



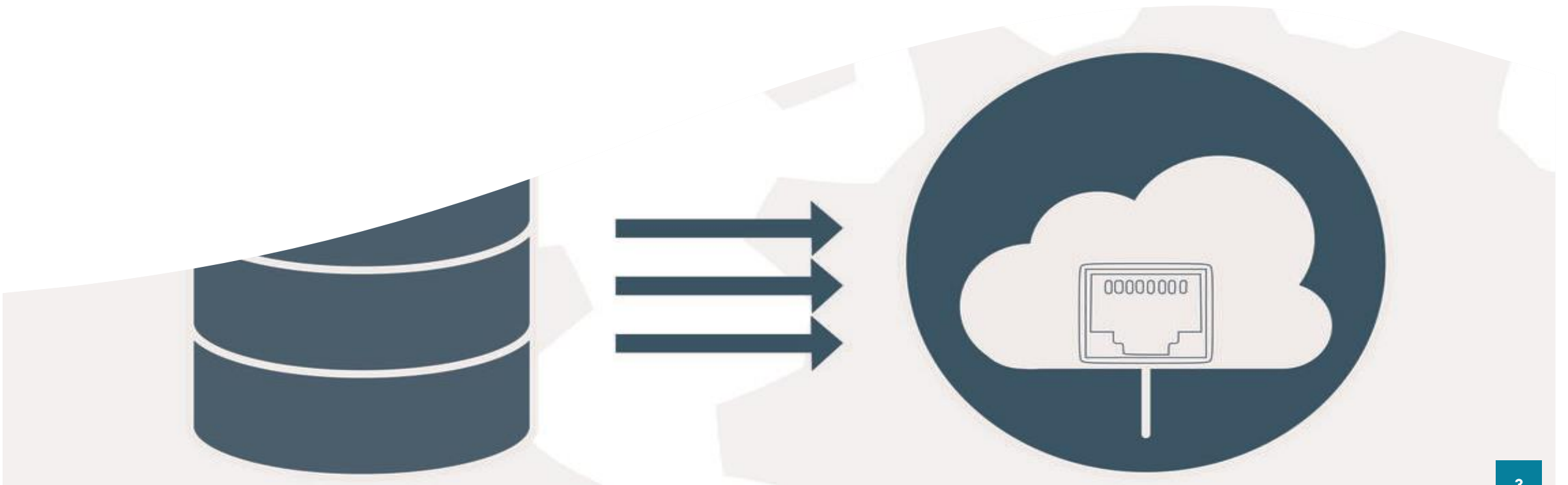
Golang Programming

Working with SQL and NoSQL databases

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

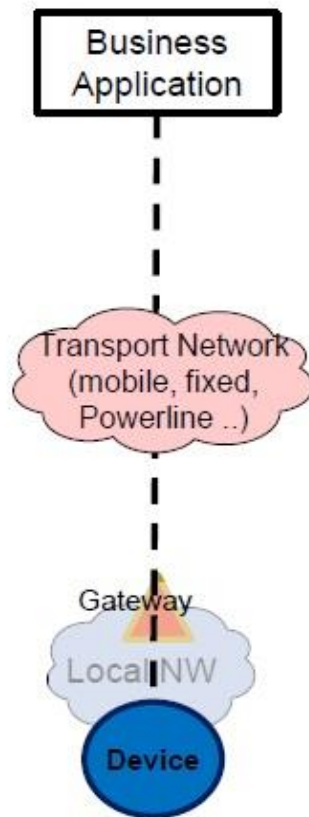
(25 rows)

NoSQL and NewSQL Databases

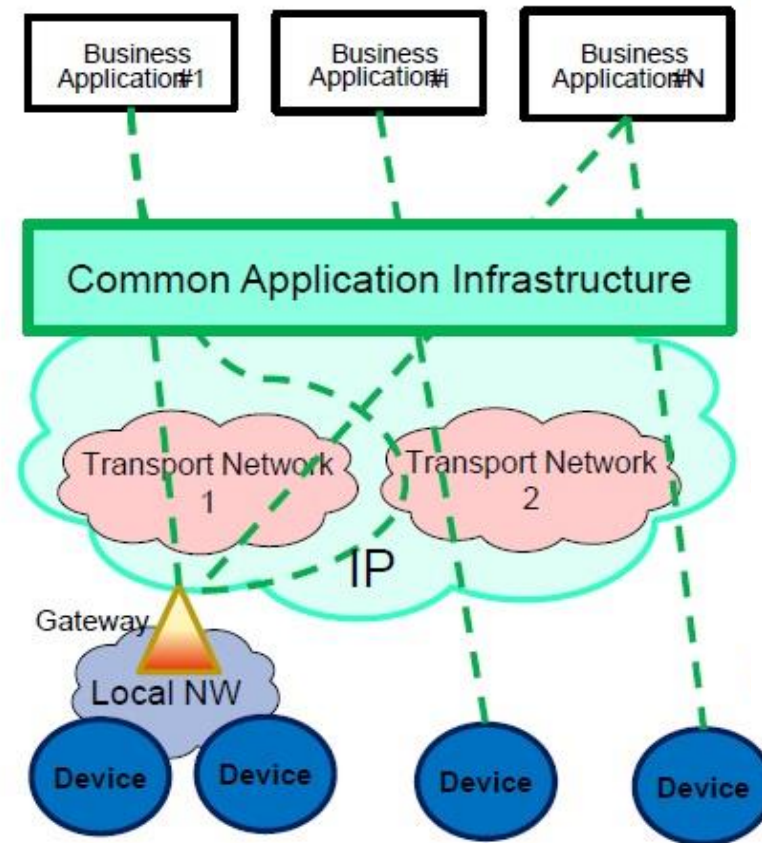
- **NoSQL databases** – massively distributed, horizontally scalable, fast, do not require fixed table schemas, avoid join operations by storing denormalized data.
- **CAP theorem**: it is impossible for a distributed system to simultaneously provide consistency, availability, and partition tolerance 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



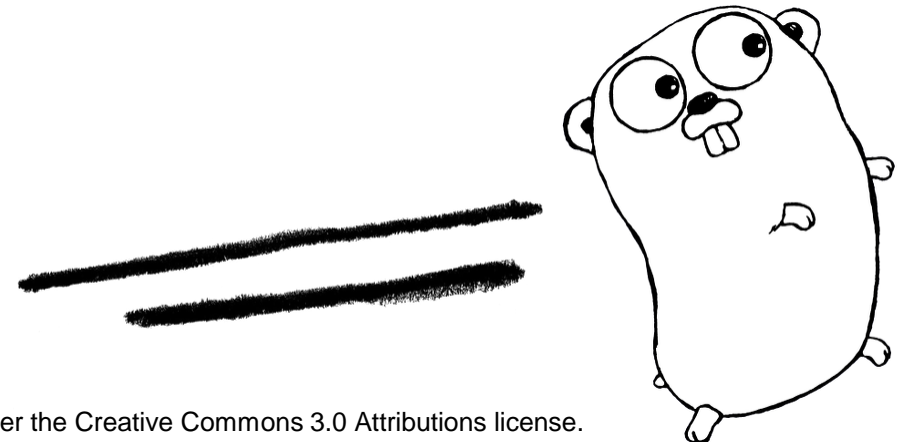
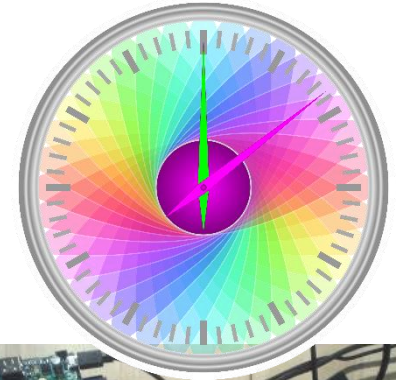
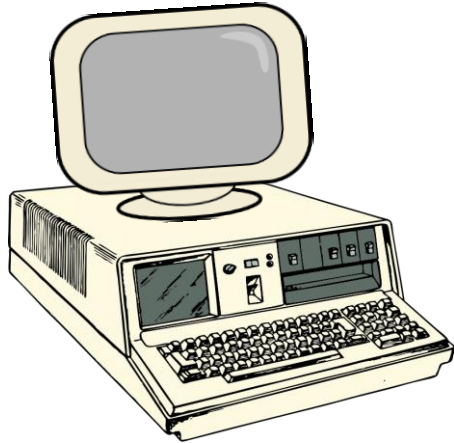
NoSQL and NewSQL Database Examples

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>

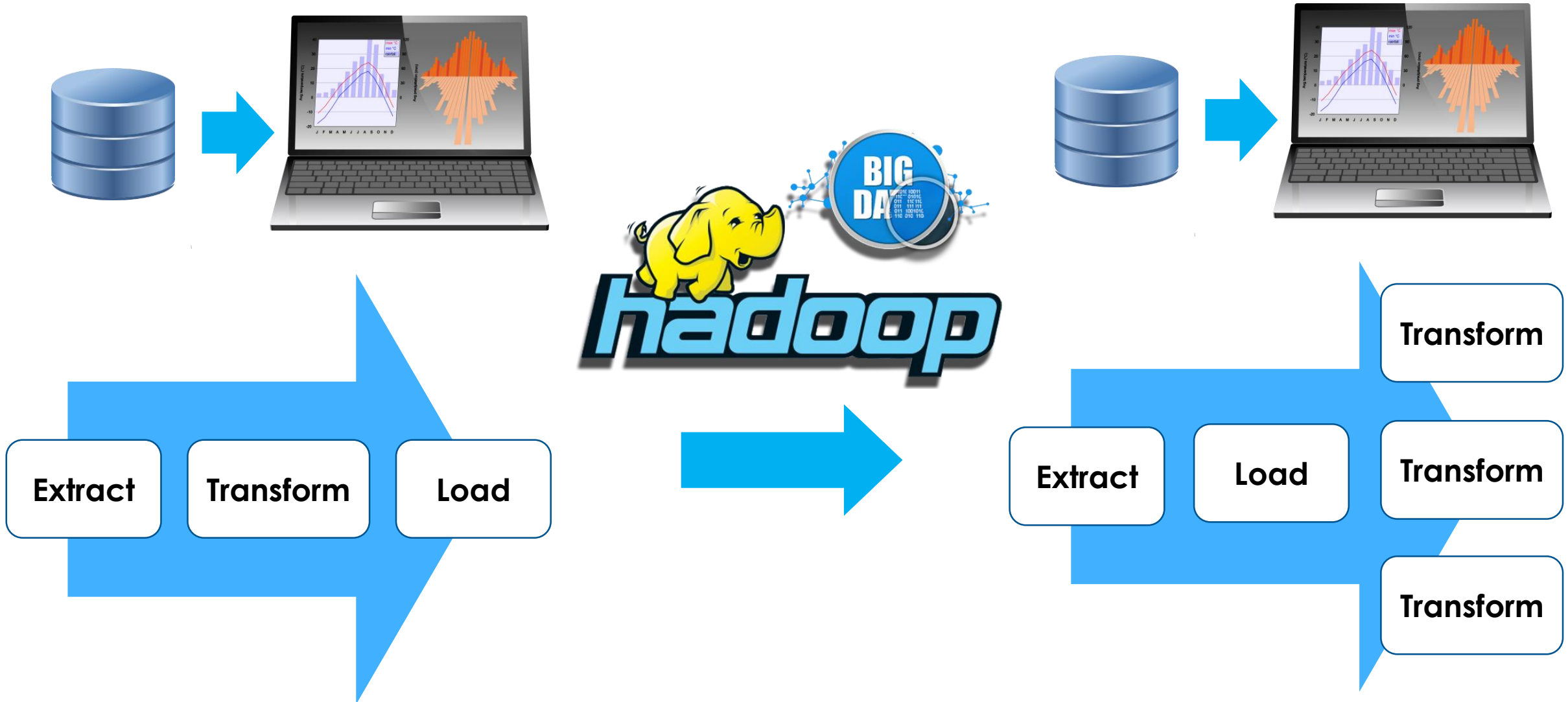
SQL and NoSQL Databases Comparison

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	<u>graph theory</u>
Relational database	variable	variable	low	moderate	<u>relational algebra</u>

Need for Speed :)

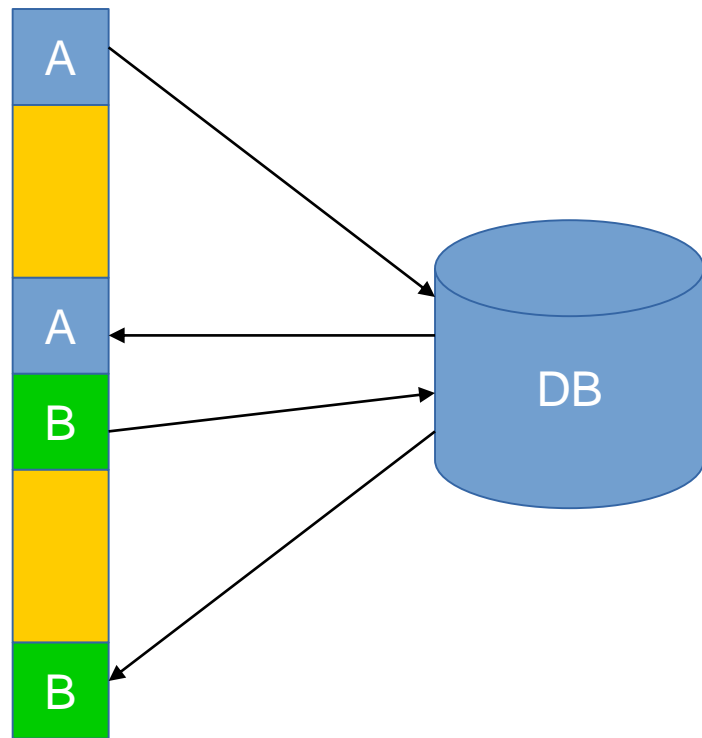


Batch Processing

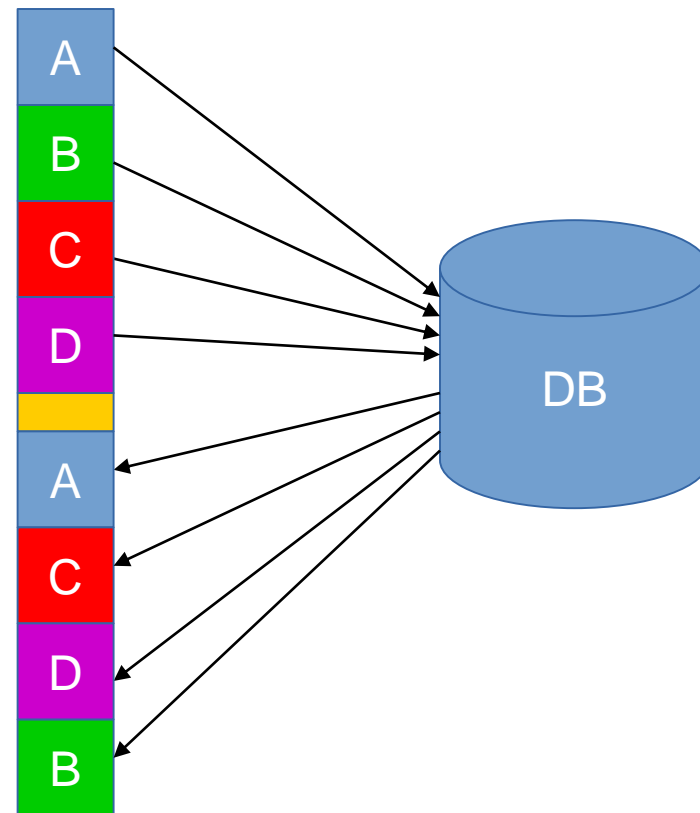


Synchronous vs. Asynchronous IO

Synchronous

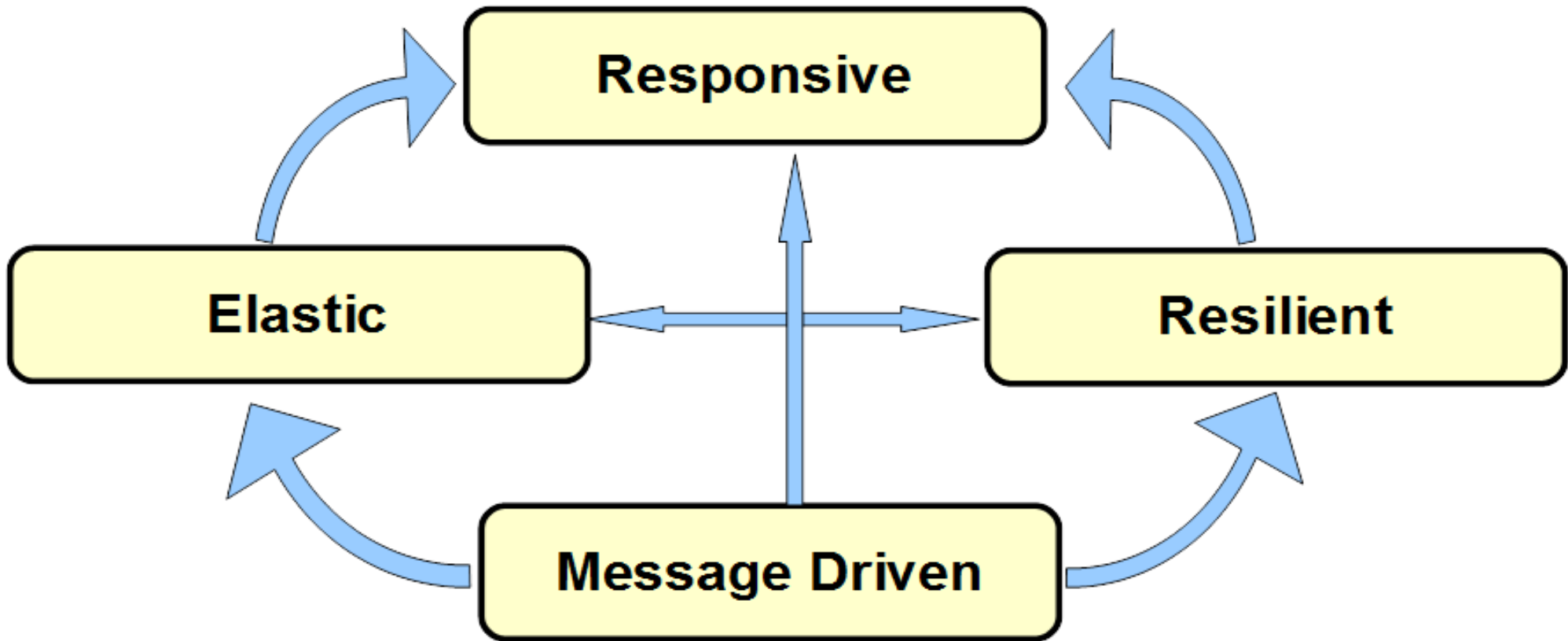


Asynchronous



Reactive Manifesto

<http://www.reactivemanifesto.org>



Scalable, Massively Concurrent

- **Message Driven** – asynchronous message-passing allows to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages [[Reactive Manifesto](#)].
- The main idea is to separate concurrent producer and consumer workers by using **message queues**.
- **Message queues** can be **unbounded** or **bounded** (limited max number of messages)
- **Unbounded** message queues can present memory allocation problem in case the producers outrun the consumers for a long period → **OutOfMemoryError**

Data / Event / Message Streams

“Conceptually, a stream is a (potentially never-ending) **flow of data records**, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as a result.”

Apache Flink: Dataflow Programming Model

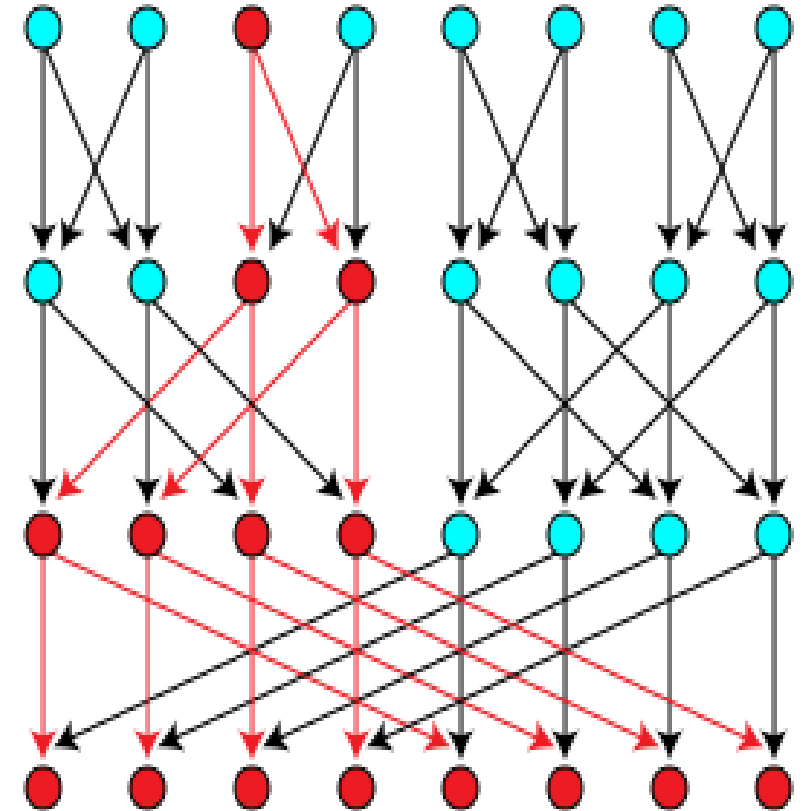
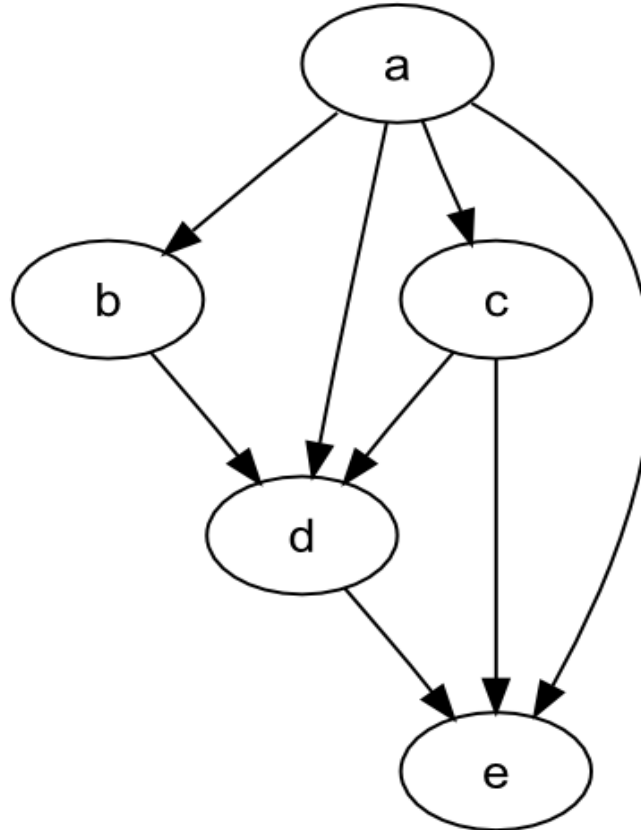
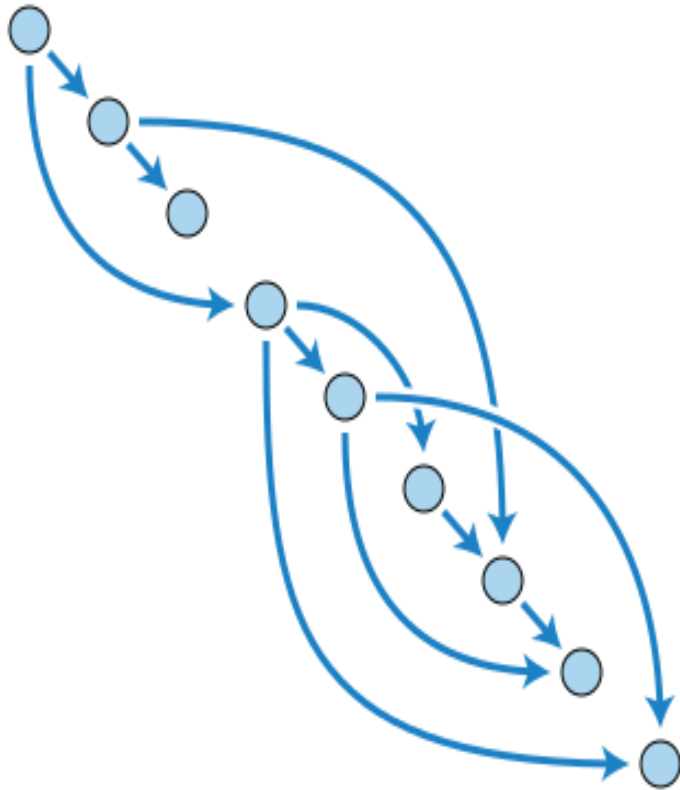
Data Stream Programming

The idea of **abstracting logic from execution** is hardly new -- it was the dream of **SOA**. And the recent emergence of **microservices** and **containers** shows that the dream still lives on.

For developers, the question is whether they want to learn yet **one more layer of abstraction** to their coding. On one hand, there's the elusive promise of a **common API to streaming engines** that in theory should let you mix and match, or swap in and swap out.

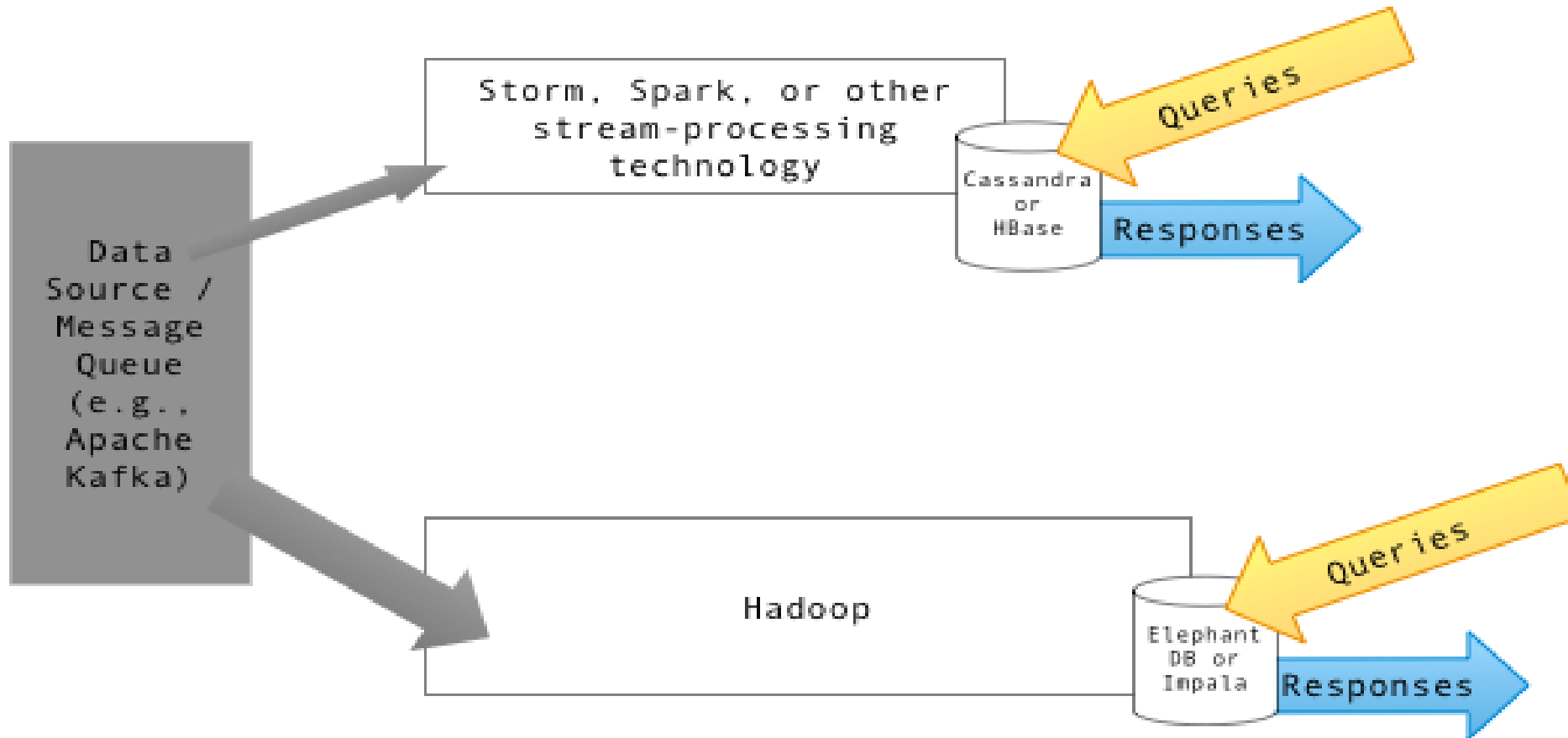
*Tony Baer (Ovum) @ ZDNet - Apache Beam and Spark:
New coopetition for squashing the Lambda Architecture?*

Direct Acyclic Graphs - DAG



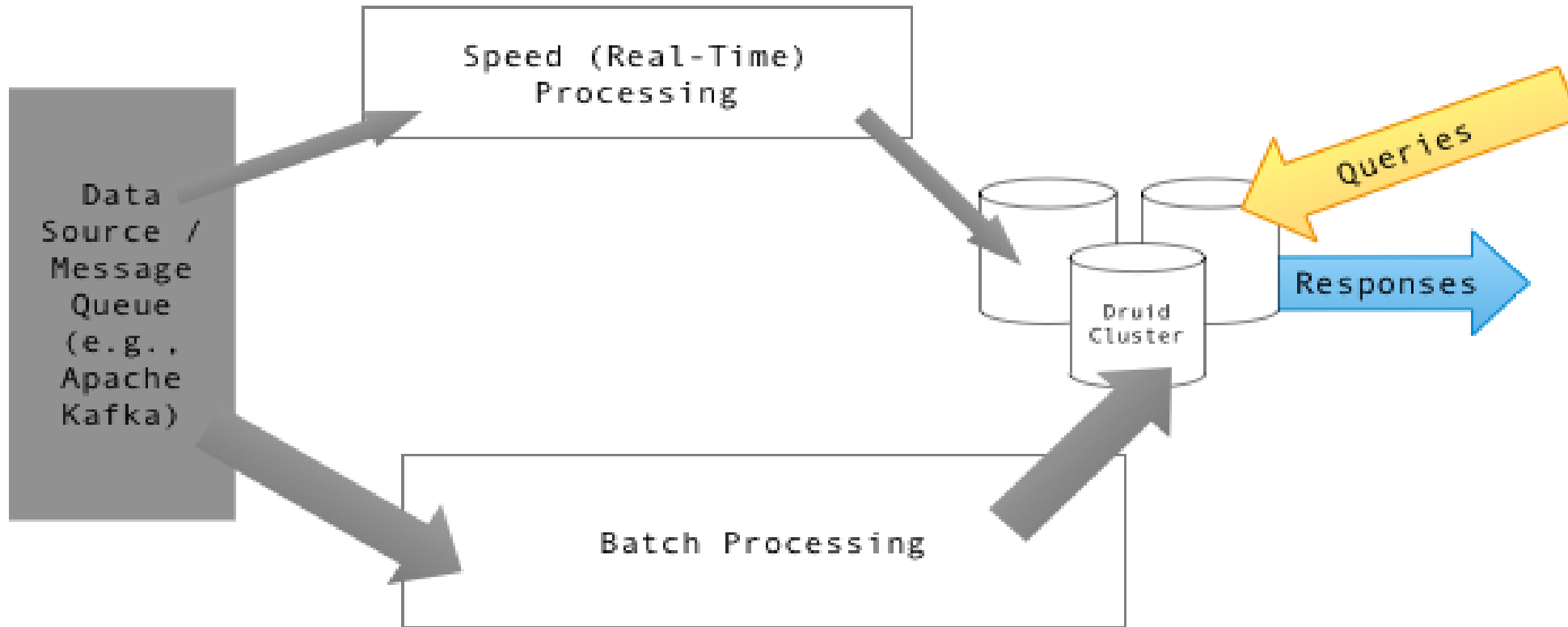
Lambda Architecture - I

Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)

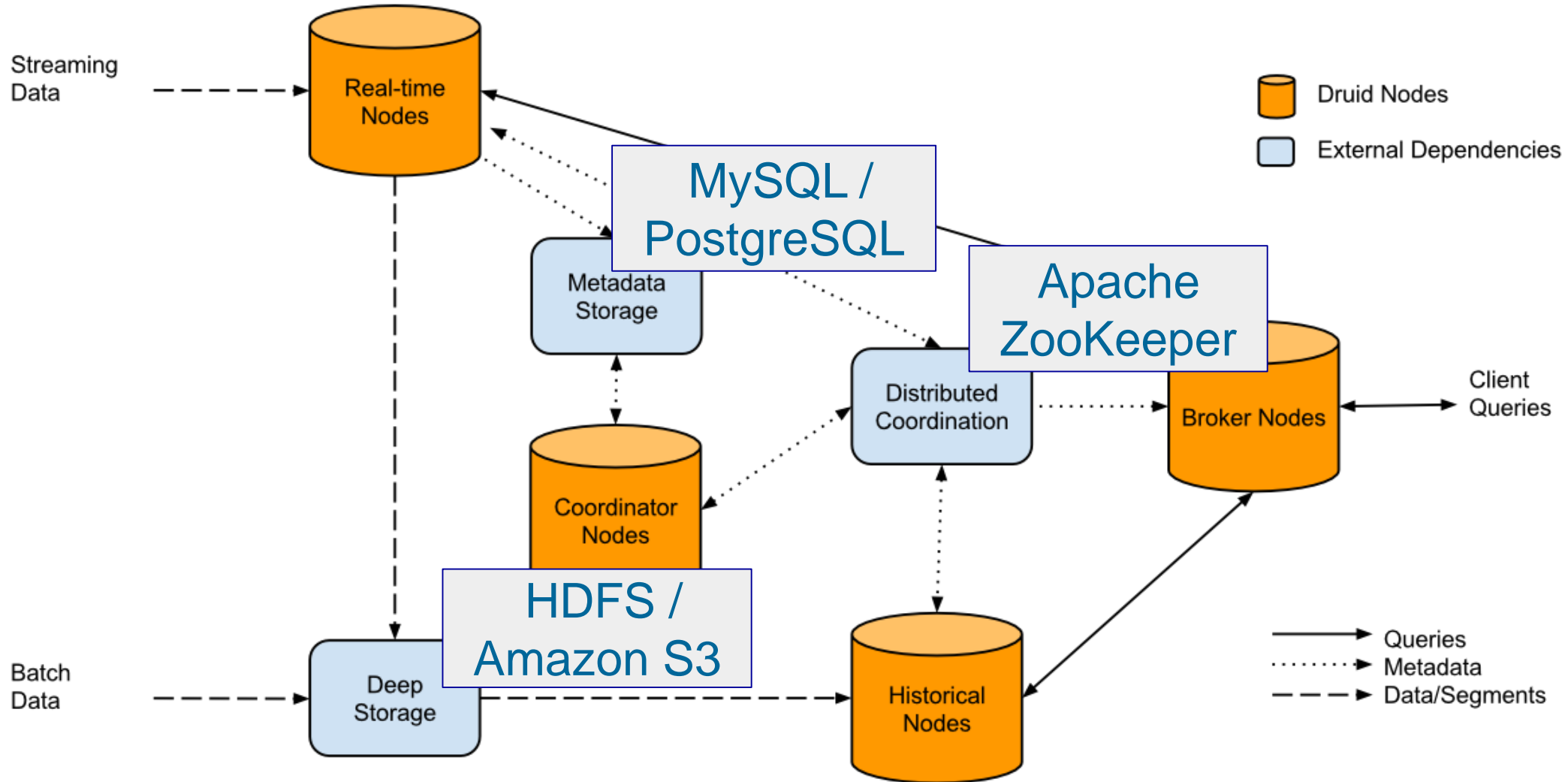


Lambda Architecture - II

Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)



Lambda Architecture - Druid Distributed Data Store



Kappa Architecture

Query = K (New Data) = K (Live streaming data)

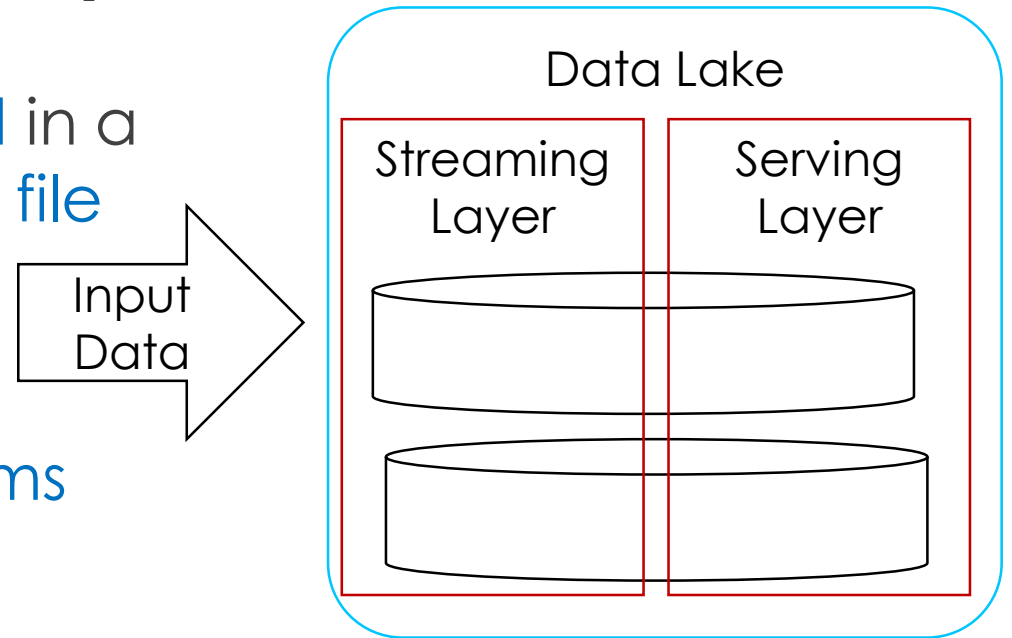
λ vs κ

- Proposed by Jay Kreps in 2014
- Real-time processing of distinct events
- Drawbacks of Lambda architecture:
 - It can result in coding overhead due to comprehensive processing
 - Re-processes every batch cycle which may not be always beneficial
 - Lambda architecture modeled data can be difficult to migrate
- Canonical data store in a Kappa Architecture system is an append-only immutable log (like Kafka, Pulsar)

Kappa Architecture II

Query = K (New Data) = K (Live streaming data)

- Multiple data events or queries are logged in a queue to be catered against a distributed file system storage or history.
- The order of the events and queries is not predetermined. Stream processing platforms can interact with database at any time.
- It is resilient and highly available as handling terabytes of storage is required for each node of the system to support replication.
- Machine learning is done on the real time basis



Zeta Architecture

- Main characteristics of [Zeta](#) architecture:
 - file system ([HDFS](#), [S3](#), [GoogleFS](#)),
 - realtime data storage ([HBase](#), [Spanner](#), [BigTable](#)),
 - modular processing model and platform ([MapReduce](#), [Spark](#), [Drill](#), [BigQuery](#)),
 - containerization and deployment ([cgroups](#), [Docker](#), [Kubernetes](#)),
 - Software solution architecture ([serverless computing](#) – e.g. [Amazon Lambda](#))
- [Recommender systems](#) and [machine learning](#)
- Business applications and dynamic global resource management ([Mesos + Myriad](#), [YARN](#), [Diego](#), [Borg](#)).

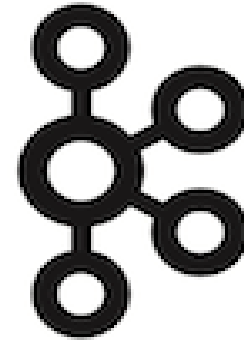
Distributed Stream Processing – Apache Projects:

- [Apache Spark](#) is an open-source cluster-computing framework. [Spark Streaming](#), [Spark Mllib](#)
- [Apache Storm](#) is a distributed stream processing – streams DAG
- [Apache Samza](#) is a distributed real-time stream processing framework.

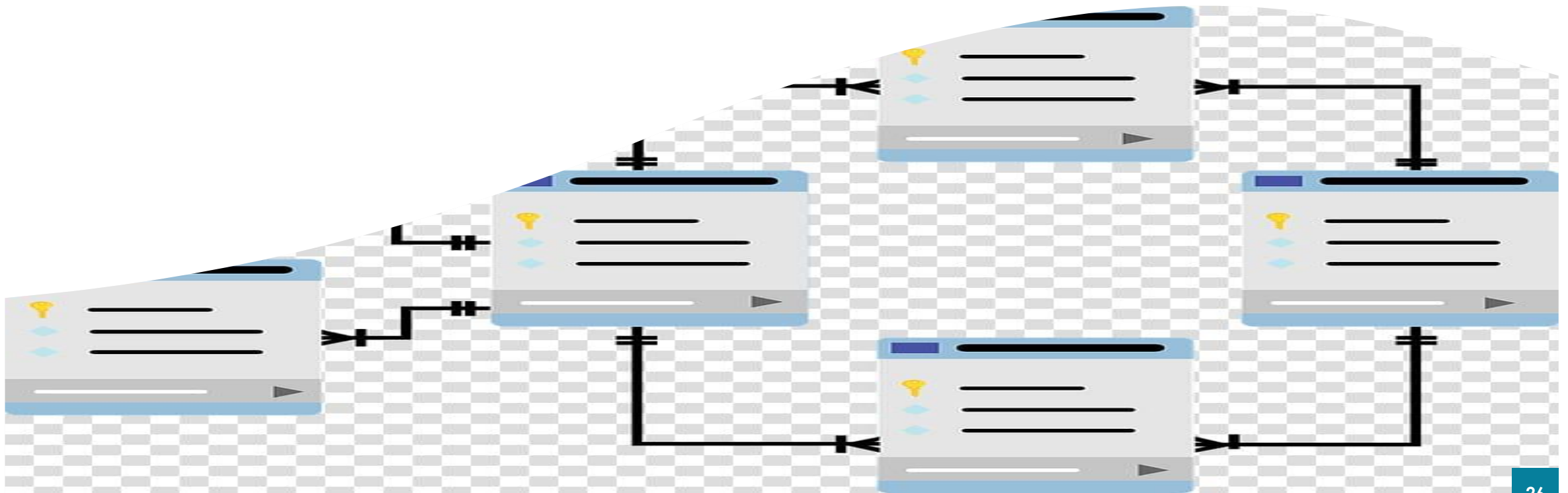


Distributed Stream Processing – Apache Projects II

- [Apache Flink](#) - open source stream processing framework – Java, Scala
- [Apache Kafka](#) - open-source stream processing (Kafka Streams), real-time, low-latency, high-throughput, massively scalable pub/sub
- [Apache Beam](#) – unified batch and streaming, portable, extensible

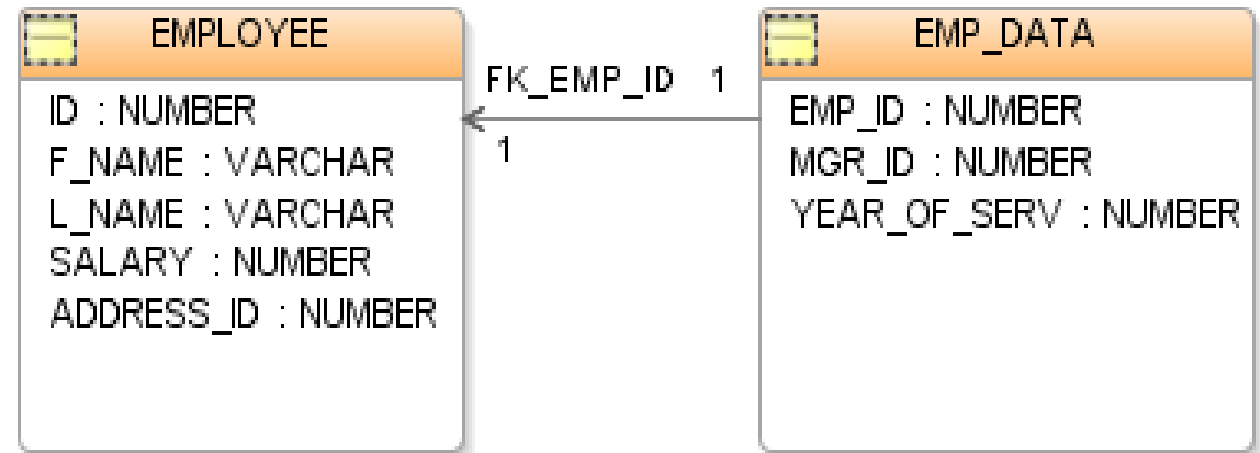
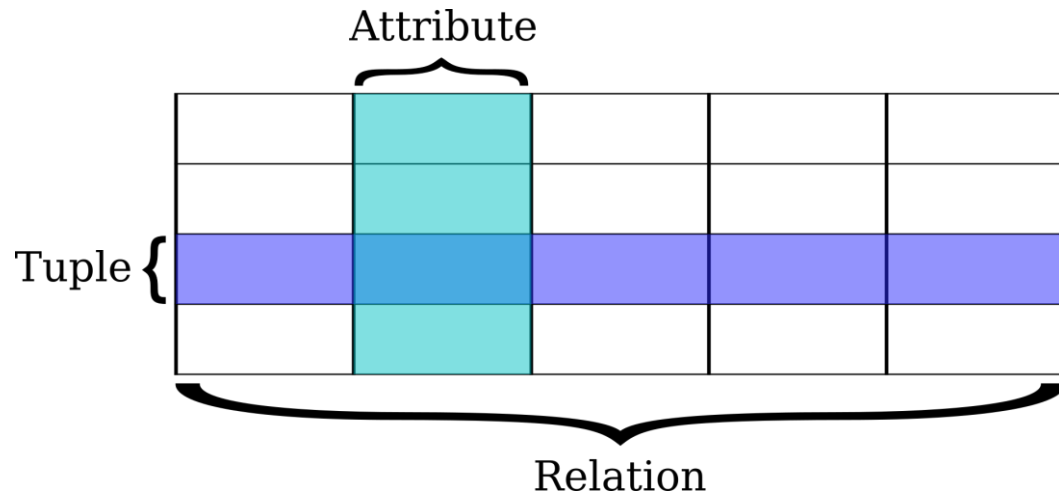


Relational Databases

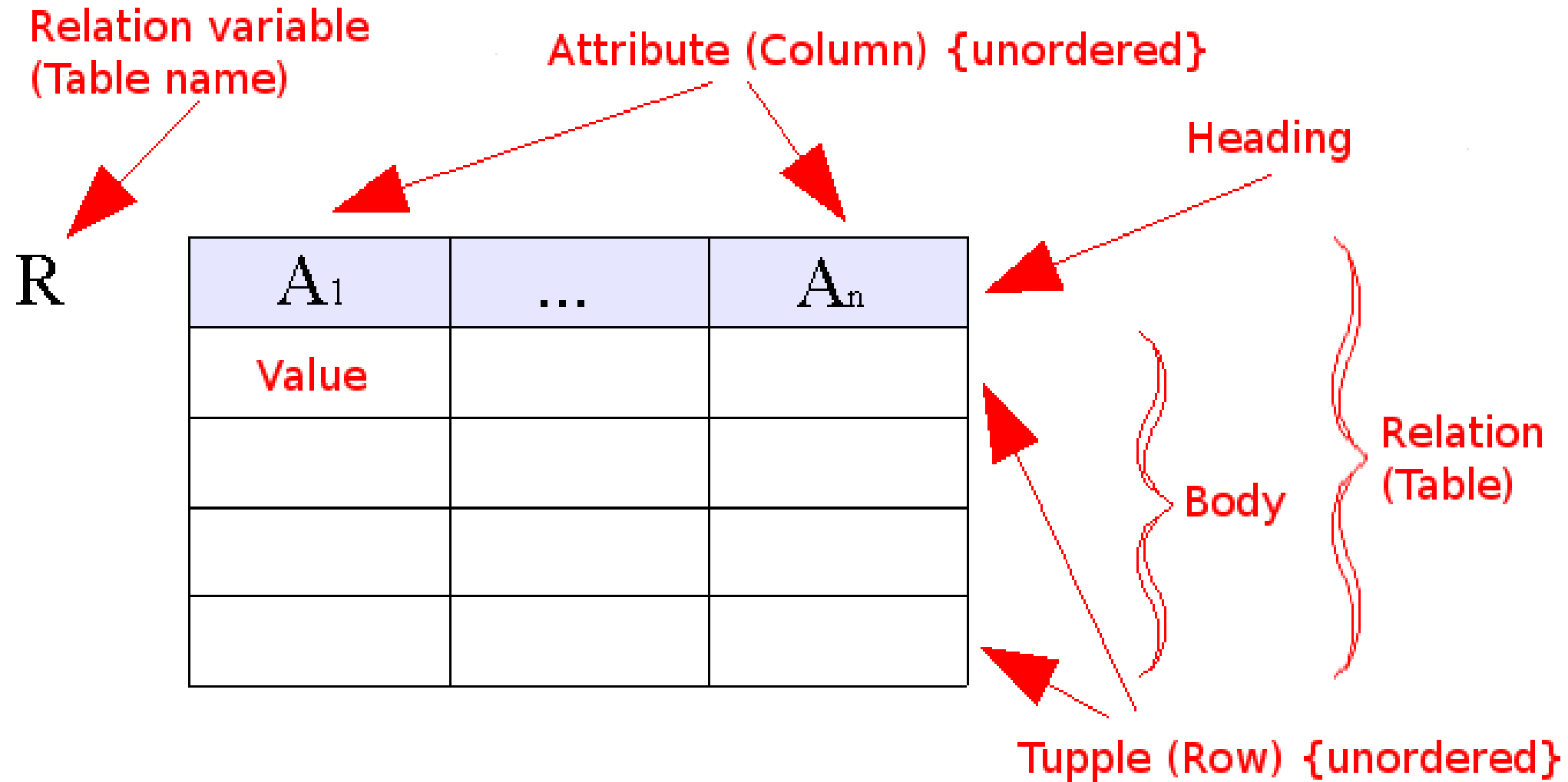


Relational Model

- 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

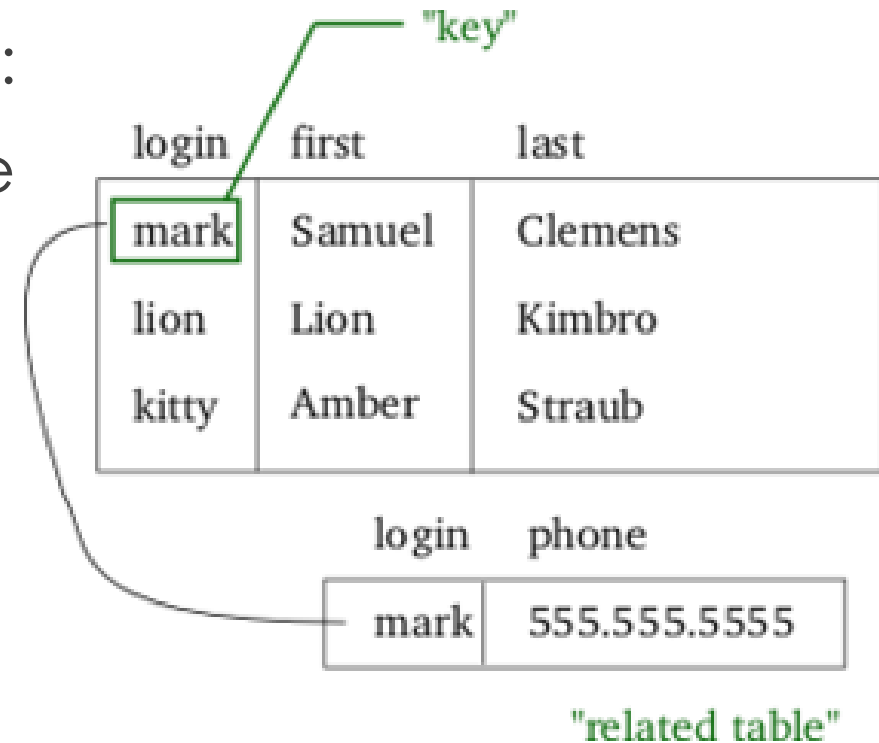
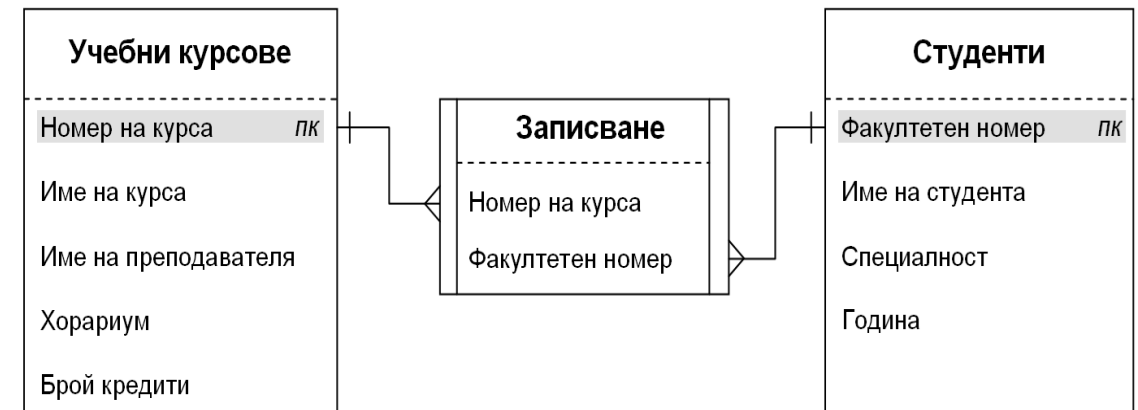
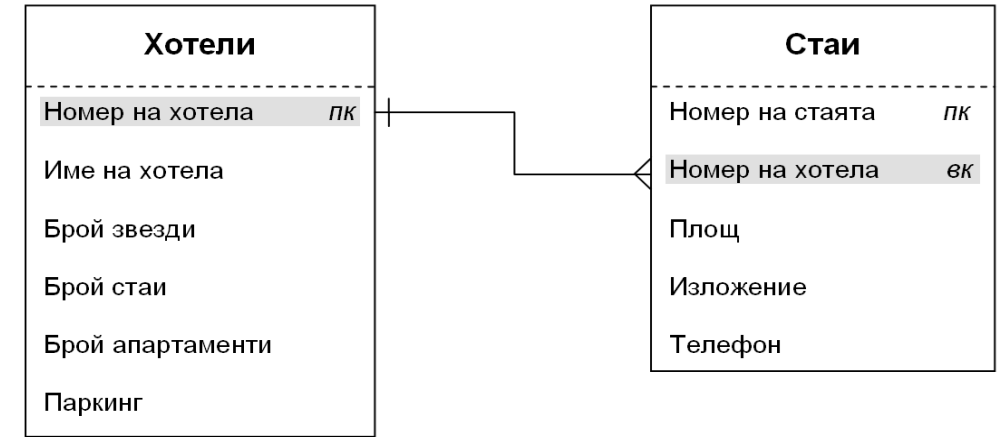
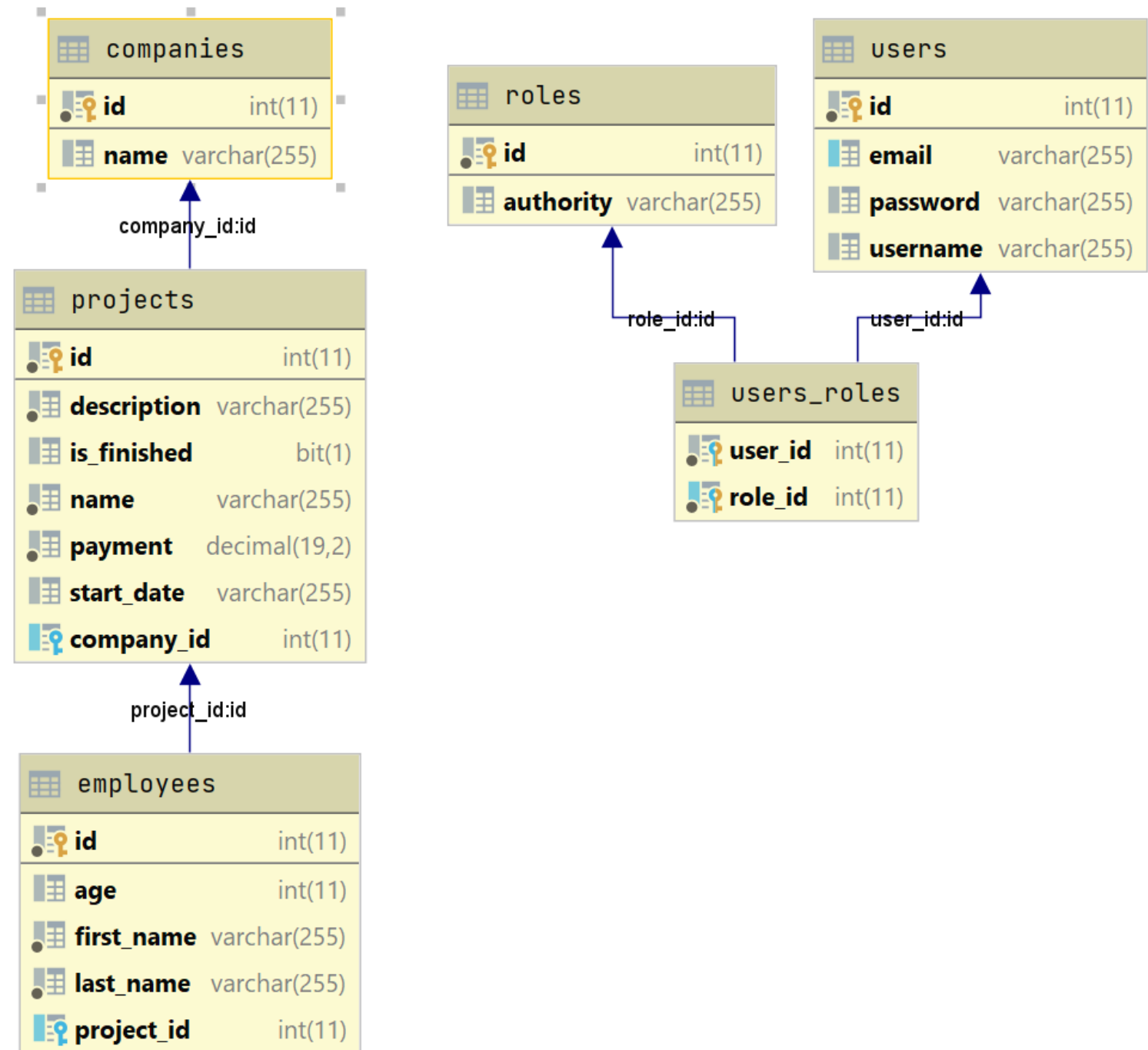


Table Relations. Cardinality

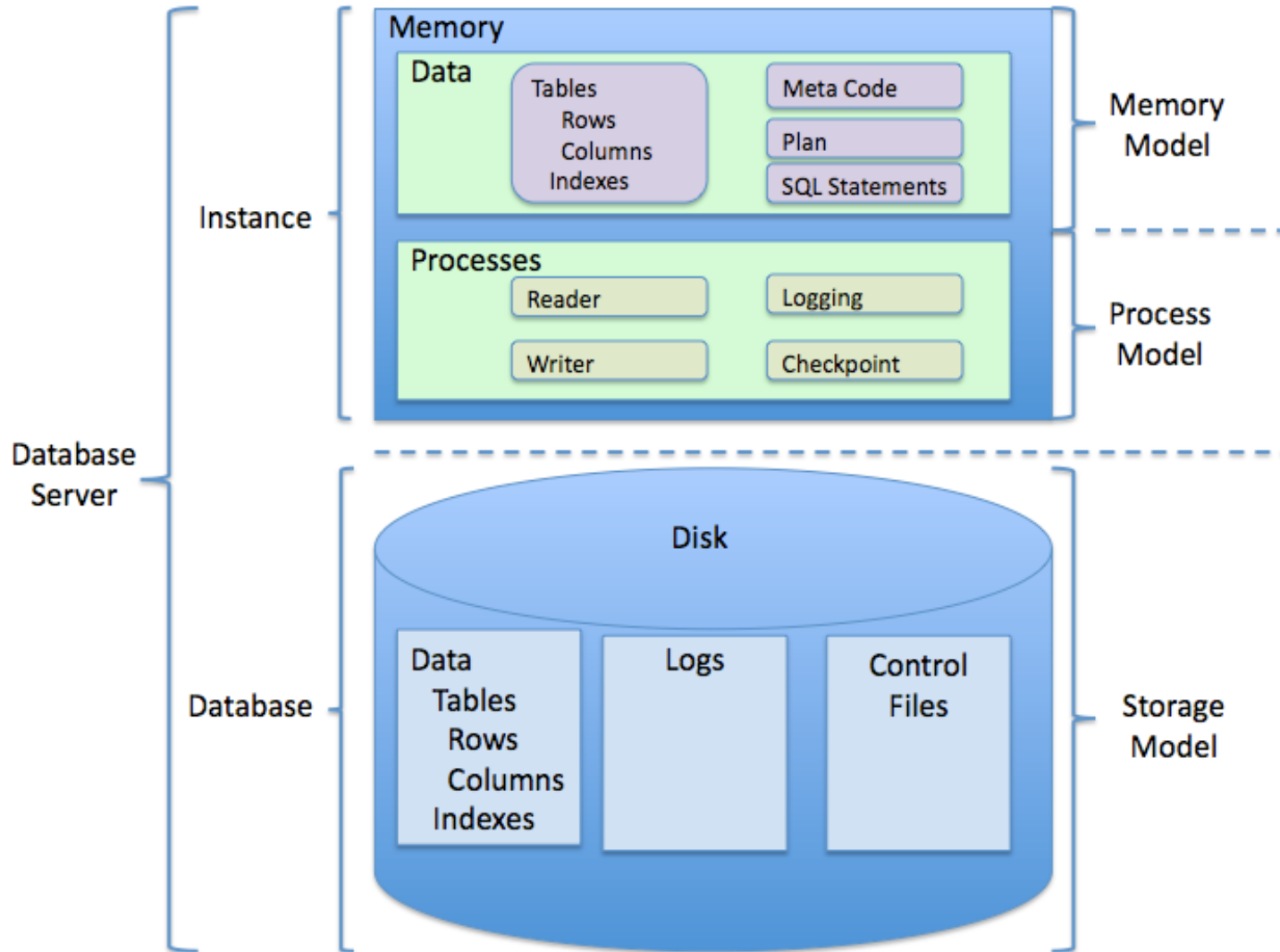
- **Relationship** is a dependency existing between two tables, when the records from first table can be connected somehow with records from second one.
- **Cardinality:**
 - One to one (1:1),
 - One to many (1:N),
 - Many to one (N:1)
 - Many to many (M:N)



Relational Schema Example ER Model



Relational Database Management System (RDBMS)

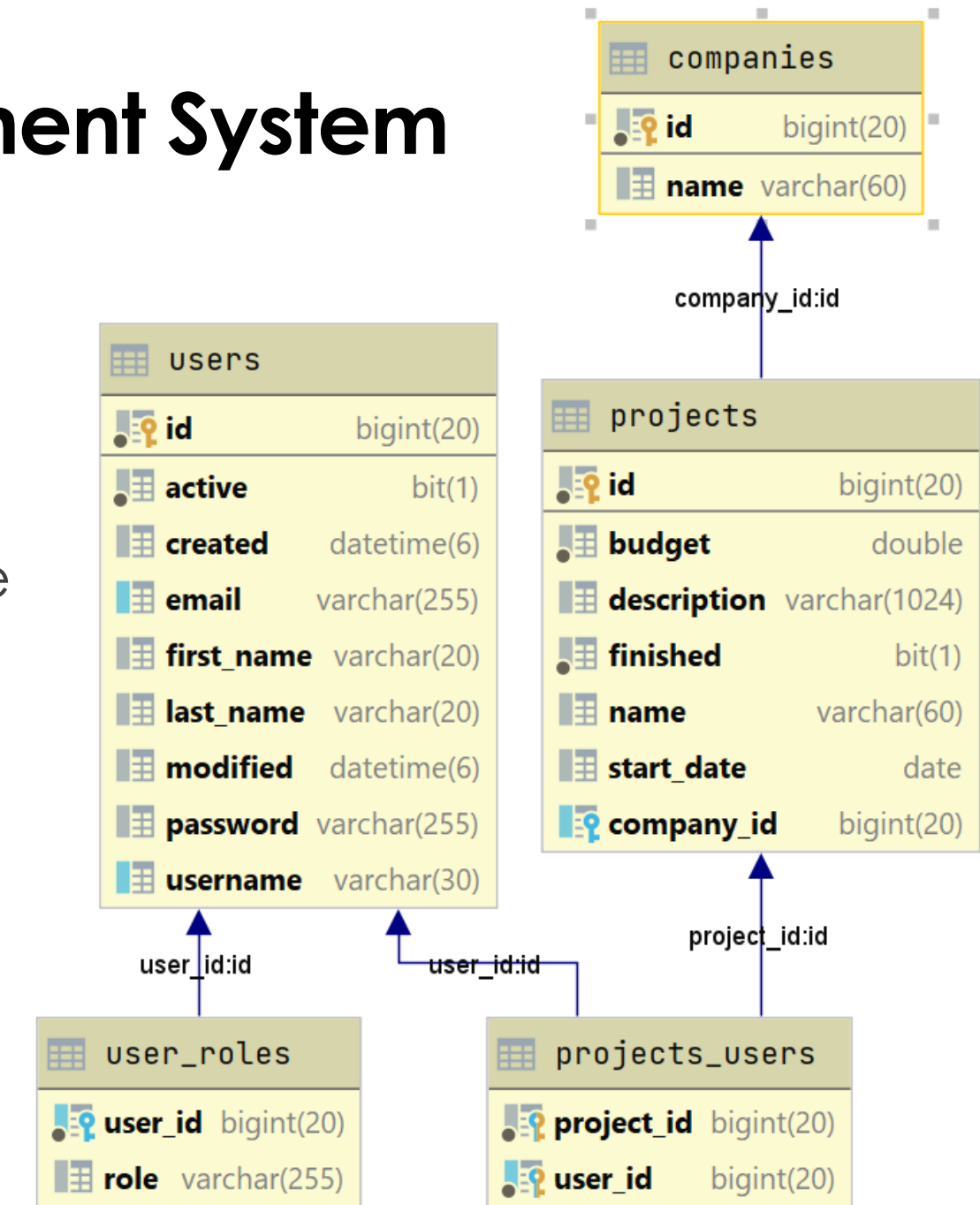


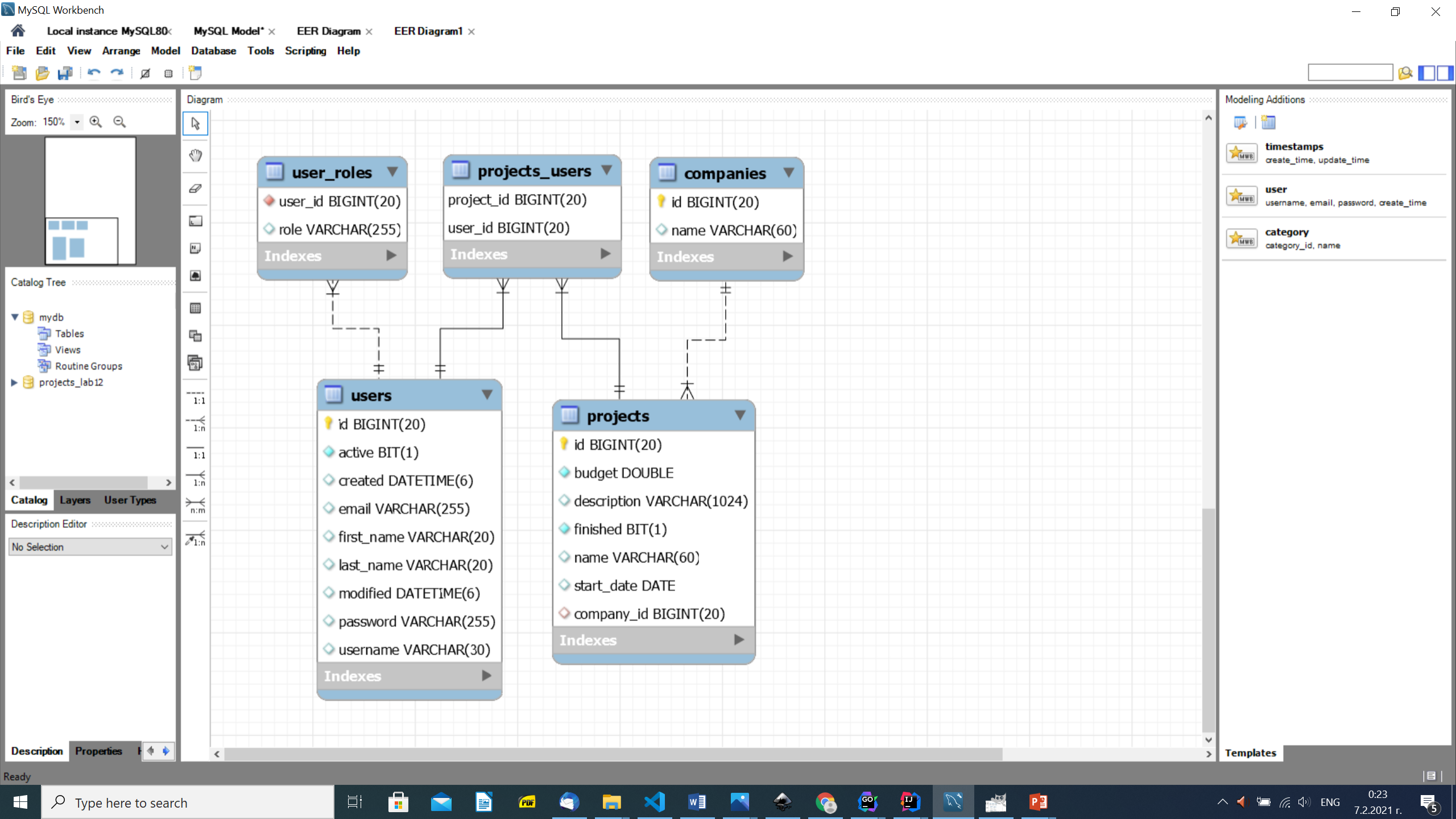
Accessing Relational Databases with Go – database/sql



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
}
```

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



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 and Query

```
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) }
result = db.Find(&users) // //Get all users - SELECT * FROM users;
if result.Error != nil { log.Fatal(result.Error) } // returns 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

```
// Get all users
// db.Session(&gorm.Session{FullSaveAssociations: true}).Updates(&user)
// result = db.Joins("Users").Find(&projects) // SELECT * FROM users;
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) // Associations mode
if err != nil { log.Fatal(result.Error) } // returns error
fmt.Printf("Number of all projects: %d\n", result.RowsAffected) // returns found records count, equals `len(users)`
utils.PrintProjects(projects)
fmt.Printf("Users in Project '%s': %v\n", projects[0].Name, users)

// SELECT * FROM companies pre-fetching projects;
result = db.Preload(clause.Associations).Find(&companies)
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)
```


Homework 4: Persistence Layer for Users and Recipes

Using **database/sql** or **Go ORM framework (gorm, xorm)** and **MySQL** implement application for **users** sharing **cooking recipes**. The application should implement *Create-Read-Update-Delete (CRUD)* functionality for **users** and **cooking recipes**.

1. Implement **User** entity with following properties:

- ID (uint, autoincrement by DB);
- unique user name;
- login name (*username* - up to 15 word characters);
- password at least 8 characters;
- gender;
- user role (*user* or *admin*);
- picture URL - optional, if missing should be substituted with a default picture of male/female avater;
- short description of the user (up to 512 characters);
- account validity status - (*active* or *deactivated*);
- date and time of registration (should be generated automatically);
- date and time of last modification (should be updated automatically);

Homework 4: Persistence Layer for Users and Recipes

2. Implement **Recipe** entity with following properties:

- ID (uint);
- author ID (the ID of the user that has shared the Recipe - should be a valid user ID);
- recipe title (up to 80 characters);
- short description of the recipe (до 256 символа);
- time to cook (in minutes);
- used products (list of products);
- recipe picture URL - optional, if missing should be substituted with a default picture;
- detailed description (up to 2048 characters);
- keywords - tags (list of tags);
- date and time of registration (should be generated automatically);
- date and time of last modification (should be updated automatically);

You can add more attributes if necessary.

Homework 4: Persistence Layer for Users and Recipes

3. Define interface **UserRepository** for user persistence (CRUD) operations with following methods:

- ***findAll()* (*[]User, error*)** - find all users
- ***findAllPagedAndSorted(pageNumber int, pageSize int, sortingAttribute string, ascending bool)* (*[]User, error*)** - should return the ***pageNumber***-th page with maximum size of ***pageSize***, ordered by ***sortingAttribute***, ***ascending*** or ***descending***.
- ***findById(id uint)* (*User, error*)** - should return the user with specific ID or nil and error if such user does not exist
- ***findByUsername(username string)* (*User, error*)** - should return the user with specific ***username***, or nil and appropriate error if such user does not exist
- ***create(user *User)* *error*** - the ID should be updated with that returned from DB, the password should be hashed using *bcrypt*
- ***update(user *User)* *error*** - updates the user data in DB, ID should be existing in DB, username can not be changed, password if changed should be hashed
- ***deleteByID(userID uint)* (*User, error*)** - should delete the user from DB and return the deleted user, or nil and error if user with such ID does not exist
- ***count()* *int*** - should return the count of users in DB

Homework 4: Persistence Layer for Users and Recipes

4. Define interface **RecipeRepository** for recipe persistence (CRUD) operations with following methods:

- ***findAll()* (*[]Recipe, error*)** - find all recipes
- ***findAllPagedAndSorted(pageNumber int, pageSize int, sortingAttribute string, ascending bool)* (*[]Recipe, error*)** - should return the ***pageNumber***-th page with maximum size of ***pageSize***, ordered, asc/desc
- ***findById(id uint)* (*Recipe, error*)** - should return the recipe with specific ID or nil and error if recipe not exist
- ***findAllByTitle(partOfTitle string)* (*[]Recipe, error*)** - returns all recipes containing part of the title as substring
- ***findAllByProducts(products []string)* (*[]Recipe, error*)** - returns all recipes containing specific at combination of all products (AND)
- ***findAllByTags(tags []string)* (*[]Recipe, error*)** - returns all recipes containing at least one of the tags (OR)
create(recipe *Recipe) error - the ID should be updated with that returned from DB
- ***createBatch(pecipes []*Recipe) error*** - the IDs should be updated with that returned from DB, **all recipes should be created atomically, in a sigle transaction**, in case of rollback DB should not be changed
- ***update(user *Recipe) error*** - updates the user data in DB, ID should be existing in DB
- ***deleteByID(recipeID uint)* (*Recipe, error*)** - should delete the recipe from DB and return the deleted recipe, or nil and error if recipe with such ID does not exist
- ***count()* *int*** - should return the count of recipes in DB

Homework 4: Persistence Layer for Users and Recipes

5. Create implementations of the above interfaces **UserRepository** and **RecipeRepository** implementing all declared methods as described above.
6. Create utility functions that print **[]User** and **[]Recipe** as formatted tables to the console.
7. Create *main application* that demonstrates the above functionality calling all method implementations with appropriate data
8. Define and implement a simple LoginController interface with following methods:
 - **login(username, password string) (User, error)** - returning the logged user if password is the same as hashed version in DB, or nil and error if not;
 - **logout()** - logging out the current user
 - **getLoggedUser() User** - returning the currently logged user
9. Add additional demo code to *main application* to demonstrate **login/logout** functionality
10. All **find*** operations should be implemented with timeout 1 second.

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>