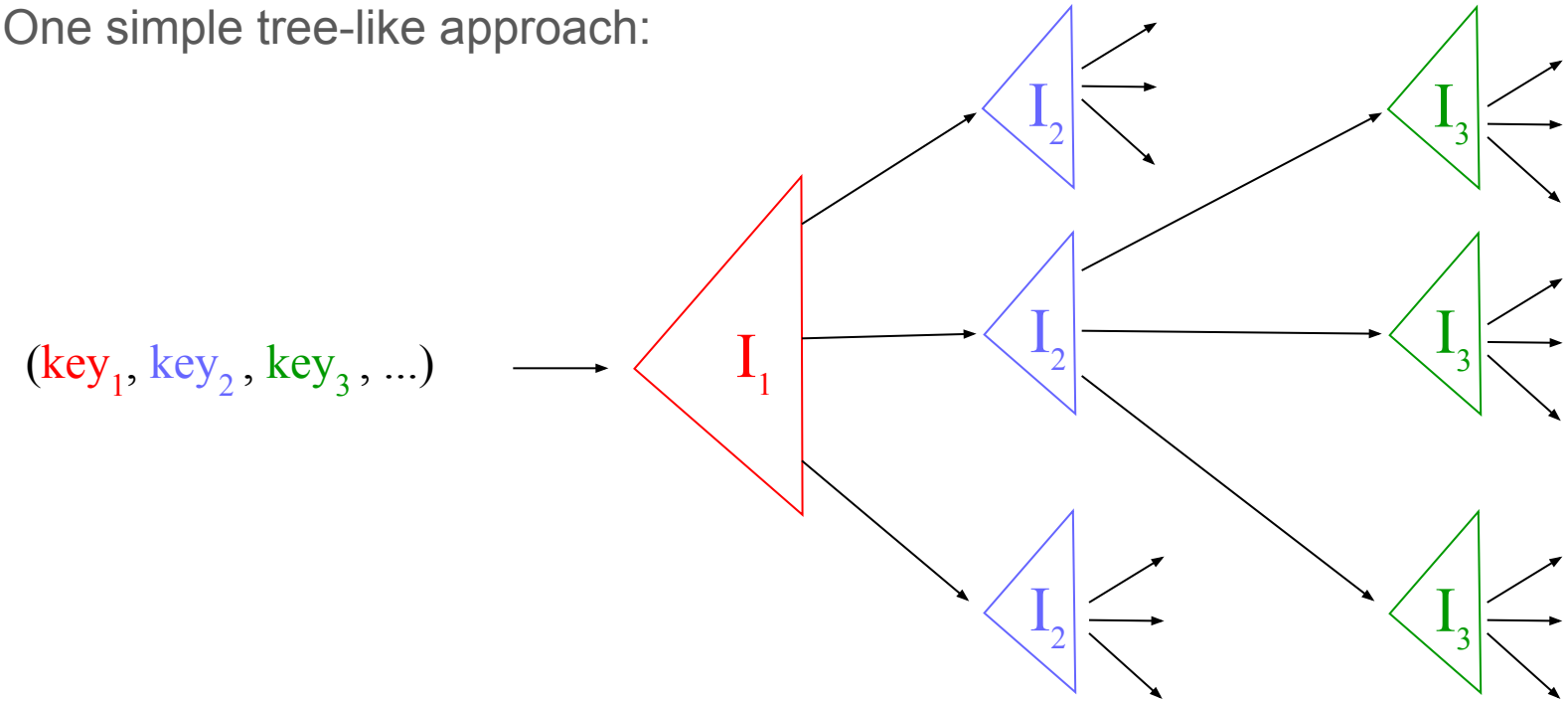
- A basic requirement is that the function should provide a uniform distribution of hash values.
- A non-uniform distribution increases the number of collisions and the cost of resolving them.
    - Uniformity is sometimes difficult to ensure by design, but may be evaluated empirically using statistical tests
- In a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that.

Collision resolution



3.1.4 Multidimensional indexes

- A multidimensional index combines several dimensions into one index
- One simple tree-like approach:



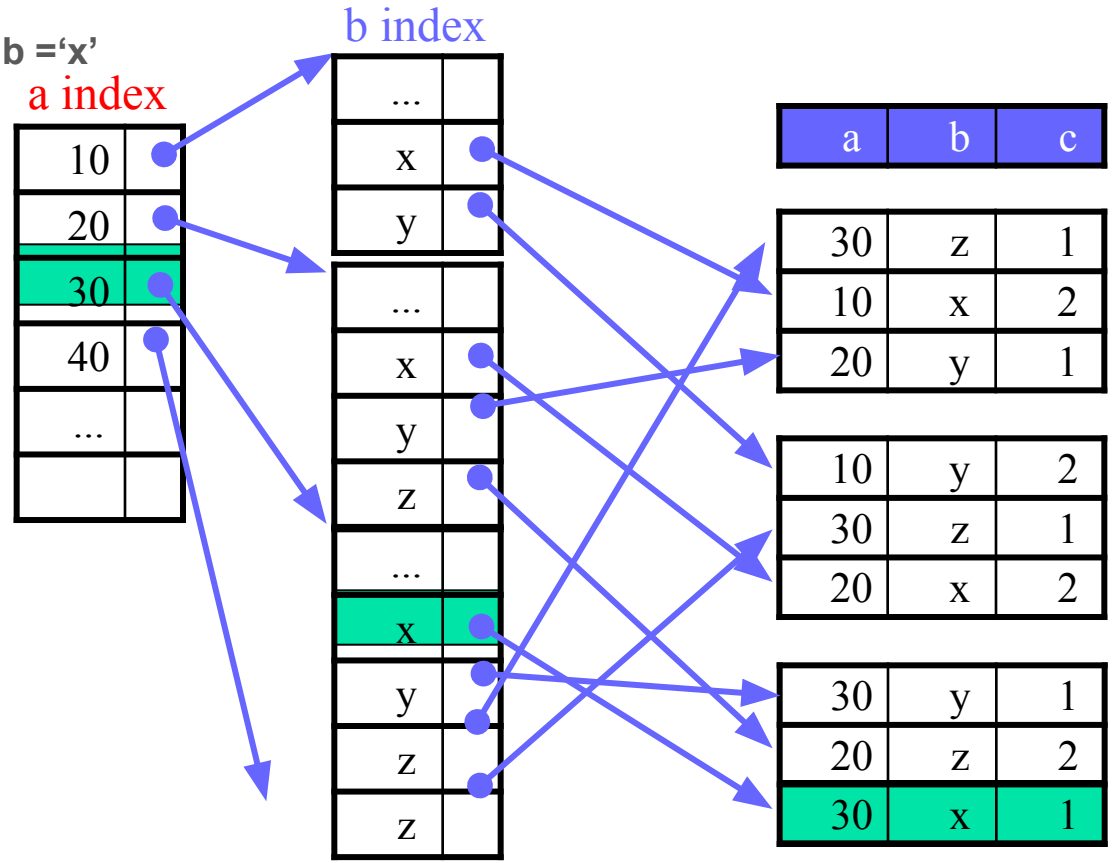
Example

SELECT ... FROM R WHERE a = 30 AND b ='x'

- search key = (30, x)
  read a-dimension

(30, x) →

- search for 30, find corresponding b-dimension index
- search for x, read corresponding disk block and get record
- select requested attributes

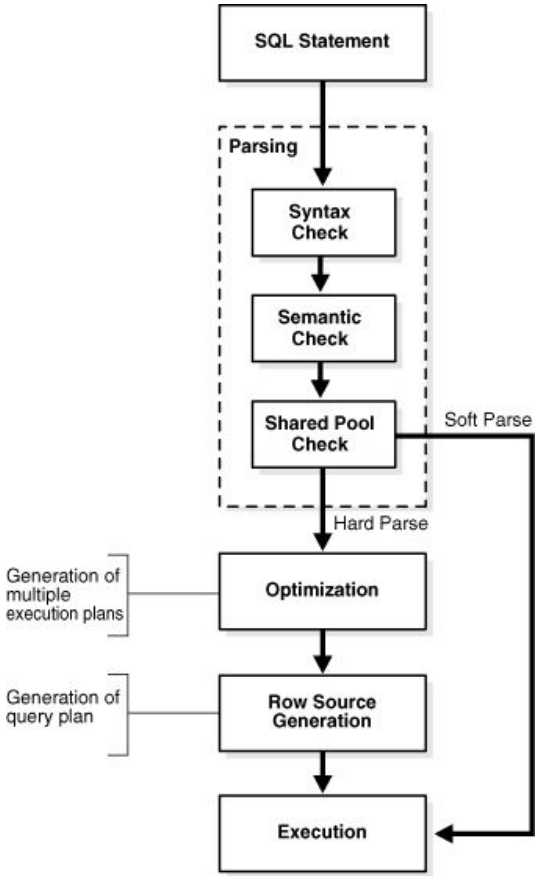


Usefull indexes

- For which queries is this index good?
  - find records where a = 10 AND b = ‘x’ -> good
  - find records where a = 10 AND b ≥ ‘x’ -> good
  - find records where a = 10 -> bad
  - find records where b = ‘x’ -> bad

3.2 Query execution

- SQL processing is the parsing, optimization, row source generation, and execution of a SQL statement.
  - The parsing stage involves separating the pieces of a SQL statement into a data structure that other routines can process.
  - During the optimization stage, database must perform a hard parse at least once for every unique DML statement and performs the optimization during this parse.
  - The row source generator receives the optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database.
  - During execution, the SQL engine executes each row source in the tree produced by the row source generator.

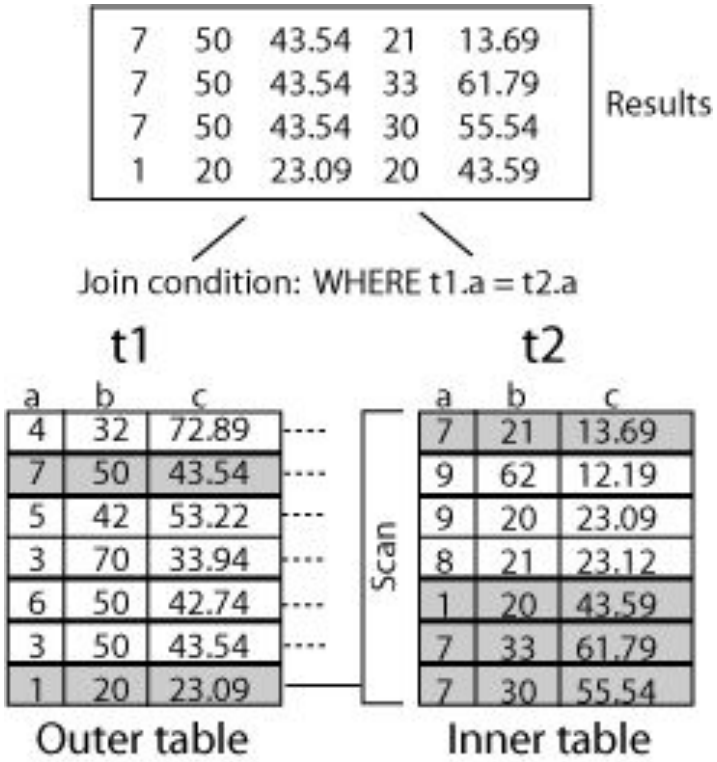


Overview of query execution

- Operations (steps) of query plan are represented using relational algebra (with bag semantics)
- Describe efficient algorithms to implement the relational algebra operations
- Major approaches are scanning, hashing, sorting and indexing
- Algorithms differ depending on how much main memory is available

3.2.1 Scanning

- Reads entire contents of relation R
- Needed for doing join, union, etc.
- To find all tuples of R:
  - Table scan: if addresses of blocks containing R are known and contiguous, easy to retrieve the tuples
  - Index scan: if there is an index on any attribute of R, use it to retrieve the tuples



3.2.2 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

- 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(Music) = 1 h(History) = 2 h(Physics) = 3 h(Elec. Eng.) = 3

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

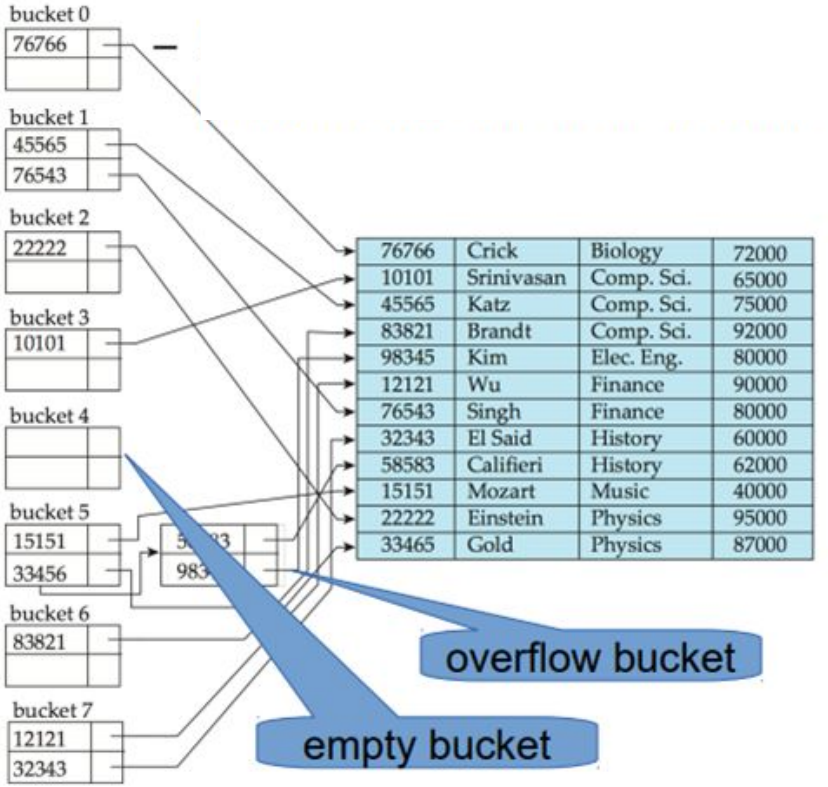
32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

Hash Index

- Hashing can be used not only for file organization, but also for index-structure creation
  - A hash index organizes the search keys, with their associated record pointers, into a hash file structure



3.2.3 Sorting

- Two steps:
  1) Created partially sorted data chunks
  2) Merge the partially sorted chunks
- First step:
  - Let *M* be the memory capacity
  - Create sorted runs. Let *i* be 0 initially

  Repeatedly do the following till the end of the relation:

  (a) Read *M* blocks of relation into memory

  (b) Sort the in-memory blocks

  (c) Write sorted data to run R_i; increment *i*

  Let the final value of *i* be *N*

Sorting (2)

- Second step: merge the runs
  - Merge the runs (*N*-way merge). We assume (for now) that $N < M$.
  - Use *N* blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page

  repeat

  Select the first record (in sort order) among all buffer pages

  Write the record to the output buffer. If the output buffer is full write it to disk.

  Delete the record from its input buffer page.

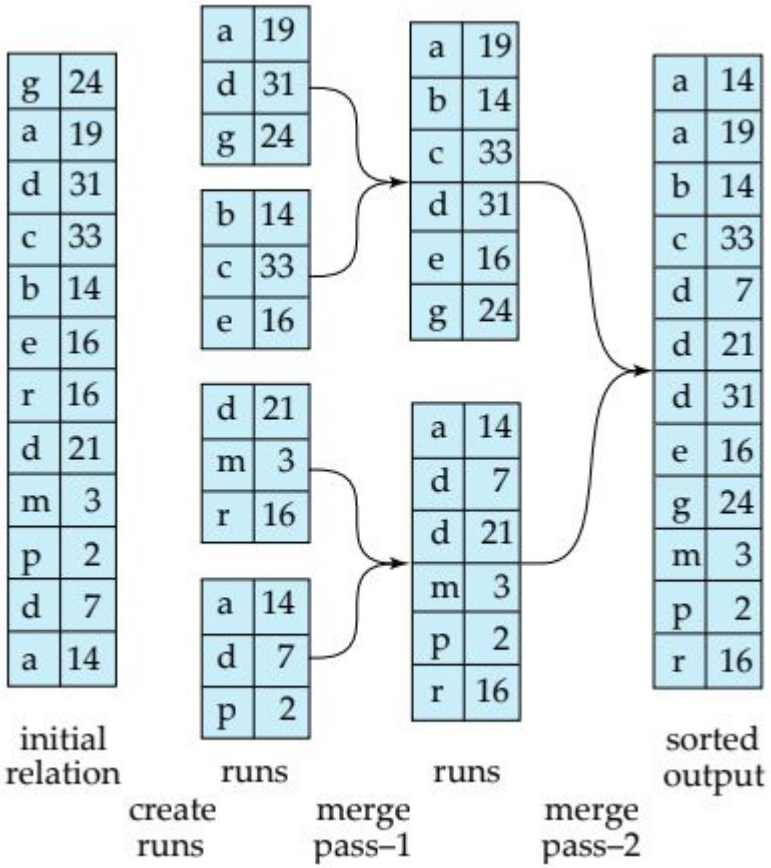
  If the buffer page becomes empty **then** read the next block (if any) of the run into the buffer.

  until all input buffer pages are empty

  • If $N >= M$, several merge passes are required

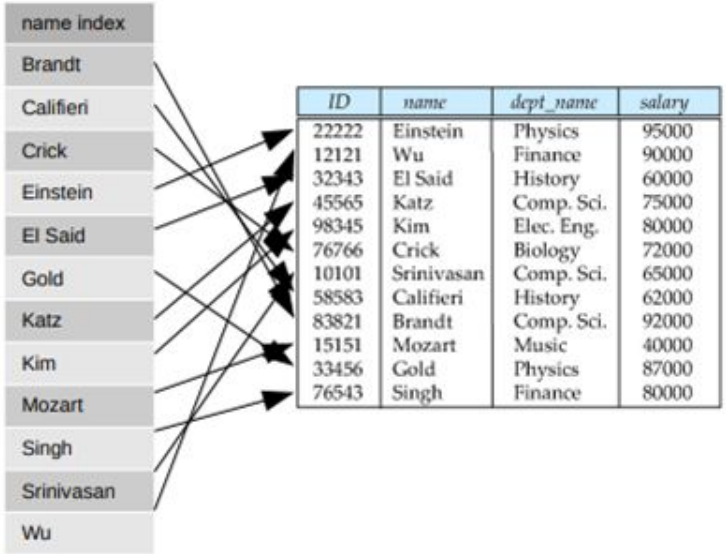
  – In each pass, contiguous groups of M - 1 runs are merged

Use sorting


initial relation
runs
runs
sorted output
create runs
merge pass–1
merge pass–2

3.2.4 Indexing

- Basic idea
  - Search in index is O(log2N)
  - Following link is O(1)
  - Each index can remain sorted
  - Create an index for each attribute which you may use in a query
- Trade-off
  - Faster queries
  - Some redundancy
    - But this is handled by the DBMS!
    - i.e., mainly a storage capacity problem, not so much a consistency problem



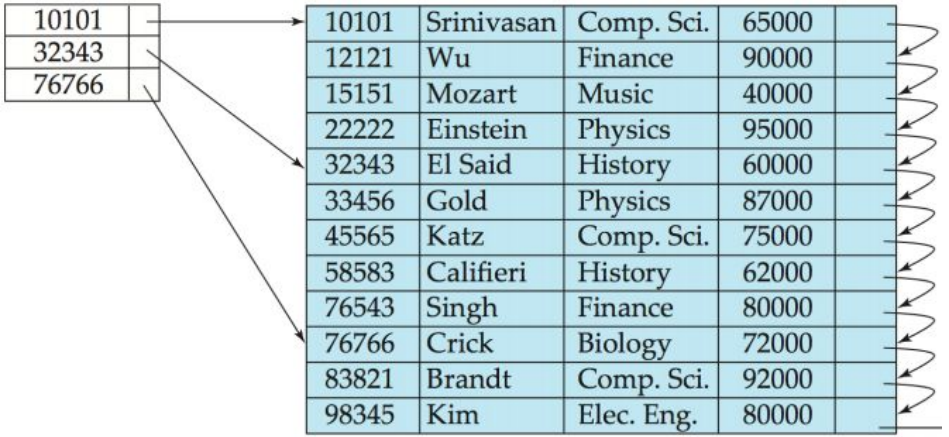
Index basics

- Indexing mechanisms used to speed up access to desired data
  - e.g., searching by a specific attribute
  - but also: joins!
    - 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:

  - 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”

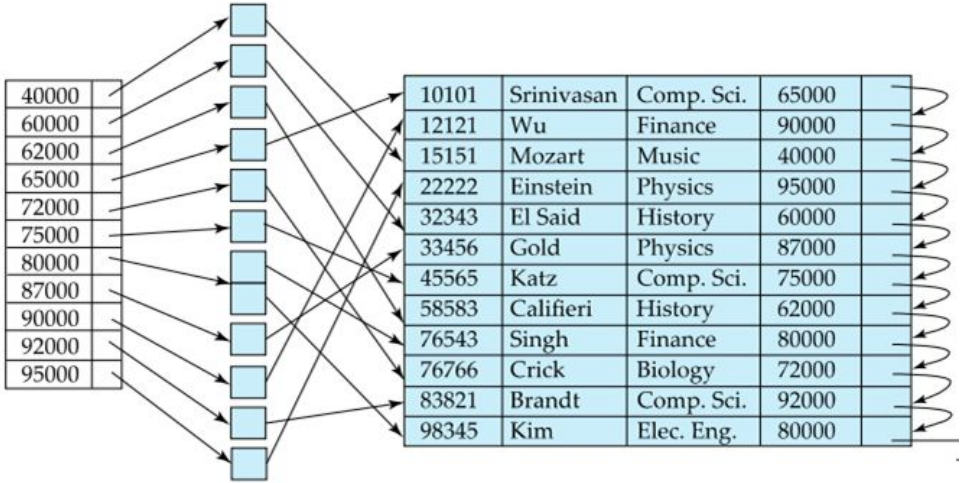
Sparse Index

- Sparse Index: contains index records for only some 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 that record



Secondary Index

- Secondary index: index on any other attribute
  - 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

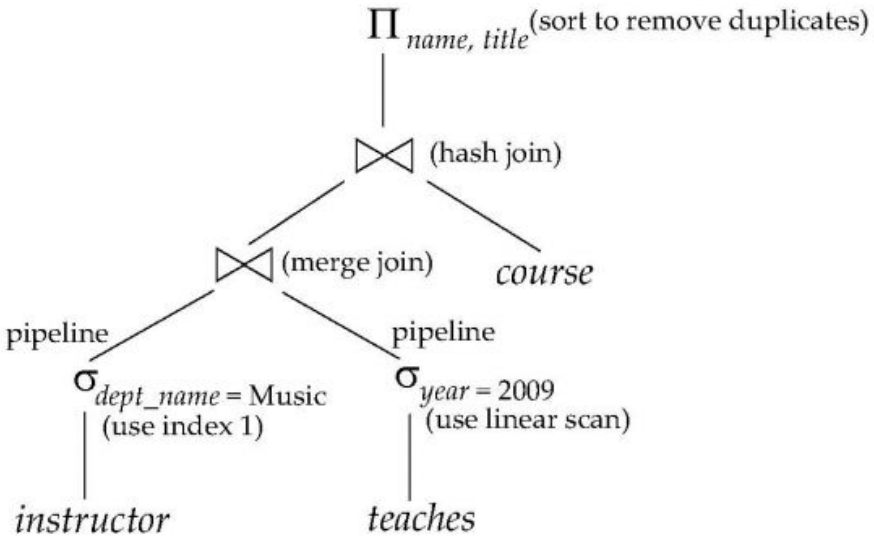


3.3 Query optimization

- Operations (steps) of query plan are represented using relational algebra (with bag semantics)
- Describe efficient algorithms to implement the relational algebra operations
- Major approaches are scanning, hashing, sorting and indexing
- Algorithms differ depending on how much main memory is available

3.3.1 Algebraic laws for improving query plan

- An evaluation plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated



Estimating costs

- 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
  - Generate logically equivalent expressions using equivalence rules
  - Annotate resultant expressions to get alternative query plans
  - Choose the cheapest plan based on estimated cost
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - 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

Equivalence in relational algebra

- Two relational algebra expressions are said to be equivalent if the two expressions generate the same set of tuples on every legal database instance
  – order of tuples is irrelevant
  – they may yield different results on databases that violate integrity constraints
- Equivalent results must not be a result of chance, e.g.
  – SELECT name FROM employee WHERE id=“12345” → “Smith”
  – SELECT name FROM employee WHERE birthday=“30.10.1974” → “Smith”
- Those results could be different on a different database instance

Equivalence rules (1)

- (1) Conjunctive selection operations can be deconstructed into a sequence of individual selections.
  $\sigma_{\theta 1 \wedge \theta 2}(E) = \sigma_{\theta 1}(\sigma_{\theta 2}(E))$

```
SELECT name, title
FROM instructor
WHERE dept_name=”Music”
AND salary>50000

SELECT name,title FROM (
SELECT name,title FROM instructor
WHERE dept_name=”Music”)
WHERE salary>50000
```

Equivalence rules (2)

- (2) Selection operations are commutative.
  $\sigma_{\theta 1}(\sigma_{\theta 2}(E)) = \sigma_{\theta 2}(\sigma_{\theta 1}(E))$

SELECT name,title FROM (
SELECT name,title FROM instructor
WHERE dept_name=”Music”)
WHERE salary>50000

SELECT name,title FROM (
SELECT name,title FROM instructor
WHERE salary>50000)
WHERE dept_name=”Music”

Equivalence rules (3)

- (3) Only the last in a sequence of projection operations is needed, the others can be omitted.
  $\pi_{L1}(\pi_{L2}(... (\pi_{Ln} (E))...)) = \pi_{L1} (E)$

```
SELECT name, title
FROM (
SELECT name,title,salary,dept_name FROM instructor
WHERE dept_name=”Music”
AND salary>50000
)

SELECT name,title FROM instructor
WHERE dept_name=”Music”
AND salary>50000
```

Equivalence rules (4)

- (4) Selections can be combined with Cartesian products and theta joins.
  $\sigma_{\theta}(E_1 X E_2) = E_1 \bowtie_{\theta} E_2$
  $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

```
SELECT name, building
FROM instructor, department
WHERE instructor.dept_name = department.dept_name
AND salary>50000

SELECT name, building FROM(
SELECT name,building FROM instructor,department)
WHERE instructor.dept_name = department.dept_name
AND SALARY>50000
```

Equivalence rules (5)

- (5) Theta-join operations (and natural joins) are commutative
  $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$

```
SELECT name, building
FROM instructor, department
WHERE instructor.dept_name = department.dept_name
AND salary>50000

SELECT name, building
FROM department, instructor
WHERE instructor.dept_name = department.dept_name
AND salary>50000
```

Equivalence rules (6)

- (6) Natural join operations are associative
  $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$

```
SELECT * FROM instructor, (
SELECT * FROM teaches, course
WHERE teaches.course_ID = course.course_ID) AS joined
WHERE instructor.inst_ID = joined.inst_ID

SELECT * FROM course, (
SELECT * FROM instructor,teaches WHERE
WHERE instructor.inst_ID = teaches.inst_ID) AS joined
WHERE course.course_ID = joined.course_ID
```

Equivalence rules (7)

- (7) Theta joins are associative in the following manner
  $(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$
  where θ2 involves attributes from only E_2 and E_3.

```
SELECT * FROM instructor, (
SELECT * FROM teaches, course
WHERE teaches.course_ID = course.course_ID) AS joined
WHERE instructor.inst_ID = joined.inst_ID
AND salary>50000

SELECT * FROM course, (
SELECT * FROM instructor,teaches WHERE
WHERE instructor.inst_ID = teaches.inst_ID) AS joined
WHERE course.course_ID = joined.course_ID
AND salary>50000
```

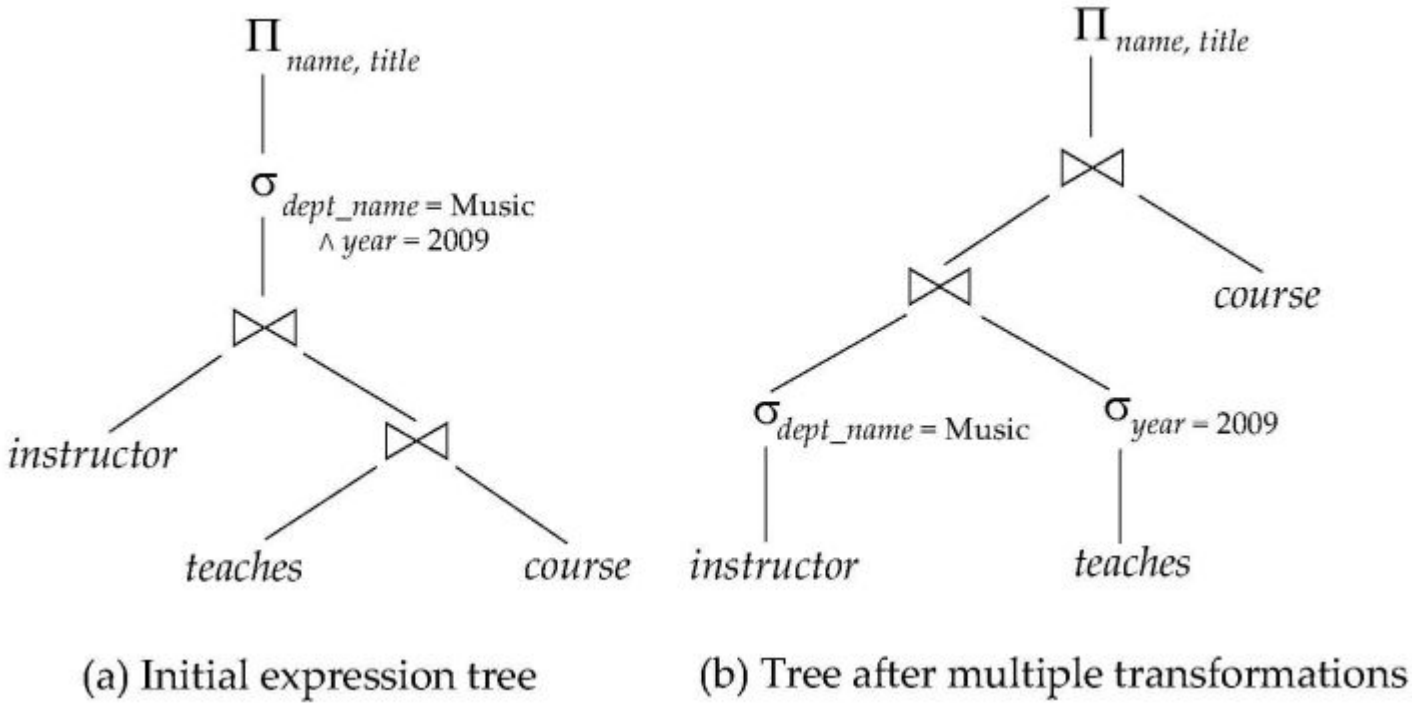
Equivalence rules (8)

- (8) The selection operation distributes over the theta join operation under the following two conditions:

  (a) If all the attributes in θ0 involve only the attributes of one of the expressions (E_1) being joined
  $\sigma_{\theta 0}(E_1 \bowtie E_2) = (\sigma_{\theta 0}(E_1)) \bowtie E_2$

  (b) If θ1 involves only the attributes of E_1 and θ2 involves only the attributes of E_2.
  $\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie E_2) = (\sigma_{\theta 1}(E_1)) \bowtie (\sigma_{\theta 2}(E_2))$

Example



(a) Initial expression tree

(b) Tree after multiple transformations

Equivalence rules (8)

- (9) The set operations union and intersection are commutative

$E_1 \cup E_2 = E_2 \cup E_1$

$E_1 \cap E_2 = E_2 \cap E_1$ (but: set difference is not commutative)

(10) Set union and intersection are associative

$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$

$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$

(11) The selection operation distributes over , and –.

$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta (E_2)$ and similarly for ∩ and ∪ in place of –

Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - E_2$ and similarly for ∩ in place of –, but not for ∪

(12) The projection operation distributes over union

$\boldsymbol{\pi}_L(E_1 \cup E_2) = (\boldsymbol{\pi}_L(E_1)) \cup (\boldsymbol{\pi}_L(E_2))$

Choosing a Good Execution Plan

- Naively: for each operation, pick the cheapest algorithm
  – given the statistics
  – caution: may not yield best overall algorithm!
- Example 1: merge-join may be costlier than hash-join
  – but may provide a sorted output which reduces the cost for an outer level aggregation
- Example 2: nested-loop join may be a costly variant
  – but provides opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches
  – Search all the plans and choose the best plan in a cost-based fashion
  – Uses heuristics to choose a plan

3.3.2 Estimating the cost of operations

- Parameters:
  - M : number of main-memory buffers available (size of buffer = size of disk block). Only count space needed for input and intermediate results, not output!
  - For relation R:
    B(R) or just B: number of blocks to store R
    T(R) or just T: number of tuples in R
    V(R,a) : number of distinct values for attribute a appearing in R
    Quantity being measured: number of disk I/Os.
    Assume inputs are on disk but output is not written to disk.

Cost of operations

–**B:** The number of data pages

–**R:** Number of records per page

–**D:** (Average) time to read or write disk page

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	D(log 2 B + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	D(log F 1.5B + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	D(1 + log F 0.15B)	D(log F 0.15B + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

Scan Primitive

- Reads entire contents of relation R
- Needed for doing join, union, etc.
- To find all tuples of R:
- Table scan: if addresses of blocks containing R are known and contiguous, easy to retrieve the tuples
- Index scan: if there is an index on any attribute of R, use it to retrieve the tuples

Costs of Scan Operators

- Table scan:
  - if R is clustered, then number of disk I/Os is approx. B(R).
  - if R is not clustered, number of disk I/Os could be as large as T(R).
- Index scan: approx. same as for table scan, since the number of disk I/Os to examine entire index is usually much much smaller than B(R).

Sort-Scan Primitive

- Produces tuples of R in sorted order w.r.t. attribute *a*
- Needed for sorting operator as well as helping in other algorithms
- Approaches:
  - If there is an index on *a* or if R is stored in sorted order of *a*, then use index or table scan.
  - If R fits in main memory, retrieve all tuples with table or index scan and then sort
  - Otherwise can use a secondary storage sorting algorithm

Costs of Sort-Scan

- See earlier slide for costs of table and index scans in case of clustered and unclustered files
- Cost of secondary sorting algorithm is:
  - approx. 3B disk I/Os if R is clustered
  - approx. T + 2B disk I/Os if R is not

One-Pass, Tuple-at-a-Time

- These are for SELECT and PROJECT
- Algorithm:
  - read the blocks of R sequentially into an input buffer
  - perform the operation
  - move the selected/projected tuples to an output buffer
  - Requires only $M \geq 1$
  - I/O cost is that of a scan (either B or T, depending on if R is clustered or not)
  - Exception! Selecting tuples that satisfy some condition on an indexed attribute can be done faster!

One Pass, Binary Operations

- Bag union:
- copy every tuple of R to the output, then copy every tuple of S to the output
- only needs $M \geq 1$
- disk I/O cost is B(R) + B(S)
- For set union, set intersection, set difference, bag intersection, bag difference, product, and natural join:
  - read smaller relation into main memory
  - use main memory search structure D to allow tuples to be inserted and found quickly
  - needs approx. min(B(R),B(S)) buffers
  - disk I/O cost is B(R) + B(S)

3.3.3 Cost-based plan selection

- 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

3.3.4 Order of joins

- For all relations r_1, r_2, and r_3,
  $(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$ (Rule 6)

  • If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

  $(r_1 \bowtie r_2) \bowtie r_3$

  so that we compute and store a smaller temporary relation

3.4 Concurrency control

- Multiple transactions are allowed to run concurrently in the system
  – Increased processor and disk utilization, leading to better transaction throughput
  • e.g., one transaction can be using the CPU while another is reading from or writing to the disk
  – Reduced average response time for transactions
- e.g., short transactions need not wait behind long ones
- Concurrency control schemes
  – mechanisms to achieve isolation
  – control the interaction among the concurrent transactions
  – prevent them from destroying the consistency of the database

Schedule

- Schedule
  – a sequence of instructions that specifies the chronological order in which instructions of concurrent transactions are executed
  – A schedule for a set of transactions must consist of all instructions of those transactions
  – Must preserve the order in which the instructions appear in each individual transaction
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  – By default, a transaction is assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Serial schedule example

- Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B
- Serial schedule: T_1 is executed as a whole, followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Wrong schedule example

- Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B
  - The sum of A and B is not maintained!

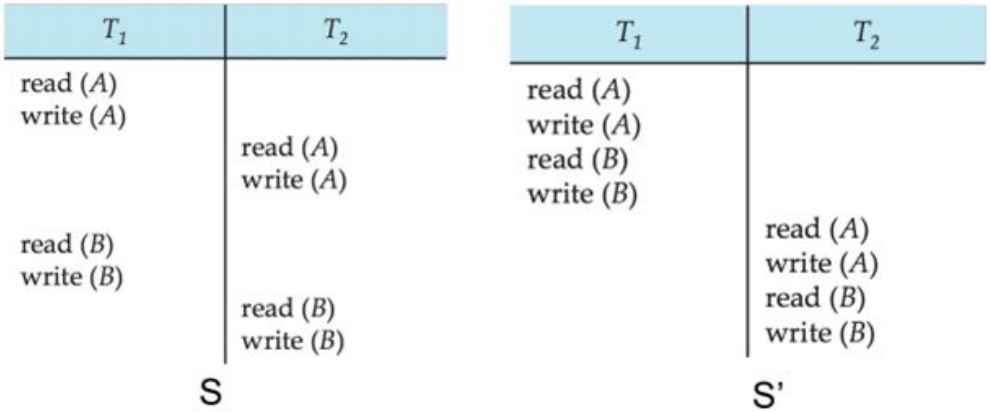
T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

3.4.1 Serial and Serializable schedules

- Basic assumption: transactions preserve database consistency
  – i.e., serial execution of a set of transactions also preserves database consistency
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule
  – We ignore operations other than read and write instructions
  – Transactions may perform arbitrary computations on data in between
  – Our simplified schedules consist of only read and write instructions

Conflict Equivalence and Serializability

- If a schedule S can be transformed into a schedule S´ by a series of swaps of non-conflicting instructions, we say that S and S´ are conflict equivalent.
  • We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule



Levels of Consistency

- Serializable: default
  • Repeatable read:
  – only committed records to be read
  – successive reads of same record must return the same value
  – transactions may not be serializable
  • Read committed:
  – only committed records can be read,
  – successive reads of record may return different (but committed) values
  • Read uncommitted:
  – even uncommitted records may be read

3.4.2 Enforcing serializability by locks

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using lock-X instruction
  2. shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction
- Lock requests are made to the concurrency-control manager
  – by the application accessing the database
  – transaction can proceed only after request is granted

3.4.3 Locking systems with several lock modes

- Transactions request locks
  – can be granted if the requested lock is compatible
- Compatibility:
  – Any number of transactions can hold shared locks on an item
  – If any transaction holds an exclusive on the item, no other transaction may hold any lock on the item
- If a lock cannot be granted
  – the requesting transaction has to wait until all incompatible locks are released

requested \ already granted	S	X
S	true	false
X	false	false

The Two-Phase Locking Protocol

- Protocol that ensures conflict serializable schedules
- Runs in two phases
- Phase 1: Growing Phase
  – Transaction may obtain and “upgrade” shared to exclusive locks
  – Transaction may not release locks
- Phase 2: Shrinking Phase
  – Transaction may release and “downgrade” exclusive to shared locks
  – Transaction may not obtain locks
- The protocol assures serializability
  – It can be proved that the transactions can be serialized in the order of their lock points,
  – i.e., the point where a transaction acquired its final lock

3.5 Transaction management

- A transaction is a unit of program execution that accesses and possibly updates various data items
- E.g., 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)
- Two main issues to deal with:
  – Failures of various kinds, such as hardware failures and system crashes
  – Concurrent execution of multiple transactions

ACID properties

- **Atomicity**: Either all operations of the transaction are properly reflected in the database, or none
- **Consistency**: Execution of a full transaction 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
  – i.e., for every pair of transactions Ti and Tj, it appears to Ti that either Tj, finished execution before Ti started, or Tj started execution after Ti finished
- **Durability**: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

3.5.1 Serializability and Recoverability

- A lock manager can be implemented as a separate process
  – transactions send lock and unlock requests to the lock manager
  – lock manager replies to a lock request by sending a lock grant messages
  – or a message asking the transaction to roll back, in case of a deadlock
  – The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a lock table to record granted locks and pending requests
  – The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Insert and Delete Operations

- If two-phase locking is used :
  – A delete operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted
  – A transaction that inserts a new tuple into the database is given an exclusive lock on the tuple
- Insertions and deletions can lead to the phantom phenomenon
- A transaction that scans a relation
  (e.g., read number of all accounts in Perryridge)
  and a transaction that inserts a tuple in the relation
  (e.g., insert a new account at Perryridge)
  (conceptually) conflict in spite of not accessing any tuple in common

Insert and Delete Operations

- The transaction scanning the relation is reading information that indicates what tuples the relation contains
  – while a transaction inserting a tuple updates the same information
- The conflict should be detected, e.g., by locking the information
- One solution:
  – Associate a data item with the relation, to represent the information about what tuples the relation contains
  – Transactions scanning the relation acquire a shared lock in the data item
  – Transactions inserting or deleting a tuple acquire an exclusive lock on the data item.
  (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/deletions
  – Index locking protocols provide higher concurrency while preventing the phantom phenomenon
  – requiring locks on certain index buckets

3.5.2 Deadlocks

- Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither T_3 nor T_4 can make progress
  – executing lock-S(B) causes T_4 to wait for T_3 to release its lock on ,
  – executing lock-X(A) causes T_3 to wait for T_4 to release its lock on A
- Such a situation is called a deadlock
  – to handle the problem, one of T_3 or T_4 must be rolled back and its locks released

Deadlocks & Starvation

- Two-phase locking protocol
  – guarantees serializability
  – does not ensure freedom from deadlocks
- In addition to deadlocks, there is a possibility of starvation:
  – A transaction may be waiting for an X-lock on an item
  – while a sequence of other transactions request and are granted an S-lock on the same item
- Starvation occurs if the concurrency control manager is badly designed
  – The same transaction is repeatedly rolled back due to deadlocks
  – Concurrency control manager can be designed to prevent starvation

Deadlocks

- The potential for deadlock exists in most locking protocols
  – but there are prevention mechanisms
- When a deadlock occurs
  – rollbacks are necessary
  – there is a possibility of cascading roll-backs
- but cascading rollbacks can be expensive
- Cascading roll-back is possible under two-phase locking
- Modified protocol called strict two-phase locking
  – a transaction must hold all its exclusive locks until it commits/aborts
  – avoids cascading rollbacks

Unit 4

Further topics

Syllabus

- 4.1 Database Systems and the Internet
  - 4.1.1 The architecture of a search engine
  - 4.1.2 Identifying Important pages
- 4.2 Specialty databases
  - 4.2.1 Object-Oriented Database
  - 4.2.2 Logic-based database
  - 4.2.3 Geographic database

4.1 Database Systems and the Internet

- Web database connectivity allows new innovative services that:
  - Permit rapid response by bringing new services and products to market quickly
  - Increase customer satisfaction through creation of Web- based support services
  - Yield fast and effective information dissemination through universal access

Web-to-Database Middleware

- Web server is the main hub through which Internet services are accessed.
- Dynamic Web pages are the heart of current generation Web sites
- Server-side extension: a program that interacts directly with the Web server
  - Also know as Web-to-database middleware
- Middleware must be well integrated

Web Server Interfaces

- Two well-defined Web server interfaces:
  - Common Gateway Interface (CGI)
  - Application Programming Interface (API)
- Disadvantage of CGI scripts:
  - Loading external script decreases system performance
  - Language and method used to create script also decrease performance
- API is more efficient than CGI
  - API is treated as part of Web server program

Web Application Servers

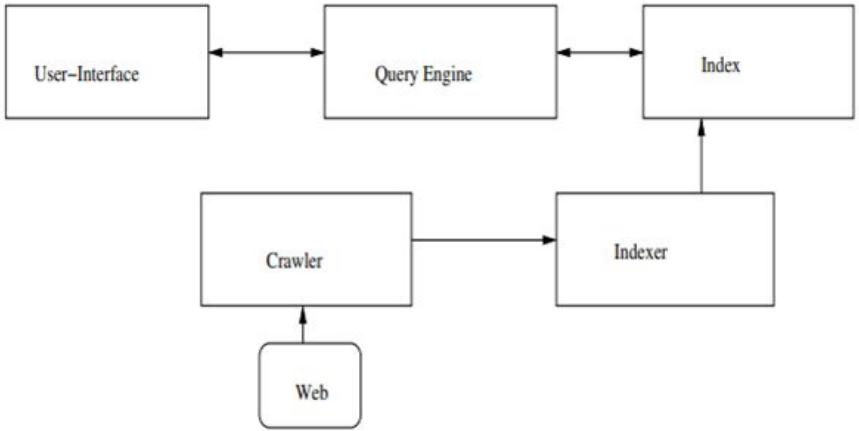
- Middleware application that expands the functionality of Web servers
  - Links them to a wide range of services
- Some uses of Web application servers:
  - Connect to and query database from Web page
  - Create dynamic Web search pages
  - Enforce referential integrity
- Some features of Web application servers:
  - Security and user authentication
  - Access to multiple services

4.1.1 The architecture of a search engine

- It consists of its software components, the interfaces provided by them, and the relationships between any two of them. (An extra level of detail could include the data structures supported.)
- The first search engines such as Excite (1994), InfoSeek (1994), Altavista(1995) employed primarily Information Retrieval principles and techniques and were search engines that were evaluating the similarity of a query relative to the web document of a corpus of web-documents retrieved from the Web.
- This determined a “rank” of document for a query.

Criteria

- Any search engine architecture must satisfy two major criteria.
  - Effectiveness (Quality) that will satisfy the relevance criterion.
  - Efficiency (Speed) that will satisfy response times and throughput requirements i.e. process as many queries as quickly as possible. Related to it is the notion of scalability.



4.1.2 Identifying Important pages

- Crawlers are programs (e.g. software agents) that traverse the Web in a methodical and automated manner sending new or updated pages to a repository for post processing. Crawlers are also referred to as robots, bots, spiders or harvesters.
- Crawler Policies. A Web crawler traverses the web according to a set of policies that include
  (a) a selection policy,
  (b) a visit policy,
  (c) an observance policy, and
  (d) a parallelization/coordination policy

Crawling

- A simple way to crawl the web is to (periodically) give the crawler program a list of URLs to visit.
- This information is provided by a URL server.
    - The crawler can then expand on additional URLs reached from that initial search.
    - This is the approach Google followed back in 1998.
- Another alternative is to stick to the list of URLs provided by the URL server; if some of the supplied URLs correspond to pages with links in them, these newly encountered links are not crawled but only sent back to the URL server: the latter decides whether it is a new link or not, and also determines the time to visit the link.
- Another alternative is to start with the most popular URLs; one more is to initiate a visit based on exhaustive search of URLs or more formally of IP addresses.
    - However the considerations of the previous pages and in particular Step 1, make such approaches too ineffective unless. they are only used the first time or times a crawl is undertaken from scratch!

Metadata

- Information collected.
- Additional work may be performed on every page crawled. The crawler might collect and maintain additional metainformation about the crawled URL. This metainformation or metadata might include the size of the document, date/time information related to the crawling, date/time information of modification (write) of this document by its user creator, a hash (signature) of the document, etc.
- A web-page first crawled might be assigned a unique ID usually called docID for document id. Subsequent crawls that retrieve the same or newer versions of the document may not change the docID of that URL

4.2 Specialty databases

- One way to classify databases involves the type of their contents, for example: bibliographic, document-text, statistical, or multimedia objects.
- Another way is by their application area, for example: accounting, music compositions, movies, banking, manufacturing, or insurance.
- A third way is by some technical aspect, such as the database structure or interface type.
- Following lists a few of the adjectives used to characterize different kinds of databases:

Types of databases (1)

- An in-memory database is a database that primarily resides in main memory, but is typically backed-up by non-volatile computer data storage.
- An active database includes an event-driven architecture which can respond to conditions both inside and outside the database.
- A cloud database relies on cloud technology.
- Data warehouses archive data from operational databases and often from external sources. Some basic and essential components of data warehousing include extracting, analyzing, and mining data, transforming, loading, and managing data so as to make them available for further use.
- A deductive database combines logic programming with a relational database.

Types of databases (2)

- A distributed database is one in which both the data and the DBMS span multiple computers.
- A document-oriented database is designed for storing, retrieving, and managing document-oriented, or semi structured, information. Document-oriented databases are one of the main categories of NoSQL databases.
- An embedded database system is a DBMS which is tightly integrated with an application software.
- A graph database is a kind of NoSQL database that uses graph structures with nodes, edges, and properties to represent and store information.

Types of databases (3)

- A knowledge base is a special kind of database for knowledge management, providing the means for the computerized collection, organization, and retrieval of knowledge.
- A mobile database can be carried on or synchronized from a mobile computing device.
- Operational databases store detailed data about the operations of an organization.
- A parallel database seeks to improve performance through parallelization for tasks such as loading data, building indexes and evaluating queries.
- Probabilistic databases employ fuzzy logic to draw inferences from imprecise data.

Types of databases (4)

- Real-time databases process transactions fast enough for the result to come back and be acted on right away.
- A spatial database can store the data with multidimensional features.
- A temporal database has built-in time aspects, for example a temporal data model and a temporal version of SQL. More specifically the temporal aspects usually include valid-time and transaction-time.
- An unstructured data database is intended to store in a manageable and protected way diverse objects that do not fit naturally and conveniently in common databases. It may include email messages, documents, journals, multimedia objects, etc.

4.2.1 Object-Oriented Database

- An object database is a database management system in which information is represented in the form of objects as used in object-oriented programming.
- Object databases are different from relational databases which are table-oriented. Object-relational databases are a hybrid of both approaches.
- Object databases have been considered since the early 1980s.
- Because the database is integrated with the programming language, the programmer can maintain consistency within one environment, in that both the OODBMS and the programming language will use the same model of representation. Some object-oriented databases are designed to work well with object-oriented programming languages such as Delphi, Ruby, Python, JavaScript, Perl, Java, C#, Visual Basic .NET, C++, Objective-C and Smalltalk.

Features

- Most object databases also offer some kind of query language, allowing objects to be found using a declarative programming approach.
- An attempt at standardization was made by the ODMG with the Object Query Language, OQL.
- Access to data can be faster because an object can be retrieved directly without a search, by following pointers.
- Multimedia applications are facilitated because the class methods associated with the data are responsible for its correct interpretation.
- Many object databases, for example Gemstone or VOSS, offer support for versioning. An object can be viewed as the set of all its versions.
- Some object databases also provide systematic support for triggers and constraints which are the basis of active databases.

Products

- Commercial products included Gemstone (Servio Logic, name changed to GemStone Systems), Gbase (Graphael), and Vbase (Ontologic).
- The early to mid-1990s saw additional commercial products enter the market. These included ITASCA (Itasca Systems), Jasmine (Fujitsu, marketed by Computer Associates), Matisse (Matisse Software), Objectivity/DB (Objectivity, Inc.), ObjectStore (Progress Software), ONTOS (Ontos, Inc., name changed from Ontologic), O2 (O2 Technology), POET (now FastObjects from Versant which acquired Poet Software), Versant Object Database (Versant Corporation), VOSS (Logic Arts) and JADE (Jade Software Corporation).
- Some of these products remain on the market and have been joined by new open source and commercial products such as InterSystems Caché.

4.2.2 Logic-based database

- A deductive database is a database system that can make deductions (i.e., conclude additional facts) based on rules and facts stored in the (deductive) database.
- Datalog is the language typically used to specify facts, rules and queries in deductive databases.
- Deductive databases have grown out of the desire to combine logic programming with relational databases to construct systems that support a powerful formalism and are still fast and able to deal with very large datasets.
- In recent years, deductive databases such as Datalog have found new application in data integration, information extraction, networking, program analysis, security, and cloud computing

Facts and rules

- A deductive database uses two main types of specifications: facts and rules
  - Facts are specified in a manner similar to the way relations are specified, except that it is not necessary to include the attribute names.
    - In a deductive database, the meaning of an attribute value in a tuple is determined solely by its position within the tuple.
  - Rules are somewhat similar to relational views. They specify virtual relations that are not actually stored but that can be formed from the facts by applying inference mechanisms based on the rule specifications.
  - The main difference between rules and views is that rules may involve recursion and hence may yield virtual relations that cannot be defined in terms of basic relational views.

4.2.3 Geographic database

- A geodatabase (also geographical database and geospatial database) is a database of geographic data, such as countries, administrative divisions, cities, and related information.
- Such databases can be useful for websites that wish to identify the locations of their visitors for customization purposes.
- A spatial database is a database that is optimized for storing and querying data that represents objects defined in a geometric space.
- Most spatial databases allow the representation of simple geometric objects such as points, lines and polygons.

Features

- Spatial databases use a spatial index to speed up database operations.
- In addition to typical SQL queries such as SELECT statements, spatial databases can perform a wide variety of spatial operations. The following operations and many more are specified by the Open Geospatial Consortium standard:
    - Spatial Measurements: Computes line length, polygon area, the distance between geometries, etc.
    - Spatial Functions: Modify existing features to create new ones, for example by providing a buffer around them, intersecting features, etc.
    - Spatial Predicates: Allows true/false queries about spatial relationships between geometries.
    - Geometry Constructors: Creates new geometries, usually by specifying the vertices (points or nodes) which define the shape.
    - Observer Functions: Queries which return specific information about a feature such as the location of the center of a circle

Spatial database systems

- AllegroGraph – a graph database which provides a mechanism for efficient storage and retrieval of two-dimensional geospatial coordinates for Resource Description Framework data.
- CouchDB a document-based database system that can be spatially enabled by a plugin called Geocouch
- Esri has a number of both single-user and multiuser geodatabases.
- GeoMesa is a cloud-based spatio-temporal database built on top of Apache Accumulo and Apache Hadoop. Supports full OGC Simple Features and a GeoServer plugin.
- IBM DB2 Spatial Extender can spatially-enable any edition of DB2, including the free DB2 Express-C, with support for spatial types

Spatial database systems

- Microsoft SQL Server has support for spatial types since version 2008
- MySQL DBMS implements the datatype geometry, plus some spatial functions implemented according to the OpenGIS specifications.
- OpenLink Virtuoso has supported SQL/MM since version 6.01.3126, with significant enhancements including GeoSPARQL in Open Source Edition 7.2.6, and in Enterprise Edition 8.2.0
- Oracle Spatial
- PostgreSQL DBMS uses the spatial extension PostGIS to implement the standardized datatype geometry and corresponding functions.
- Redis with the Geo API.