Go – Guide avancé (niveau 5e année)

illustré avec le projet GoSQL

Avant-propos

Ce fascicule est pensé comme la synthèse que j'aurais aimé recevoir en début de stage de fin d'études : un pont entre la **théorie du langage Go** et la **réalité d'un projet conséquent** (le moteur de base de données *GoSQL*). L'objectif est double :

1. Exposer clairement les concepts clés de Go – de la syntaxe à la gestion mémoire – avec un niveau de détail adapté à une 5e année d'école d'ingénieur.
2. Illustrer chaque notion par un extrait du code GoSQL ou par une mini-expérience mesurée sur macOS, de façon à ancrer la théorie dans la pratique.

Vous n'y trouverez ni « Hello World » isolé, ni digression académique ; seulement de la matière utile pour coder, profiler, déployer et… expliquer votre travail lors d'un oral.

Table des matières

Toutes les sections sont indépendantes : si vous maîtrisez déjà la syntaxe basique, sautez directement aux chapitres 7 ou 9.

1. Historique et philosophie de Go

L'histoire de Go commence en 2007, dans les couloirs du campus Google. Trois vétérans – Robert Griesemer, Rob Pike et Ken Thompson – compilent quotidiennement d'énormes bases de code C++ et partagent la même frustration : compilation lente, complexité croissante, outillage hétérogène. D'où l'idée d'un langage « C-like » simplifié :

- **Simple** : pas d'héritage multiple, formatage unique (`gofmt`) – la lisibilité prime.
- **Sûr** : typage statique, pas d'arithmétique de pointeur sauvage (sauf `unsafe`).
- **Efficace** : goroutines légères, garbage-collector concurrent, temps de compilation mesuré en secondes.

La première version stable, Go 1.0, paraît en 2012 avec une garantie rare : *tout code conforme à Go 1 plantera de la même façon en Go 2* (promesse de compatibilité ascendante). Depuis, le langage évolue par incréments annuels ; la version 1.18 introduit les **génériques**, et la 1.22 fluidifie encore le scheduler et le GC.

Pour nous, futurs développeurs systems ou data-platforms, Go rime avec services cloud, chaînes CI/CD rapides et binaires uniques faciles à déployer – ce que démontre GoSQL : un exécutable de 8 Mo qui embarque un parseur SQL, un moteur d'exécution et un mini cache disque.

2. Modèle mémoire et garbage-collector

Comprendre le modèle mémoire de Go est crucial pour écrire du code efficace et sûr. Là où des langages comme C/C++ vous laissent gérer manuellement l'allocation et la libération de la mémoire (via `malloc`, `free`, etc.), Go délègue cette responsabilité à son runtime. Cela signifie que le développeur se concentre sur la logique métier, tandis que le langage gère la vie des objets en arrière-plan. C'est un vrai soulagement pour éviter les fuites mémoire… mais cela implique aussi de connaître les rouages du *garbage collector* (GC).

2.1 Le modèle M:P:G – une gestion intelligente des goroutines

Lorsque vous lancez une goroutine en Go (`go maFonction()`), vous créez une tâche légère gérée par le runtime. Contrairement à un thread classique, une goroutine consomme très peu de mémoire (2 à 4 ko au démarrage) et ne bloque pas directement le système.

Go utilise un modèle de planification nommé **M:P:G** :

- **G** (Goroutine) : unité de tâche que vous créez.
- **P** (Processor) : structure qui gère une file de goroutines prêtes à exécuter.
- **M** (Machine) : thread système utilisé pour exécuter les goroutines.

Quand une goroutine est bloquée (exemple : appel réseau, lecture disque, `select{}` inactif), le scheduler gare son thread (M) et donne le P à une autre goroutine prête. Cela permet d'optimiser l'utilisation des cœurs logiques et de paralléliser efficacement sans générer des milliers de threads systèmes.

Exemple concret :

```
go fmt.Println("Bonjour") // Une goroutine
```

Vous pouvez lancer 10 000 de ces instructions sans crash ni surcharge – testez-le avec une boucle !

2.2 Fonctionnement du garbage collector (GC) tricolore

Le GC de Go fonctionne avec un algorithme **tricolore** basé sur trois étapes :

1. **Marquage initial (stop-the-world)** : pause très courte (quelques microsecondes) pour identifier les objets racines (accessibles).
2. **Marquage concurrent** : le programme continue à tourner pendant que le GC marque tout ce qui est accessible depuis les racines.
3. **Balayage (sweeping)** : libération des objets inaccessibles.

Le tout est possible grâce à une *barrière d'écriture* (write barrier) qui détecte les modifications pendant que le GC tourne en arrière-plan.

Illustration : imaginez que votre programme construit un arbre d'objets (par ex. une requête SQL avec des sous-expressions). Une fois la requête exécutée, Go identifie automatiquement que ces objets ne sont plus accessibles et les libère.

2.3 Paramètre GOGC : gérer le compromis mémoire / performance

Le paramètre `GOGC` (Garbage Collection Target Percentage) contrôle la fréquence de déclenchement du GC. Une valeur de `100` signifie :

- dès que le heap a doublé, le GC démarre.
- valeur plus haute = moins de GC = plus rapide mais plus gourmand en RAM.

- valeur plus basse = nettoyage plus fréquent = RAM plus faible, mais plus de pauses.

Cas pratique avec GoSQL : lors d'un import massif de 1 million de lignes CSV, le GC s'active fréquemment. Voici les mesures obtenues :

GOGC	Temps total d'import	Pic RAM
100	4,8 s	320 Mo
200	4,5 s	450 Mo

On remarque que doubler le `GOGC` réduit légèrement le temps d'import, mais fait grimper l'usage mémoire de 40 %. Pour nous, la valeur `150` offre un bon équilibre.

Astuce développeur : vous pouvez modifier ce paramètre dynamiquement dans votre code avec :

```
import "runtime/debug"
debug.SetGCPercent(150)
```

Cela permet de régler le comportement du GC différemment selon la charge.

2.4 En résumé

- Le modèle M:P:G permet une exécution concurrente légère, sans bloquer le système.
- Le garbage collector est conçu pour tourner majoritairement *en parallèle* avec l'application.
- Le paramètre `GOGC` vous donne une clé pour équilibrer vitesse et mémoire.
- Sur des projets comme GoSQL, ce fonctionnement vous permet de charger de grands volumes en mémoire, sans fuite ni arrêt brutal, tout en conservant de bonnes performances.

Visualisez-le ainsi : vous écrivez du code synchrone, lisible et maintenable, mais Go orchestre les détails bas-niveau pour vous – un vrai gain pour la productivité et la robustesse.

Contrairement à C/C++, Go confie l'allocation et la libération mémoire au runtime. Chaque goroutine démarre avec un stack de 2 à 4 kiB, extensible au besoin. Le *scheduler* utilise un modèle *M:P*:G* (Machines = threads, Processors ≈ cœurs logiques, Goroutines = tâches). Lorsqu'une goroutine s'endort (appel système bloquant, `select{}` sans cases prêtes), le scheduler parque son thread et en réveille un autre – l'effet est quasi transparent.

Le **garbage-collector** fonctionne selon un algorithme tri-couleur concurrent :

1. Marquage des objets accessibles (phase stop-the-world micro-seconde).
2. Marquage incrémental concurrent : l'application tourne, les *write-barriers* signalent les mutations.
3. Balayage et libération de la mémoire inatteignable.

Le réglage `GOGC` (ratio entre heap actuel et heap majorée) permet de choisir entre empreinte mémoire et fréquence de pause. Sur GoSQL, importer un CSV de 1 M lignes déclenche fréquemment le GC ; nous avons mesuré :

GOGC	Temps total d'import	Pic RAM
100	4,8 s	320 Mo
200	4,5 s	450 Mo

À retenir : augmenter `GOGC` accélère parfois le throughput, mais au prix d'une RAM plus haute ; chez nous, le GOGC 150 constitue un bon compromis.

3. Système de types modernes : génériques et interfaces implicites

Le système de types de Go a beaucoup évolué depuis ses débuts. Longtemps réputé pour sa simplicité presque minimaliste (pas d'héritage, pas de surcharge, etc.), il s'est renforcé avec l'arrivée des **génériques** en Go 1.18, tout en conservant ses principes fondateurs. Deux piliers sont à connaître : les **interfaces implicites**, et les **types paramétriques**.

3.1 Interfaces : du « duck typing » à la sécurité à la compilation

En Go, une interface ne se déclare pas comme en Java ou C#. Il n'y a **pas besoin d'annoncer qu'on implémente une interface** : c'est le compilateur qui vérifie la correspondance. Si une struct possède les méthodes exigées, alors elle **satisfait automatiquement l'interface**.

Exemple réel du projet GoSQL :

```
type Query interface {
    Execute(e *Engine) (*ResultSet, error)
    String() string
}
```

Ici, `Query` est une interface décrivant une entité capable de s'exécuter (avec un moteur `Engine`) et de fournir une représentation texte (`String()`).

Maintenant, n'importe quelle structure comme `SelectQuery`, `InsertQuery`, ou même `DeleteQuery` peut devenir une `Query` si elle implémente ces deux méthodes, **sans avoir à le déclarer**.

```
type SelectQuery struct {
    Table string
    Where Expression
}

func (s *SelectQuery) Execute(e *Engine) (*ResultSet, error) { ... }
func (s *SelectQuery) String() string                         { ... }
```

Avantage : ce découplage fort permet de faire évoluer le code sans cascade de changements. Par exemple, lors de notre sprint de développement d'un `DELETE`, il a suffi de créer une nouvelle struct `DeleteQuery` avec les deux méthodes : tout a fonctionné sans modifier une ligne ailleurs.

En clair :
Une interface Go = une promesse implicite.
Une struct = une candidate silencieuse.
Le compilateur = l'arbitre.

Cela favorise une approche modulaire et testable : on peut injecter des *mocks* dans les tests, tant qu'ils respectent la signature.

3.2 Génériques : facteur X de productivité depuis Go 1.18

Pendant longtemps, Go ne permettait pas d'écrire des fonctions ou structures génériques. Il fallait utiliser `interface{}` (équivalent de `Object` en Java) et faire des conversions avec des assertions de type (`.(int)`, `.(string)`, etc.). Cela introduisait du flou, de la lenteur, et surtout un manque de sécurité.

Depuis Go 1.18, on peut écrire :

```
func Min[T constraints.Ordered](a, b T) T {
    if a < b { return a }
    return b
}
```

Ici, `T` est un type paramétré. Il doit appartenir à l'ensemble des types « ordonnables » (`int`, `float64`, `string`, etc.).

Application dans le projet GoSQL

Nous avions un module `btree` utilisé pour créer des index. Avant les génériques, chaque clé (int, string, etc.) devait être castée à la volée, ce qui ralentissait l'accès et nuisait à la clarté.

Avec les génériques, nous avons défini :

```
type Ordered interface {
    constraints.Ordered // types comparables avec < > <= >=
}

type Node[T Ordered] struct {
    Keys   []T
    Values []any
    Child  []*Node[T]
}
```

Ainsi, que les clés soient des `int` (clé primaire) ou des `string` (clé secondaire), une seule implémentation générique suffit. Le compilateur instancie pour chaque type utilisé.

4. Organisation du code : packages, modules et workspace

L'organisation du code est l'un des points forts de Go. Contrairement à d'autres langages où la structure est souvent libre ou dictée par le framework, Go propose une convention claire et standardisée basée sur trois niveaux :

1. **Le module** : unité de distribution (fichier `go.mod`)
2. **Les packages** : unités logiques (répertoire avec fichiers `.go`)
3. **Le workspace (go.work)** : gestion multi-modules (optionnel)

4.1 Le module : base de toute application Go

Un **module** est déclaré via `go mod init` et décrit votre projet comme un tout : nom, dépendances, version, etc. Cela remplace le vieux `GOPATH` global et permet d'avoir une gestion locale et précise des dépendances.

Exemple dans GoSQL :

```
$ go mod init github.com/user/gosql
```

Ce fichier `go.mod` contiendra ensuite les dépendances (comme `constraints`, `testify`, etc.), ainsi que d'éventuelles directives `replace` ou `require`.

4.2 Packages : découpage interne logique

Un **package** en Go correspond à un répertoire contenant des fichiers `.go` avec le même nom de package. Il peut être public (`pkg`) ou privé (`internal`).

Organisation typique (GoSQL ou projet web) :

```
mydb/                  # module racine
├── go.mod
├── cmd/               # applications principales (CLI, API, etc.)
│   └── db_cli/        # CLI pour interroger la base
├── internal/          # packages non exportés
│   ├── engine/        # moteur d'exécution SQL
│   └── storage/       # accès aux fichiers binaires
├── pkg/               # packages réutilisables (publics)
│   ├── sql/           # types de requête, parseurs
│   └── btree/         # structures d'index génériques
```

Pourquoi `internal/` ?

Le mot-clé `internal` protège le code de mauvaise utilisation externe. Par exemple, un projet web ne devrait pas pouvoir accéder directement à `storage.ReadPage` depuis l'extérieur du module.

Cela évite les **dependency leaks** : quand un module tiers dépend de parties internes de votre code (ex: votre cache, vos constantes internes) qu'il n'aurait jamais dû utiliser.

4.3 Cas réel : projet web avec API REST + background workers

Prenons un exemple classique : une API web écrite en Go (par exemple avec `Gin` ou `Echo`), accompagnée d'un processus en tâche de fond (worker de messagerie, cron, etc.) :

```
webapp/
├── go.mod
├── cmd/
│   ├── api/          # démarrage de l'API REST
│   └── worker/       # tâche de fond (ex: process emails)
├── internal/
│   ├── services/     # logique métier
│   ├── db/           # accès base de données
│   └── mailer/       # client SMTP ou SendGrid
├── pkg/
│   └── models/       # structs partagées (User, Product…)
```

4.4 Et si on a plusieurs modules ? Le go.work

Depuis Go 1.18, `go work` permet d'agréger plusieurs modules dans un workspace. Très pratique en mono-repo avec des microservices Go indépendants :

```
go.work
go.work.sum
```

Contenu du `go.work` :

```
go 1.22

use (
    ./gosql
    ./gosql-cli
    ./gosql-ui
)
```

Chaque sous-répertoire a son propre `go.mod`, mais peut interagir sans passer par Git ou un `replace` manuel. Idéal pour le développement local.

5. Syntaxe essentielle : déclarations, contrôle de flux

Le langage Go est volontairement sobre dans sa syntaxe. Cette simplicité n'est pas une faiblesse, bien au contraire : elle permet une lecture rapide, une relecture facile, et évite les effets de bord complexes. Voici un passage en revue des syntaxes clés que tout développeur Go doit maîtriser.

5.1 Déclarations : var, const, :=

Go propose plusieurs manières de déclarer une variable, selon le contexte.

```
var age int = 28        // déclaration explicite
const Pi = 3.14         // constante
name := \"Alice\"         // déclaration courte avec inférence
```

La forme `:=` est particulièrement pratique en local (dans une fonction), car elle permet de combiner déclaration et initialisation en une seule ligne. C'est aussi celle que vous verrez le plus souvent dans les boucles ou dans les tests d'erreurs.

Exemple dans GoSQL :

```
page, err := ReadPage(f, pageID)
if err != nil { return nil, err }
```

5.2 Conditions : if, switch

Go permet d'initier des variables directement dans le `if`, ce qui évite de polluer l'espace de noms.

```
if f, err := os.Open(\"data.db\"); err == nil {
    defer f.Close()
    fmt.Println(\"Fichier ouvert !\")
} else {
    log.Fatal(err)
}
```

Cela encourage une programmation défensive claire : chaque ressource ouverte est gérée immédiatement, souvent via `defer` (voir section GC).

Le `switch` est tout aussi pratique : pas besoin d'écrire `break`, pas besoin de regrouper les cas manuellement.

```
switch cmd {
case \"select\":
    handleSelect()
case \"insert\":
    handleInsert()
default:
    fmt.Println(\"Commande inconnue\")
}
```

5.3 Boucles : le for universel

Go a une seule structure de boucle : `for`. Elle remplace les `while`, les `foreach`, les `repeat…until` de nombreux autres langages.

```
// comme un for classique
for i := 0; i < 10; i++ {
```

```
    fmt.Println(i)
}

// boucle infinie
for {
    fmt.Println(\"Serveur actif\")
    time.Sleep(1 * time.Second)
}

// boucle sur slice
names := []string{\"Alice\", \"Bob\", \"Charlie\"}
for _, name := range names {
    fmt.Println(name)
}
```

5.4 Le pattern “machine à états” dans GoSQL

Ce qui est intéressant avec Go, c’est que certains patterns de design deviennent naturels grâce à la syntaxe. Dans GoSQL, le lexer (analyseur lexical) est construit comme une **machine à états**.

```
for state := lexText; state != nil; {
    state = state(l)
}
```

Ici, chaque fonction d’état (`stateFn`) reçoit un lexer et retourne la prochaine fonction d’état. Cela évite un `switch` géant, rend le code modulaire et lisible comme un automate.

Ce que ça apporte :

- Une transition claire entre les étapes d’analyse (`lexText` → `lexNumber` → `lexIdent`…).
- Un code testable étape par étape.
- Une meilleure séparation des responsabilités (chaque fonction d’état gère un cas précis).

À retenir :

Syntaxe Go	Équivalent en d'autres langages	Pourquoi c'est mieux en Go ?
`:=`	`int x = 5;` (C) ou `let x = 5` (JS)	moins verbeux, plus lisible
`if init; cond`	`if (...) {}` avec bloc externe	scope limité à la condition
`for {...}`	`while (true) {...}`	un seul mot-clé à apprendre
Machine à états	`while / switch` imbriqués	structure claire et réutilisable

Cette syntaxe volontairement minimaliste peut désarçonner au début, mais elle est redoutablement efficace sur le long terme. Dans GoSQL, elle a permis de créer un parseur compact, lisible et maintenable, ce qui est précieux dans un moteur de base de données.

6. Fonctions, méthodes et design d'API

En Go, les méthodes peuvent être définies sur un type en utilisant un **receiver**. Ce receiver peut être soit une **valeur**, soit un **pointeur**. Le choix dépend de l'usage prévu.

- **Receiver valeur** : utilisé si la méthode ne modifie pas la structure.
- **Receiver pointeur** : utilisé si la méthode modifie l'état interne ou si on veut éviter de copier un objet lourd.

Par exemple, la méthode `PageSizeKB()` de `HeaderInfo` est en lecture seule, donc définie sur une valeur. En revanche, `SetEncoding()` modifie le champ `Encoding`, elle est donc définie sur un pointeur `*HeaderInfo`.

```
func (h HeaderInfo) PageSizeKB() int {
    return int(h.PageSize) / 1024
}

func (h *HeaderInfo) SetEncoding(enc string) {
    h.Encoding = enc
}
```

Pour construire des objets complexes de manière claire, on privilégie dans l'API les **méthodes chaînables** et les **options fonctionnelles**. Cela permet de rendre la configuration fluide, sans avoir à passer de longues listes de paramètres.

```
engine := NewEngine(
    WithCache(256),
    WithLogger(myLogger),
)
```

Cette approche améliore la lisibilité, limite les erreurs et rend le code plus facile à maintenir ou à tester. C'est un choix de style que nous avons appliqué dans GoSQL pour garder l'interface simple à utiliser.

7. Concurrence : du scheduler aux patterns de canaux

7.1 La goroutine, unité de travail ultra légère

En Go, une **goroutine** est une tâche légère, bien plus légère qu'un thread système. Elle est gérée entièrement par le runtime de Go. Sa création est quasi instantanée et sa mémoire initiale est de quelques kilo-octets seulement.

Concrètement, on crée une goroutine simplement avec le mot-clé `go` :

```
go fmt.Println("Bonjour")
```

Cela exécute `fmt.Println` en parallèle, sans bloquer le programme principal.

Pourquoi c'est utile ?

Prenons l'exemple de GoSQL : lorsqu'on lit plusieurs pages d'un fichier `.db`, au lieu de le faire une par une, on lance une goroutine pour chaque page à lire. Cela permet de tirer parti des cœurs multiples du processeur.

```
pages := make(chan *Page, 256) // canal tamponné
for _, id := range pageIDs {
    go func(pid int64) {
        pages <- storage.ReadPage(file, pid) // lecture parallèle
    }(id)
}
```

Chaque goroutine lit une page (opération lente car disque) et envoie le résultat dans un **channel**. Un autre morceau du programme récupère les résultats et les traite.

Cette approche permet :

- de ne pas attendre que chaque lecture soit finie avant de commencer la suivante,
- de lisser les pics de latence disque,
- d'être beaucoup plus rapide dès qu'on a plusieurs cœurs ou un disque SSD.

`context.Context` : gestion de l'annulation

Go fournit un type `context.Context` qui permet de gérer l'annulation ou les timeouts. Dans GoSQL, si l'utilisateur interrompt l'import ou la requête, on peut tout arrêter proprement grâce à ce `context`.

7.2 `select` et gestion des timeouts

Le mot-clé `select` permet d'écouter plusieurs canaux en même temps. Il est essentiel en concurrence pour éviter de rester bloqué indéfiniment.

Exemple : on veut lire depuis le canal `pages`, mais ne pas bloquer si rien ne vient dans les 100 ms :

```
select {
case p := <-pages:
    handle(p)
case <-time.After(100 * time.Millisecond):
```

```
    log.Println("Lecture trop lente")
}
```

Cela améliore la robustesse de l’application : si un disque est lent ou un fichier corrompu, on évite de geler le programme.

8. Outils de la toolchain Go

Le binaire `go` embarque la majeure partie du quotidien d'un développeur :

- **`go vet`** : détecte erreurs courantes (absence de `err` check, printf mismatch).
- **`go test -race`** : identifie les accès concurrents non protégés.
- **`go tool pprof`** : profile CPU/mémoire, sert de boussole avant toute optimisation.
- **`go test -trace`** : traceur chronologique, idéal pour comprendre l'ordonnancement des goroutines.

Une session typique :

```
$ go test ./internal/engine -run=^$ -bench=. -benchmem -cpuprofile cpu.prof
$ go tool pprof -http=:0 cpu.prof   # ouvre l'UI dans le navigateur
```

9. Étude de cas GoSQL : de la page b-tree au plan d’exécution

9.1 Décodage du header SQLite

Le fichier `.db` commence par une signature « SQLite format 3\0 ». Dans `storage/header.go`, nous lisons les 100 premiers octets et convertissons la taille de page big-endian :

```
pageSize := binary.BigEndian.Uint16(buf[16:18])
if pageSize == 1 { pageSize = 65536 } // cas spécial SQLite
```

9.2 Organisation en pages et cellules

Chaque page *b-tree* contient un tableau d’offsets vers les cellules ; c’est un véritable « mini-filesystem » interne. GoSQL charge d’abord la page, puis les cellules en parallèle pour remplir un canal de `Row` traité par le moteur.

9.3 Plan d’exécution naïf puis optimisé

Le premier prototype exécutait un **table-scan** linéaire. Après profilage, nous avons ajouté :

1. **Push-down predicate** : appliquer `WHERE age > 30` avant la projection.
2. **Sélection d’index** : si un index B-tree existe, éviter le scan complet.
3. **Pipeline goroutines** : chaque étape (`Scan → Filter → Project`) tourne dans sa goroutine et passe les résultats par channel.

Résultat : la requête `SELECT name FROM users WHERE id = 42` passe de 8 ms à 0,7 ms sur 1 M lignes.

10. Bonnes pratiques & anti-patterns

- **Ne pas exposer `*sql.DB` globalement** ; passez-le par injection.
- **Toujours vérifier `err`** – même si `fmt.Println(err)` paraît redondant.
- **Éviter l'over-engineering** : commencer simple, profiler, puis optimiser.

Dans GoSQL, la tentation d'écrire un optimiseur de requête complet était forte ; nous avons au contraire limité le MVP à trois passes et réservé le CBO (cost-based) pour la v 1.1.

12. Glossaire, bibliographie et ressources

- **G** : Goroutine, instance du code utilisateur.
- **P** : Processor, jeton qui exécute des G sur un thread M.
- **Escape analysis** : décision du compilateur pour placer une variable sur le stack ou le heap.