



Golang Programming

Working with SQL and NoSQL databases in Go

Where to Find The Code and Materials?

<https://github.com/iproduct/coursego>

Introduction to Databases and Distributed Data Processing



Databases, DBMSs and DB Models

- **Database** - an organized collection of data, generally stored and accessed electronically from a computer system. Can be developed using formal design and modeling techniques.
- **DataBase Management System (DBMS)** – software that interacts with end users, applications, and the database to capture and analyze the data, providing core facilities to create and administer databases.
- DBMSs can be classified according to the **database models** that they support:
 - In 1980s **relational databases** became dominant, modelling data as rows and columns in a series of tables, and the vast majority use **Structured Query Language (SQL)** for writing and querying data.
 - In the 2000s, **non-relational databases** became popular, referred to as **NoSQL** because they use **different query languages**.

Relational Databases

- “Relational database” term – invented by E. F. Codd at IBM in 1970, paper: "A Relational Model of Data for Large Shared Data Banks".
- Present the data to the user as **relations** (a presentation in **tabular form**, i.e. as a **collection of tables** with each table consisting of a set of **rows** and **columns**)
- Provide **relational operators** to manipulate the data in tabular form.
- As of 2009, most **commercial relational DBMSs** employ **SQL** as their query language.

Examples: **Oracle**, **MySQL**, **Microsoft SQL Server**, **PostgreSQL**, **IBM DB2**, **SQLite**

```
dvdrental=# select title, release_year, length, replacement_cost from film
dvdrental=#   where length > 120 and replacement_cost > 29.50
dvdrental=#   order by title desc;
```

title	release_year	length	replacement_cost
West Lion	2006	159	29.99
Virgin Daisy	2006	179	29.99
Uncut Suicides	2006	172	29.99
Tracy Cider	2006	142	29.99
Song Hedwig	2006	165	29.99
Slacker Liaisons	2006	179	29.99
Sassy Packer	2006	154	29.99
River Outlaw	2006	149	29.99
Right Cranes	2006	153	29.99
Quest Mussolini	2006	177	29.99
Poseidon Forever	2006	159	29.99
Loathing Legally	2006	140	29.99
Lawless Vision	2006	181	29.99
Jingle Sagebrush	2006	124	29.99
Jericho Mulan	2006	171	29.99
Japanese Run	2006	135	29.99
Gilmore Boiled	2006	163	29.99
Floats Garden	2006	145	29.99
Fantasia Park	2006	131	29.99
Extraordinary Conquerer	2006	122	29.99
Everyone Craft	2006	163	29.99
Dirty Ace	2006	147	29.99
Clyde Theory	2006	139	29.99
Clockwork Paradise	2006	143	29.99
Ballroom Mockingbird	2006	173	29.99

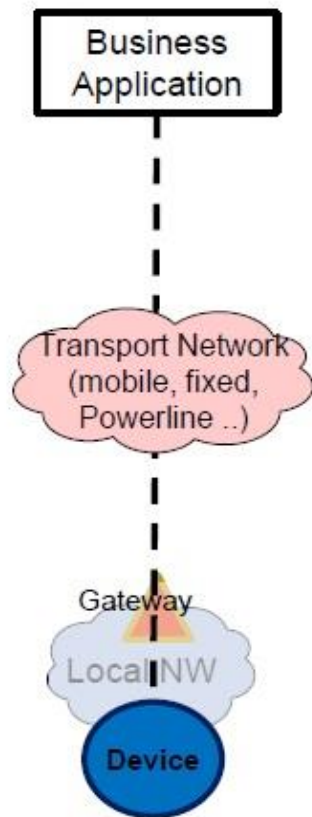
(25 rows)

NoSQL and NewSQL Databases

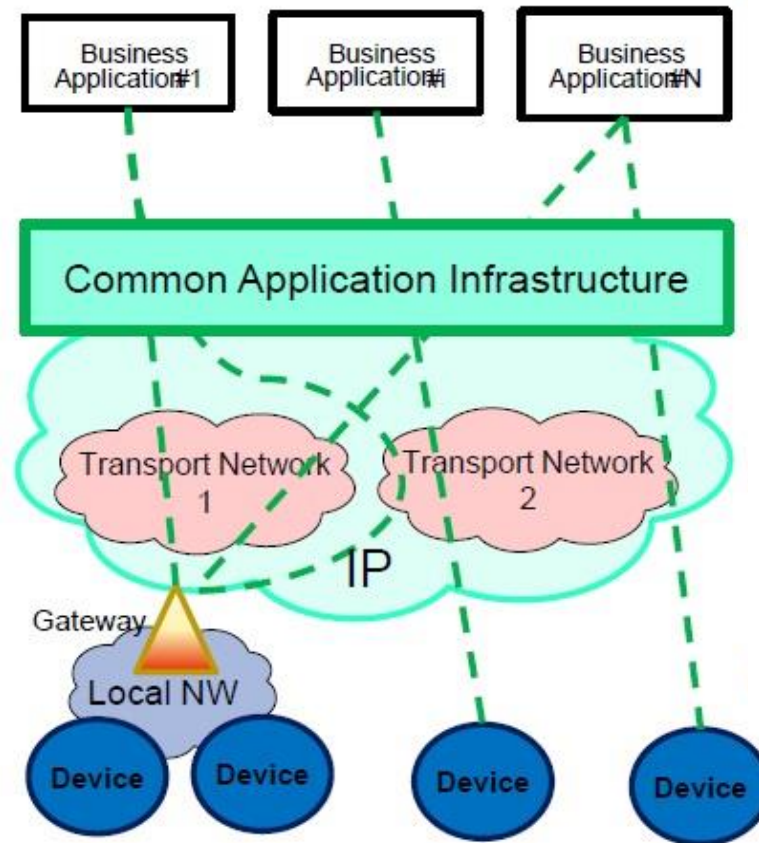
- **NoSQL databases** – massively distributed, horizontally scalable, fast, do not require fixed table schemas, avoid join operations by storing denormalized data.
- **CAP theorem**: it is impossible for a distributed system to simultaneously provide consistency, availability, and partition tolerance guarantees → eventual consistency = high availability and partition tolerance with a reduced level of data consistency.
- **NewSQL** is a class of modern relational databases that aims to provide the same scalable performance of NoSQL systems for online transaction processing (read-write) workloads while still using SQL and maintaining the ACID guarantees of a traditional database system.

Vertical vs. Horizontal Scaling

Pipe (vertical):
1 Application, 1 NW,
1 (or few) type of Device



Horizontal (based on common Layer)
Applications share common infrastructure, environments
and network elements



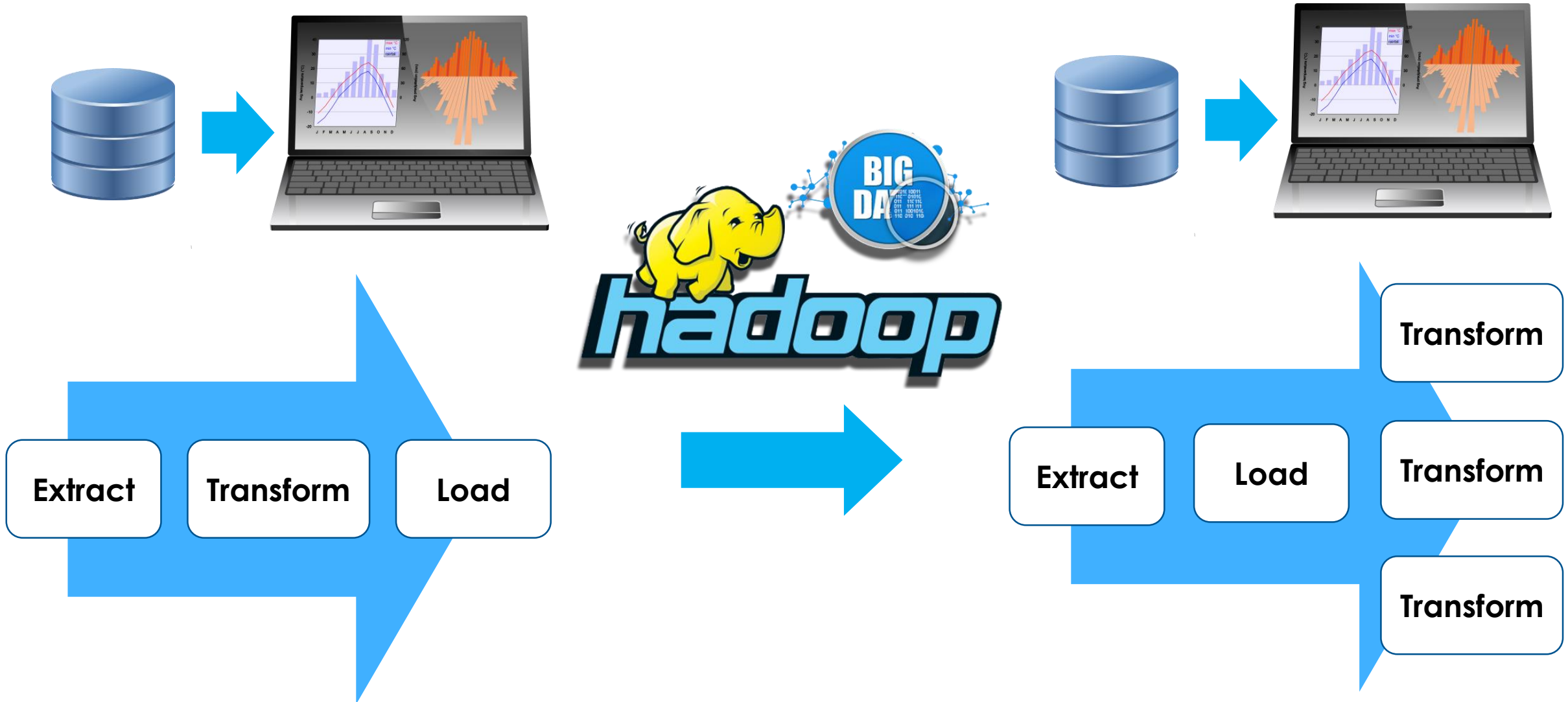
NoSQL and NewSQL Database Examples

Type	Notable examples of this type
<u>Wide column</u>	<u>Accumulo</u> , <u>Cassandra</u> , <u>Scylla</u> , <u>HBase</u>
<u>Document:</u>	<u>Apache CouchDB</u> , <u>ArangoDB</u> , <u>BaseX</u> , <u>Clusterpoint</u> , <u>Couchbase</u> , <u>Cosmos DB</u> , <u>eXist-db</u> , <u>IBM Domino</u> , <u>MarkLogic</u> , <u>MongoDB</u> , <u>OrientDB</u> , <u>Qizx</u> , <u>RethinkDB</u>
<u>Key-value:</u>	<u>Aerospike</u> , <u>Apache Ignite</u> , <u>ArangoDB</u> , <u>Berkeley DB</u> , <u>Couchbase</u> , <u>Dynamo</u> , <u>FoundationDB</u> , <u>InfinityDB</u> , <u>MemcacheDB</u> , <u>MUMPS</u> , <u>Oracle NoSQL Database</u> , <u>OrientDB</u> , <u>Redis</u> , <u>Riak</u> , <u>SciDB</u> , <u>SDBM/Flat File dbm</u> , <u>ZooKeeper</u>
<u>Graph:</u>	<u>AllegroGraph</u> , <u>ArangoDB</u> , <u>InfiniteGraph</u> , <u>Apache Giraph</u> , <u>MarkLogic</u> , <u>Neo4J</u> , <u>OrientDB</u> , <u>Virtuoso</u>
<u>New SQL</u>	<u>CockroachDB</u> , <u>Citus</u> , <u>Vitess</u>

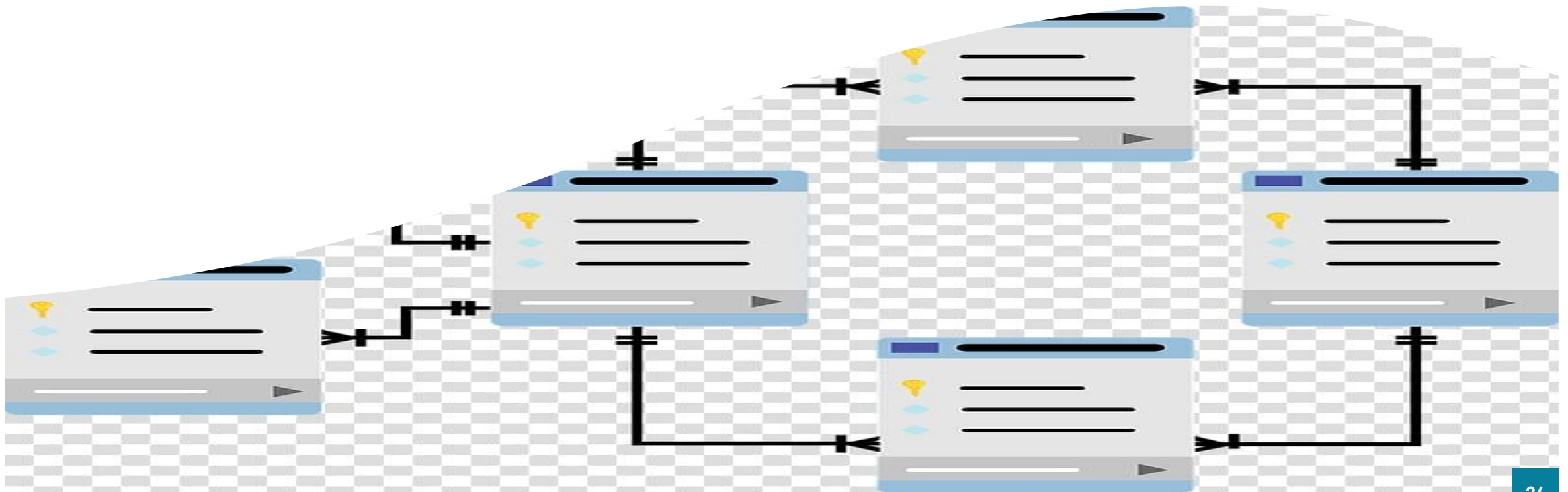
SQL and NoSQL Databases Comparison

Data model	Performance	Scalability	Flexibility	Complexity	Functionality
Key-value store	high	high	high	none	variable (none)
Column-oriented store	high	high	moderate	low	minimal
Document-oriented store	high	variable (high)	high	low	variable (low)
Graph database	variable	variable	high	high	graph theory
Relational database	variable	variable	low	moderate	relational algebra

Batch Processing

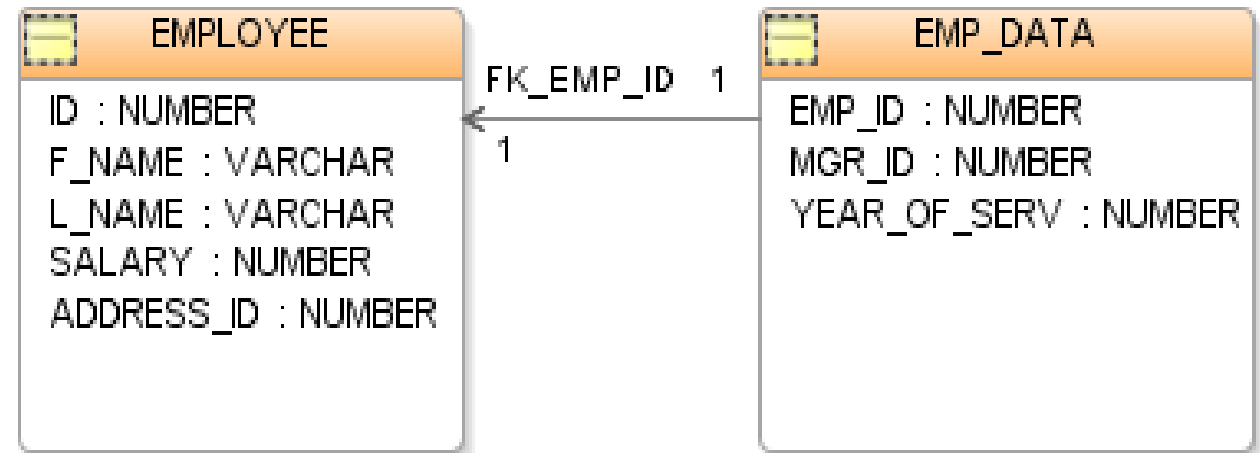
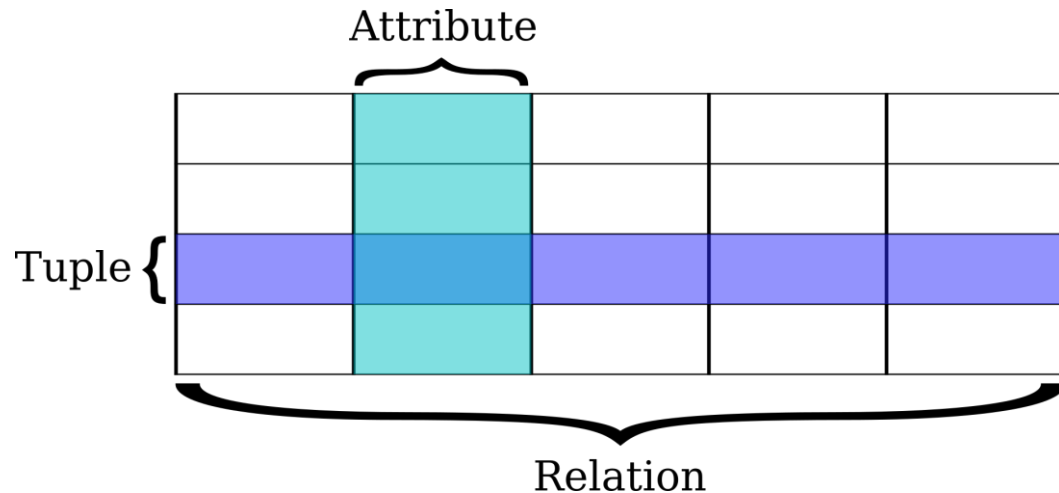


Relational Databases

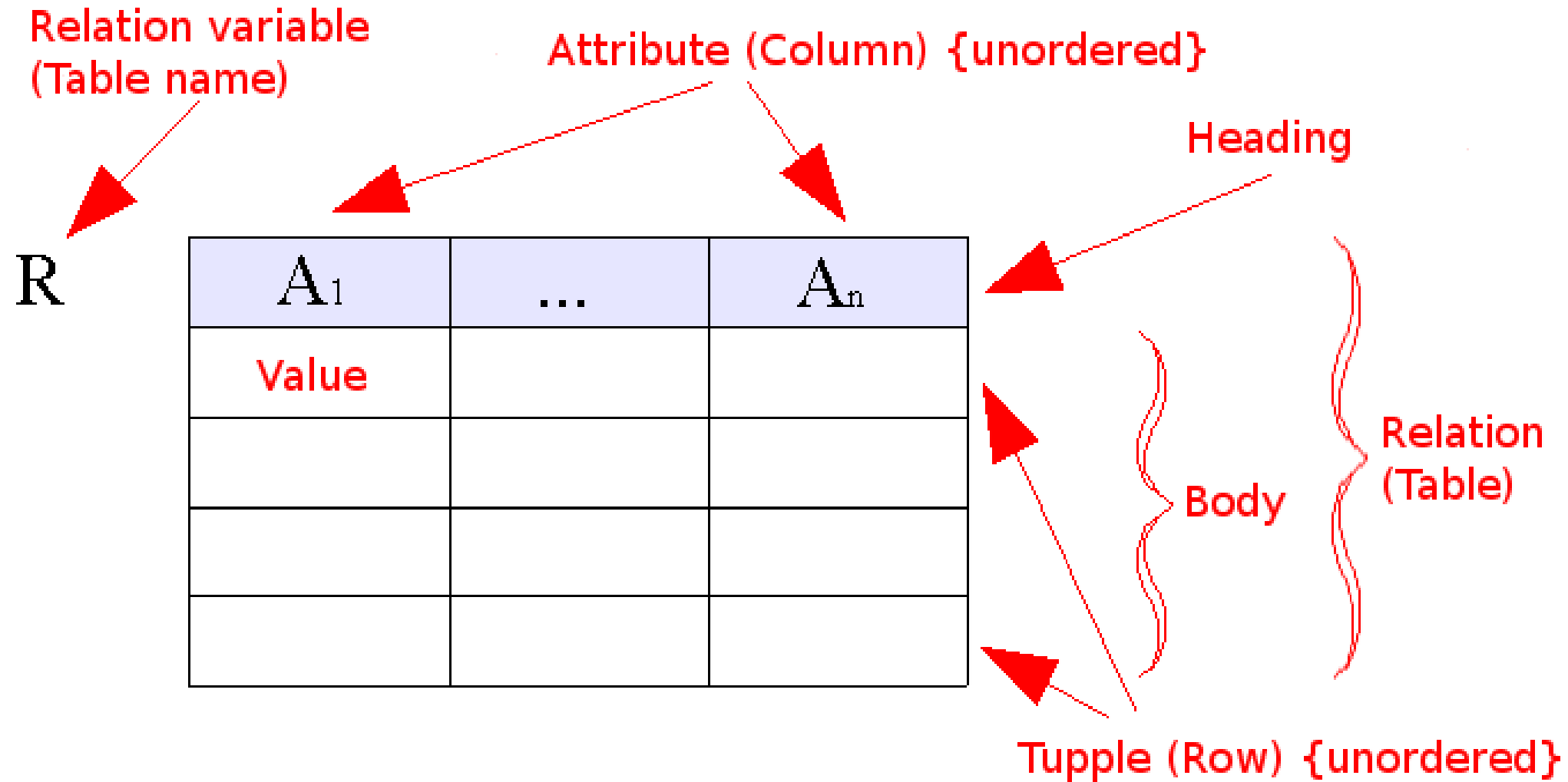


Relational Model

- relation \leftrightarrow table
- record, tuple \leftrightarrow row
- attribute \leftrightarrow column



Relational Model



Views. Domains. Constraints

- Def: Relations which store primary data are called **base relations** or **tables**. Other relations, which are derived from primary relations are **queries** and **views**.
- Def: **Domain** in database is a set of allowed values for a given attribute in a relation – an existing constraint about valid the type of values for given attribute.
- Def: **Constraints** allow more flexible specification of values that are valid for given attribute – e.g. from 1 to 10.

Keys

- **Key** consists of one or more attributes, such that:
 - 1) relation has no two records with the same values for these attributes
 - 2) there is no proper subset of these attributes with the same property
- **Primary Key** is an attribute (less frequently a group of attributes), which uniquely identifies each record (tuple) in the relation
- **Foreign key** is necessary when there exists a relation between two tables – usually it is an attribute in second table referring to primary key of the first table

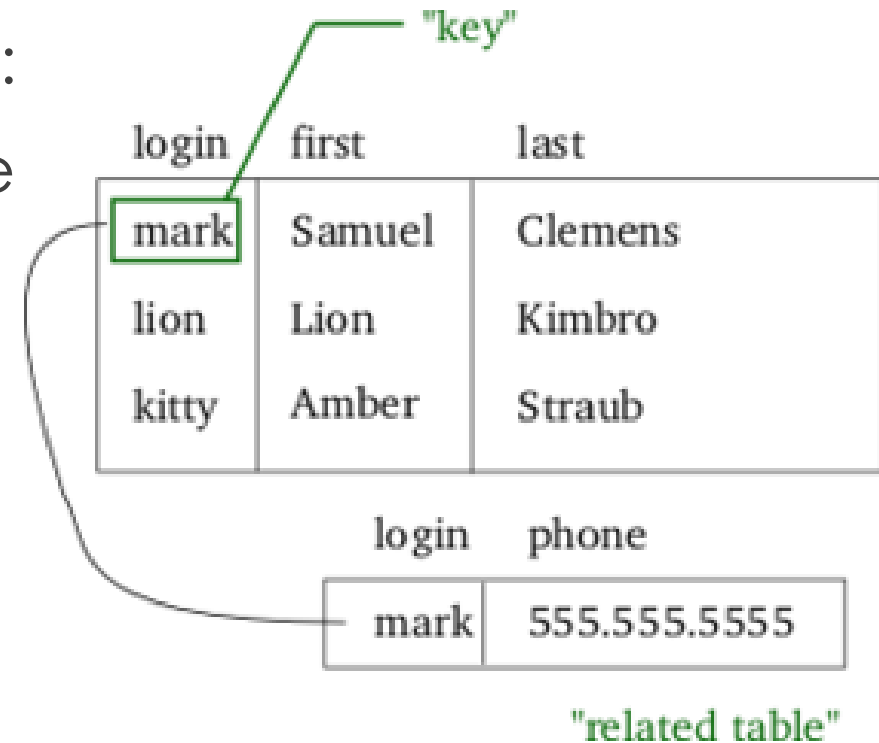
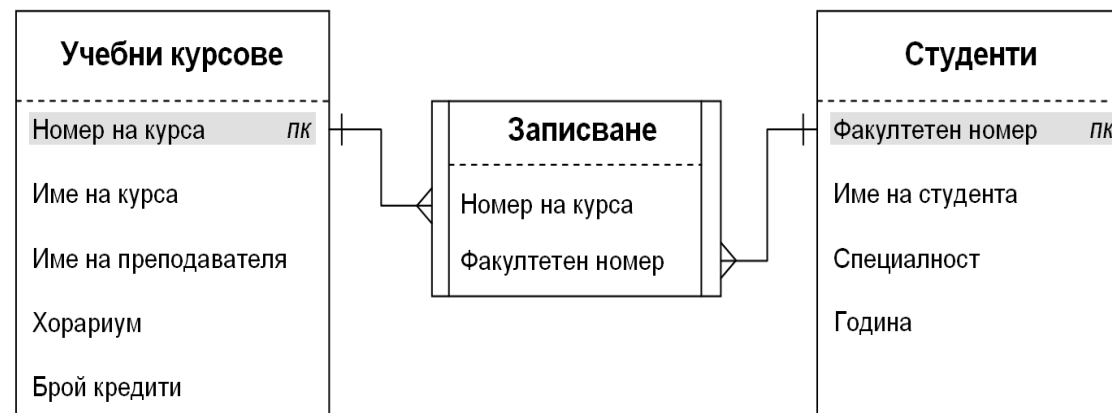
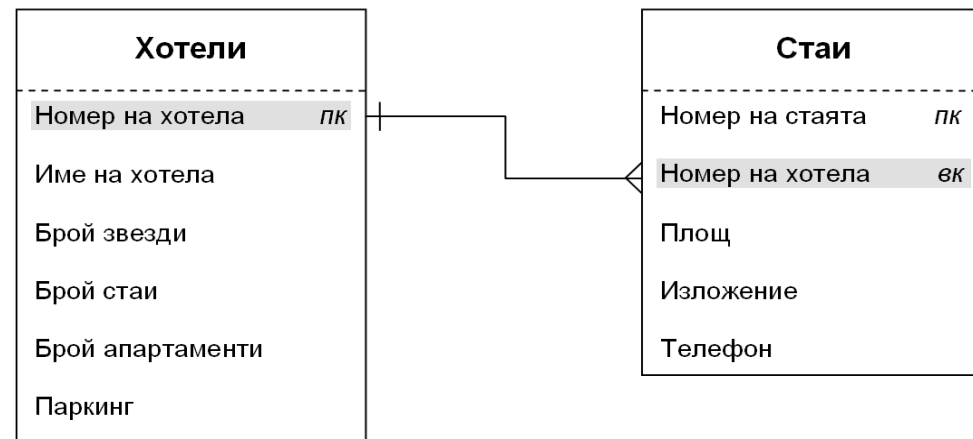
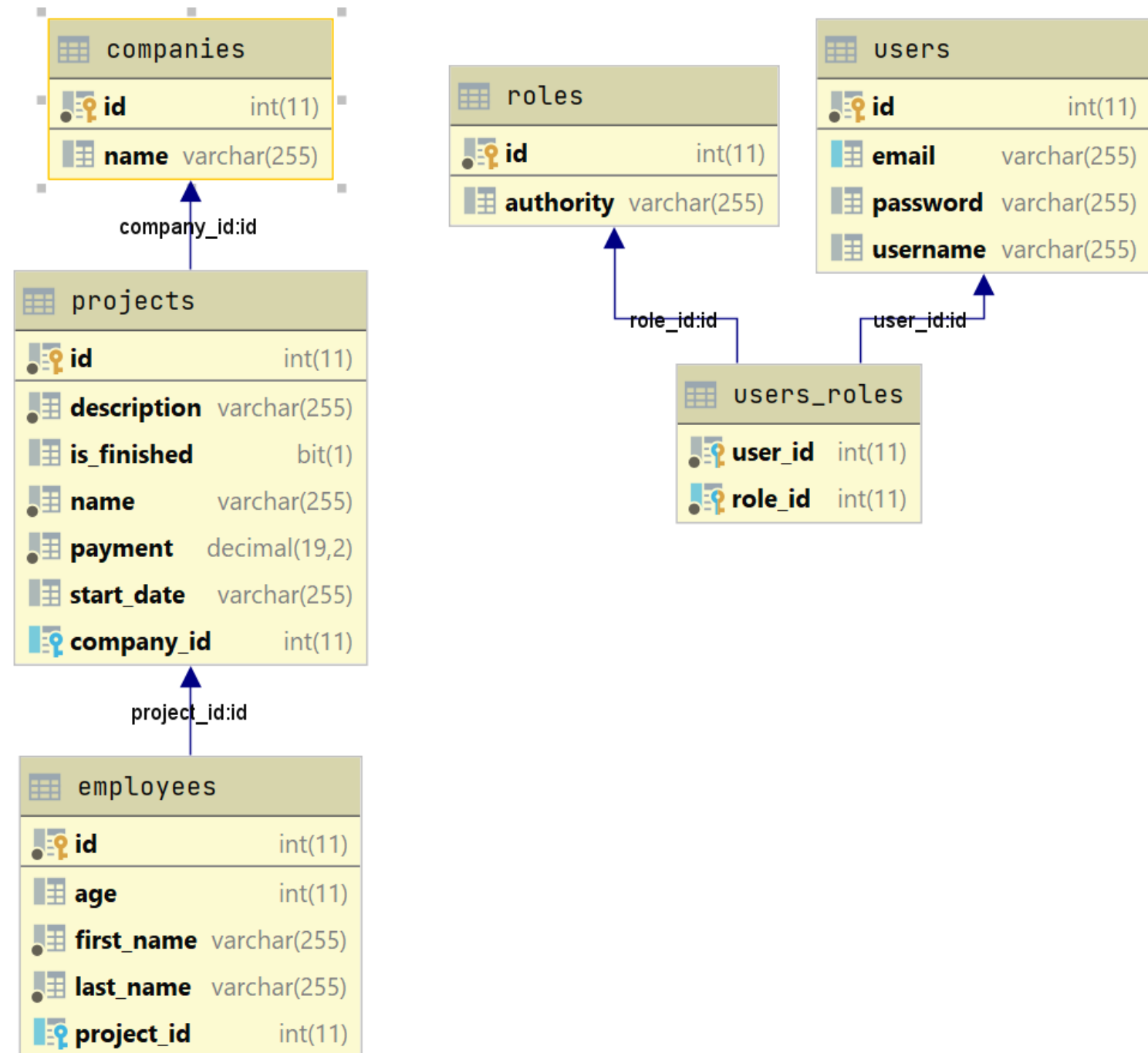


Table Relations. Cardinality

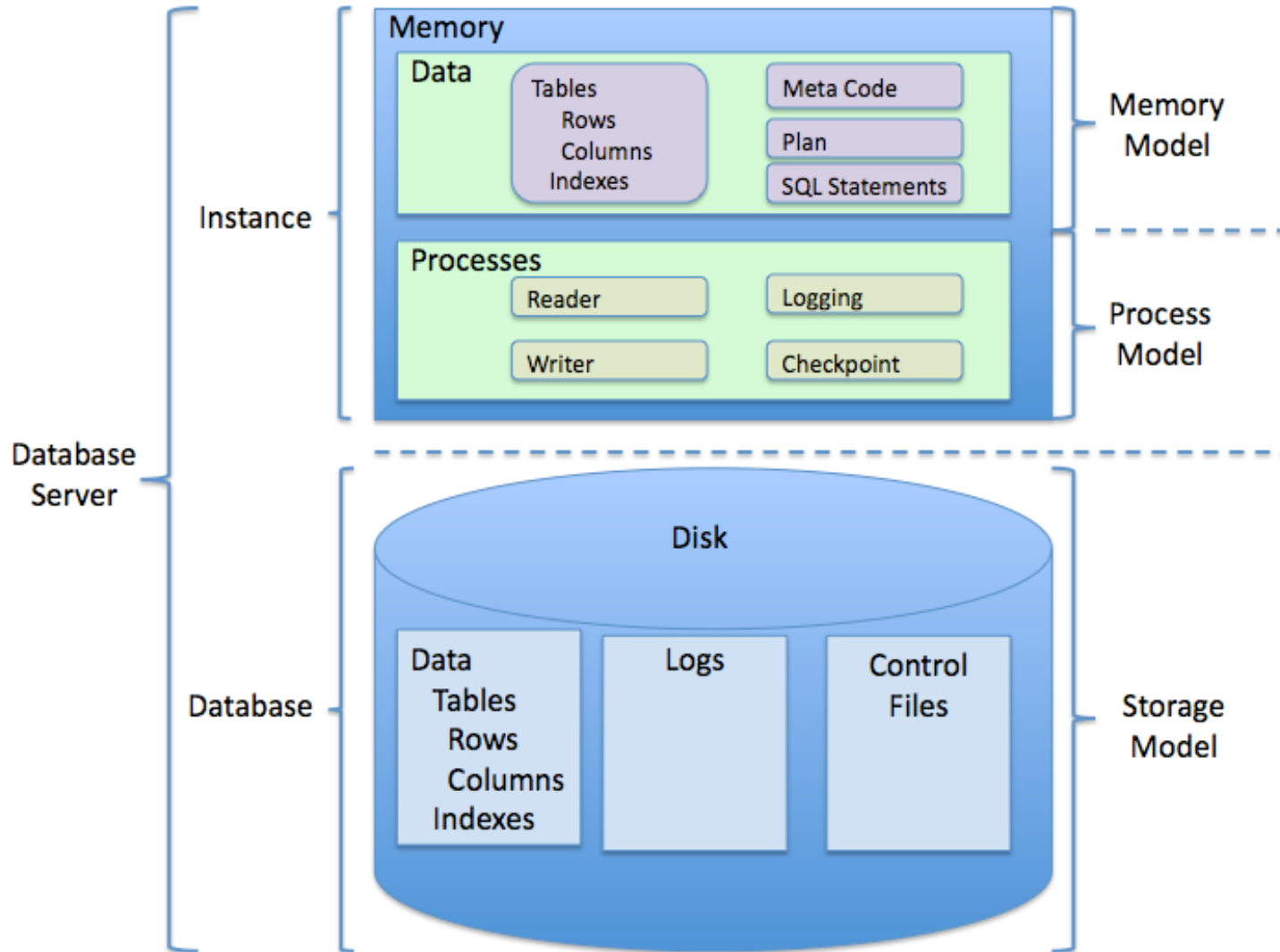
- **Relationship** is a dependency existing between two tables, when the records from first table can be connected somehow with records from second one.
- **Cardinality:**
 - One to one (1:1),
 - One to many (1:N),
 - Many to one (N:1)
 - Many to many (M:N)



Relational Schema Example ER Model



Relational Database Management System (RDBMS)

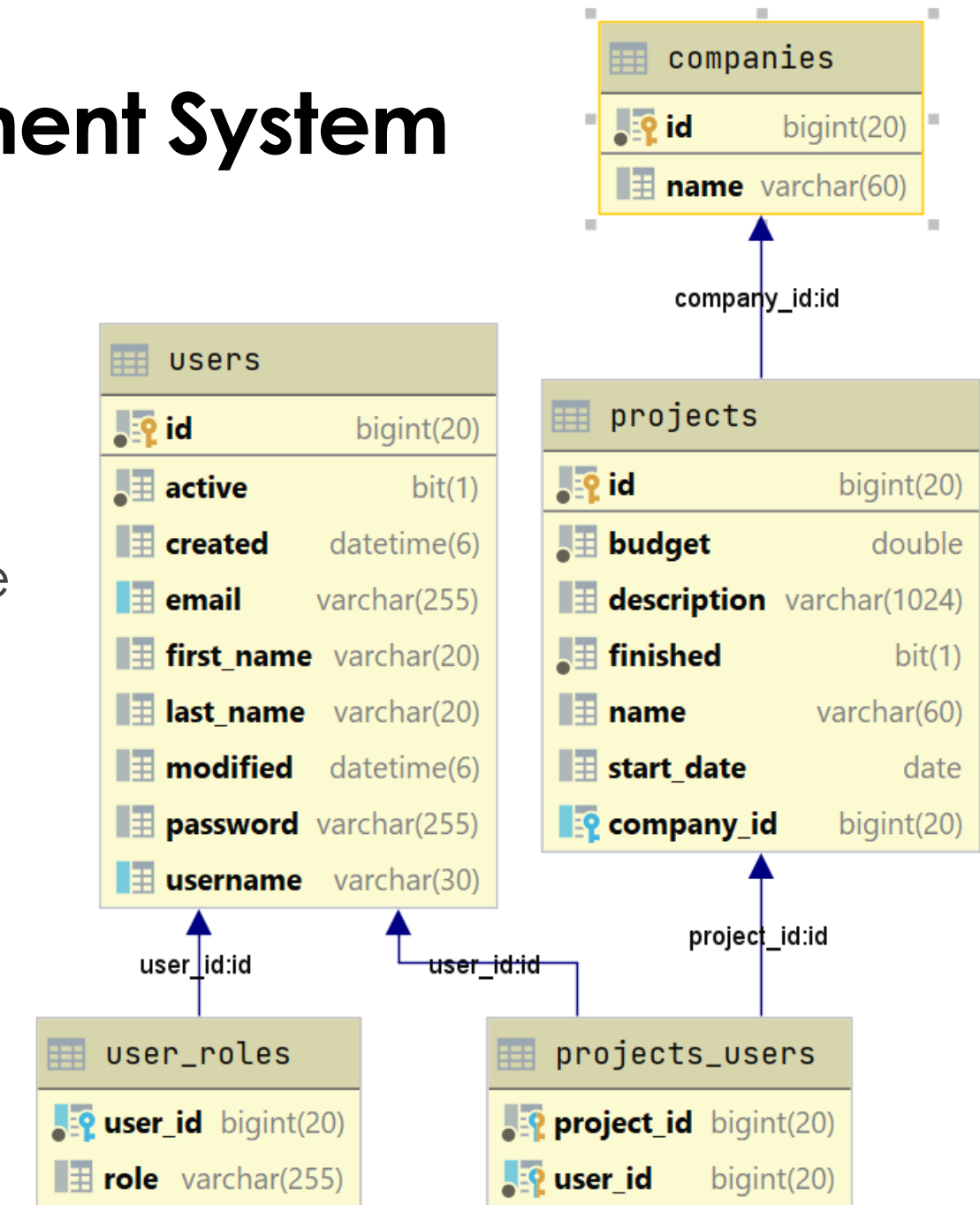


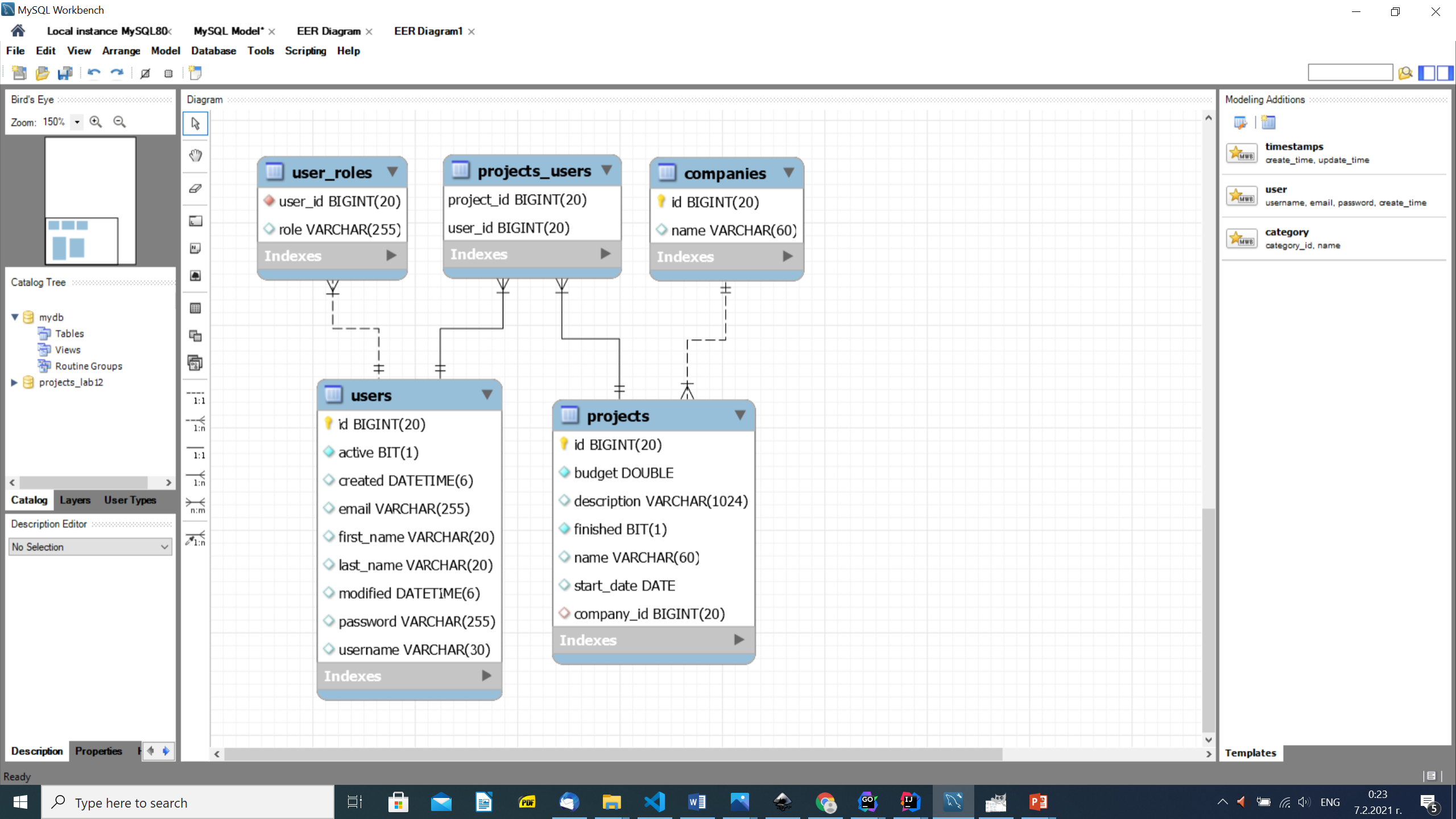
Accessing Relational Databases with Go – database/sql



Simple Project Management System

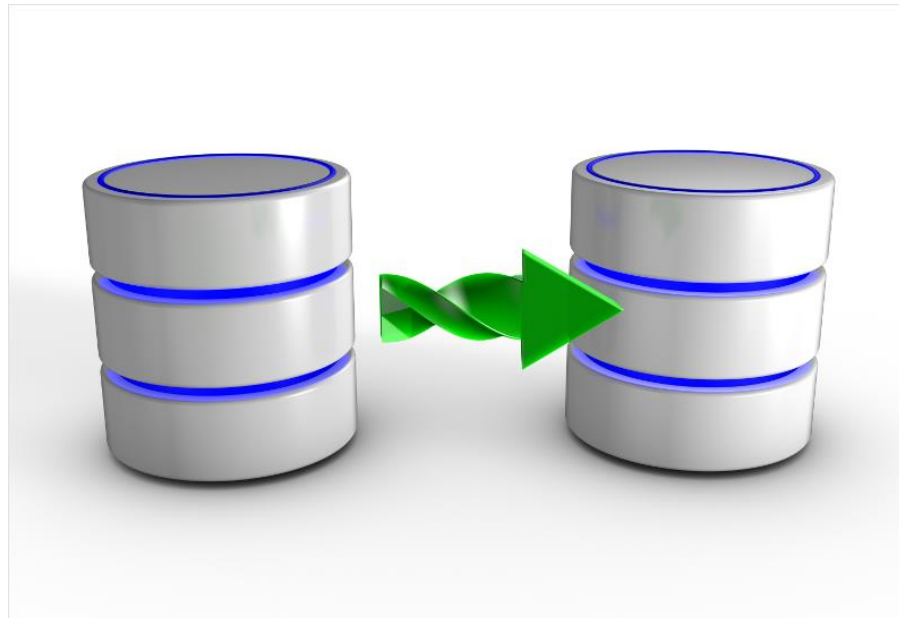
- **Companies** can develop multiple projects
- Each **project** can involve multiple **users** (employees)
- **Users** can have multiple **roles** in the project management system and can work on multiple **projects**
- **Users** should have unique **username** used for logging in the system together with a **password**





SQL Tutorials and Resources

- W3Schools SQL Tutorial – https://www.w3schools.com/sql/sql_select.asp
- MySQL 8.0 reference Manual – <https://dev.mysql.com/doc/refman/8.0/en/>
- MySQL Tutorial – <https://www.mysqltutorial.org/basic-mysql-tutorial.aspx>
- W3Resource MySQL Tutorial – <https://www.w3resource.com/mysql/mysql-tutorials.php>



database/sql: <https://golang.org/pkg/database/sql/>

- Create a connections pool to MySQL DB:

```
db, err := sql.Open("mysql", "root:root@/golang_projects?parseTime=true")
if err != nil {
    log.Fatal(err)
}
defer db.Close()
```

- Can add more settings to `*sql.DB`:

```
db.SetConnMaxLifetime(time.Minute * 5) // ensure connections are closed by the
//driver safely before MySQL server, OS, or other middlewares, helps load ballancing
db.SetMaxOpenConns(10) // maximum size of connection pool
db.SetMaxIdleConns(10) // maximum size of idel connections in the pool
db.SetConnMaxIdleTime(time.Minute * 3) // maximum time connection is kept if idle
```

database/sql: Ping the DB:

```
ctx, cancel := context.WithTimeout(context.Background(), 1*time.Second)
defer cancel()
```

```
status := "up"
if err := db.PingContext(ctx); err != nil {
    status = "down"
}
```

```
log.Println(status)
```

database/sql: Simple SELECT Query

```
func FindAllProjects(db *sql.DB) (projects []entities.Project, err error) {  
    rows, err := db.Query("SELECT * FROM projects")  
    if err != nil {  
        return  
    }  
    defer rows.Close()  
    for rows.Next() {  
        p := entities.Project{}  
        err = rows.Scan(&p.ID, &p.Name, &p.Description, &p.Budget, &p.Finished, &p.StartDate, &p.ComplID)  
        if err != nil {  
            return  
        }  
        projects = append(projects, p)  
    }  
    if err = rows.Err(); err != nil {  
        return  
    }  
    return  
}
```

database/sql: Better SELECT Query

```
func FindAllProjects(db *sql.DB) (projects []entities.Project, err error) {  
    rows, err := db.Query("SELECT * FROM projects")  
    if err != nil { return }  
    defer rows.Close()  
    for rows.Next() {  
        p := entities.Project{}  
        err = rows.Scan(&p.ID, &p.Name, &p.Description, &p.Budget, &p.Finished, &p.StartDate, &p.ComplID)  
        if err != nil { return }  
        projects = append(projects, p)  
    }  
    // If the database is being written to ensure to check for Close errors that may be returned from the driver. The  
    // query may encounter an auto-commit error and be forced to rollback changes.  
    err = rows.Close()  
    if err != nil {  
        return  
    }  
    // Rows.Err will report the last error encountered by Rows.Scan.  
    if err = rows.Err(); err != nil { return }  
    return  
}
```


database/sql: Prepared Statement – INSERT

```
stmt, err = db.Prepare(`INSERT INTO projects(name, description , budget, start_date, finished, company_id)
                        VALUES( ?, ?, ?, ?, ?, ? )`)
if err != nil { log.Fatal(err) }
defer stmt.Close() // Prepared statements take up server resources and should be closed after use.

for i, _ := range projects {
    projects[i].Finished = true
    result, err := stmt.Exec(projects[i].Name, projects[i].Description, projects[i].Budget, projects[i].StartDate,
        projects[i].Finished, projects[i].CompanyId);
    if err != nil { log.Fatal(err) }
    numRows, err := result.RowsAffected()
    if err != nil || numRows != 1 { log.Fatal("Error creating new Project", err) }
    insId, err := result.LastInsertId()
    if err != nil { log.Fatal(err) }
    projects[i].Id = insId
}
```

Transactions and Concurrency

- **Transaction** = **Business Event**
- **ACID rules:**
 - **Atomicity** – the whole transaction is completed (commit) or no part is completed at all (rollback).
 - **Consistency** – transaction should preserve existing integrity constraints
 - **Isolation** – two uncompleted transactions can not interact
 - **Durability** – successfully completed transactions can not be rolled back

Transaction Isolation Levels

- **DEFAULT** - use the default isolation level of the underlying datastore
- **READ_UNCOMMITTED** – dirty reads, non-repeatable reads and phantom reads can occur
- **READ_COMMITTED** – prevents dirty reads; non-repeatable reads and phantom reads can occur
- **REPEATABLE_READ** – prevents dirty reads and non-repeatable reads; phantom reads can occur
- **SERIALIZABLE** – prevents dirty reads, non-repeatable reads and phantom reads

database/sql: Update Project Budgets in Transaction

```
tx, err := conn.BeginTx(ctx, &sql.TxOptions{Isolation: sql.LevelSerializable}) // or db.BeginTx()
if err != nil { log.Fatal(err) }
result, execErr := tx.ExecContext(ctx, `UPDATE projects SET budget = ROUND(budget * 1.2)
                                         WHERE start_date > ?;`, startDate)

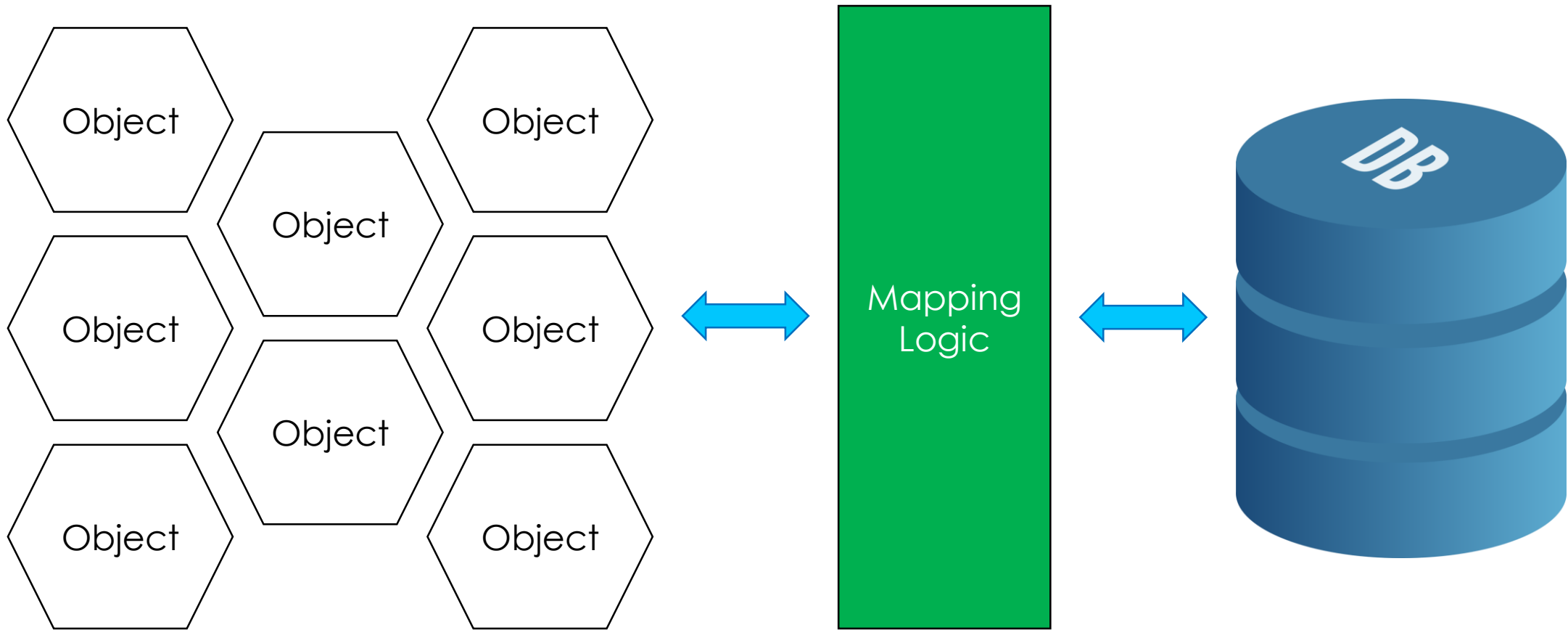
if execErr != nil {
    if rollbackErr := tx.Rollback(); rollbackErr != nil {
        log.Fatalf("update failed: %v, unable to rollback: %v\n", execErr, rollbackErr)
    }
    log.Fatalf("update failed: %v", execErr)
}
rows, err := result.RowsAffected()
if err != nil { log.Fatal(err) }
log.Printf("Total budgets updated: %d\n", rows)

if err := tx.Commit(); err != nil {
    log.Fatal(err)
}
```

Go Object to Relation Mapping with GORM



Object To Relational Mapping (ORM)



gorm.io/gorm: <https://gorm.io/>

```
dsn := "root:root@tcp(127.0.0.1:3306)/gorm_projects?charset=utf8mb4&parseTime=True&loc=Local"
db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
if err != nil { log.Fatal(err) }
user := entities.User{FirstName: "Rob", LastName: "Pike2", Email: "pike2@golang.com", Username: "rob2",
    Password: "rob", Active: true, Model: gorm.Model{CreatedAt: time.Now(), UpdatedAt: time.Now()}}

result := db.Create(&user) // pass pointer of data to Create

if result.Error != nil {
    log.Fatal(result.Error) // returns error
}

fmt.Printf("New user created with ID: %d -> %+v\nRows affected: %d\n",
    user.ID, // returns inserted data's primary key
    user,
    result.RowsAffected, // returns inserted records count
)
```

gorm.io/gorm: Detailed Configuration

```
db, err := gorm.Open(mysql.New(mysql.Config{  
    DSN: "root:root@tcp(127.0.0.1:3306)/golang_projects?charset=utf8&parseTime=True&loc=Local",  
    DefaultStringSize: 256, // default size for string fields  
    DisableDatetimePrecision: true, // disable datetime precision, which not supported before MySQL 5.6  
    DontSupportRenameIndex: true, // drop & create when rename index (not supported before MySQL 5.7)  
    DontSupportRenameColumn: true, // `change` when rename column (not supported before MySQL 8)  
    SkipInitializeWithVersion: false, // auto configure based on currently MySQL version  
}), &gorm.Config{})
```

gorm.io/gorm: Batch User Creation

```
db.AutoMigrate(&entities.User{})           // Automatically create database schema – all the tables
db.AutoMigrate(&entities.Company{})
db.AutoMigrate(&entities.Project{})
users := []entities.User{
    {FirstName: "Linus", LastName: "Torvalds", Email: "linus@linux.com", Username: "linus", Password: "linus",
      Active: true, Model: gorm.Model{CreatedAt: time.Now(), UpdatedAt: time.Now()}},
    {FirstName: "Rob", LastName: "Pike", Email: "pike@golang.com", Username: "rob", Password: "rob",
      Active: true, Model: gorm.Model{CreatedAt: time.Now(), UpdatedAt: time.Now()}},
}

result := db.Create(&users) // pass pointer of data to Create
//db.CreateInBatches(users, 100) // batch size 100

if result.Error != nil { log.Fatal(result.Error) } // returns error

fmt.Printf("New users created with IDs: ")

for _, user := range users { fmt.Printf("%d, ", user.ID) }
```

gorm.io/gorm: ... and Query

```
result = db.Find(&users)    // Get all users - SELECT * FROM users;

if result.Error != nil {    // if returns error

    log.Fatal(result.Error)
}

fmt.Printf("Number of users: %d\n", result.RowsAffected) // returns found records count, equals `len(users)`

utils.PrintUsers(users)
```

gorm.io/gorm: Preloading (Prefetching) Associations

```
result = db.Preload(clause.Associations).Find(&companies) // SELECT * FROM companies fetching projects
if result.Error != nil {
    log.Fatal(result.Error) // returns error
}
fmt.Printf("Number of companies: %d\n", result.RowsAffected) // returns found records count, equals `len(users)`
utils.PrintCompanies(companies)
```

```
result = db.Preload(clause.Associations).Find(&projects) // SELECT * FROM users with associations
if result.Error != nil { log.Fatal(result.Error) } // returns error
err = db.Model(&(projects[0])).Association("Users").Find(&users) // Association mode
if err != nil { log.Fatal(result.Error) } // returns error
fmt.Printf("Users in Project '%s': %v\n", projects[0].Name, users)
```

```
db.Session(&gorm.Session{FullSaveAssociations: true}).Updates(&user) // Saving all associations
```

Common Pitfalls when Using RDBs with Go - I

- Deferring `rows.Close()` inside a loop → memory and connections
- Opening many dbobjects (`sql.Db`) → many TCP connections in `TIME_WAIT`
- Not doing `rows.Close()` when done → Run `rows.Close()` as soon as possible, you can run it later again (no problem). Chain `db.QueryRow()` & `.Scan()`
- Unnecessaey use of prepared statements → if concurrency is high, consider whether prepared statements are necessary → re-prepared on busy connections → should be used only if executed many times
- Too much `strconv` or casts → let conversions to `.Scan()`
- Custom error-handling and retry → database/sql should handle connection pooling, reconnecting, and retries

Common Pitfalls when Using RDBs with Go - II

- Don't forgetting to check errors after **rows.Next()** → **rows.Next()** can exit with error
- Using **db.Query()** for **non-SELECT** queries → iterating over a result set when there is no one, leaking connections.
- Don't assuming subsequent statements are executed on same connection → Two statements can run on different connections → to solve the problem execute all statements on a single transaction (**sql.Tx**).
- Don't mix **db** access while using a **tx** → **sql.Tx** is bound to transaction, **db** not
- Unexpected **NULL** → to scan for **NULL** use one of the **NullXXX** types provided by the **database/sql** package – e.g. [NullString](#)

Recommended Literature

- The Go Documentation - <https://golang.org/doc/>
- The Go Bible: Effective Go - https://golang.org/doc/effective_go.html
- David Chisnall, *The Go Programming Language Phrasebook*, Addison Wesley, 2012
- Alan A. A. Donovan, Brian W. Kernighan, *The Go Programming Language*, Addison Wesley, 2016
- Nathan Youngman, Roger Peppé, *Get Programming with Go*, Manning, 2018
- Naren Yellavula, *Building RESTful Web Services with Go*, Packt, 2017

Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>