

# MAC - Méthode d'accès aux données

## Bases de données orientées documents

09 November 2025

### Table des matières

<b>1 Concepts fondamentaux .....</b>	<b>2</b>
1.1 Introduction .....	3
1.2 Différences avec les bases relationnelles .....	3
1.2.1 Stockage des documents .....	3
1.2.2 Schémas flexibles .....	3
1.2.3 Adaptation au développement agile .....	3
1.2.4 Requêtes puissantes .....	3
1.2.5 Systèmes distribués .....	3
1.3 Couchbase .....	3
<b>2 Modélisation des données .....</b>	<b>4</b>
2.1 Modélisation logique .....	5
2.1.1 Schéma dynamique et flexible .....	5
2.1.2 Structuration des données: deux approches .....	5
2.1.2.1 1. Approche de séparation .....	5
2.1.2.2 2. Approche d'imbrication .....	5
2.1.3 Choix de modélisation: avantages et inconvénients .....	5
2.1.3.1 Avantages de l'imbrication .....	5
2.1.3.2 Désavantages de l'imbrication .....	6
2.1.3.3 Avantages de séparation .....	6
2.1.3.4 Désavantages de séparation .....	6
2.1.4 Règles empiriques .....	6
2.2 Modélisation physique .....	6
2.2.1 Concepts de base .....	6
2.2.2 Correspondance avec le modèle relationnel .....	6
2.2.3 Collection et Scope .....	7
2.2.4 Valeurs .....	7
2.2.4.1 Exemple de valeur JSON .....	7
<b>3 N1QL: Langage de requête .....</b>	<b>7</b>
3.1 Introduction .....	8
3.2 Découverte du schéma (INFER) .....	8
3.3 Opérations de base .....	8
3.3.1 SELECT avec * .....	8
3.3.2 SELECT avec RAW .....	8
3.3.3 Accès aux champs imbriqués .....	8
3.3.4 Pattern matching .....	8
3.3.5 Métadonnées et clés (META) .....	8
3.3.5.1 META() .....	8
3.3.5.2 USE KEYS .....	8
3.4 Opérateurs sur collections et tableaux .....	9
3.4.1 IN / NOT IN .....	9

---

3.4.2	WITHIN / NOT WITHIN .....	9
3.4.3	ANY / EVERY .....	9
3.4.4	ARRAY .....	9
3.4.5	FIRST .....	9
3.5	Agrégation et sous-requêtes .....	9
3.5.1	GROUP BY .....	9
3.5.2	HAVING .....	9
3.5.3	Sous-requêtes .....	9
3.6	Gestion des valeurs NULL et MISSING .....	10
3.6.1	Tests de présence .....	10
3.6.2	Diagramme de Venn des états .....	10
3.7	Indexation .....	10
3.7.1	Types d'index .....	10
3.7.2	Index sur éléments de tableau .....	10
3.7.3	Consultation des index .....	11
3.7.4	Avantages et inconvénients .....	11
3.8	Jointures et relations .....	11
3.8.1	ANSI JOIN .....	11
3.8.2	Exemple de JOIN .....	11
3.8.3	Optimisation avec index couvrant .....	11
3.8.4	NEST .....	11
3.8.4.1	Exemple NEST vs JOIN .....	11
3.8.5	UNNEST .....	12
3.8.5.1	Exemple UNNEST .....	12
3.9	Fonctions sur les tableaux .....	12
3.9.1	ARRAY_AGG .....	12
3.9.2	ARRAY_COUNT .....	13
3.9.3	ARRAY_MAX / ARRAY_MIN .....	13
3.9.4	Autres fonctions utiles .....	13
3.10	Aspects avancés .....	13
3.10.1	Requêtes paramétrées .....	13
3.10.2	Portée des alias .....	13
3.11	Bonnes pratiques .....	13

# 1 Concepts fondamentaux

## 1.1 Introduction

Les bases de données orientées documents sont un moyen avancé de stocker des données au format natif (ex: JSON) plutôt que des colonnes/lignes. Ce système permet de rechercher, récupérer et stocker directement les documents dans leur format natif.

Exemples de systèmes: **MongoDB**, **Couchbase**, **OrientDB**, **RavenDB**

## 1.2 Différences avec les bases relationnelles

### 1.2.1 Stockage des documents

Les bases de données relationnelles stockent généralement les données dans des tables séparées et liées, permettant à des objets uniques d'être répartis sur plusieurs tables.

Dans les bases de données de documents, toutes les informations pour un document ou un objet donné peuvent être stockées dans une seule instance. Il n'est pas nécessaire de faire un mappage lors du chargement ou de la récupération d'éléments.

**Avantage:** Les bases de données de documents sont généralement plus rapides pour cette raison.

### 1.2.2 Schémas flexibles

Les stores de documents ont un **schéma auto-descriptif et dynamique**, adaptable au changement.

- Pas besoin de prédéfinir le schéma
- Les champs peuvent varier d'un document à l'autre
- On peut modifier la conception à n'importe quelle étape
- Pas besoin de migration de schéma

### 1.2.3 Adaptation au développement agile

Grâce au modèle de données intuitif, les bases de données orientées documents rendent le développement plus simple et plus rapide.

- Les objets dans le code peuvent être mappés naturellement sur les documents
- Pas besoin de décomposer des données entre les tables
- Pas besoin d'intégrer une couche ORM distincte ou d'utiliser des JOIN coûteux

### 1.2.4 Requêtes puissantes

Les bases de données de documents permettent d'interroger les documents de manière flexible, avec un langage de requête expressif offrant diverses fonctionnalités (filtrage, jointures, agrégations, etc.) tout en utilisant la fonction d'indexation.

#### ⚠ Warning

Les transactions ACID sont fondamentales dans les bases relationnelles. Dans les bases de données orientées documents, la gestion des transactions peut poser des défis.

### 1.2.5 Systèmes distribués

Les bases de données de documents sont essentiellement des systèmes distribués.

- Chaque document est une unité indépendante, plus facilement répartis sur les serveurs
- Haut niveau de disponibilité grâce à la réplication
- Permet d'isoler plusieurs charges de travail dans un cluster

## 1.3 Couchbase

Couchbase est une base de données à la fois orientée documents et clé-valeur. Elle offre une architecture distribuée pour garantir les performances, l'évolutivité et la disponibilité.

- Crée par Damien Katz (créateur original de CouchDB)
- Fusion entre CouchDB et Membase

- **Important:** Couchbase  $\neq$  CouchDB
- Langage de requête: **N1QL** (syntaxe proche de SQL)

## 2 Modélisation des données

### 2.1 Modélisation logique

#### 2.1.1 Schéma dynamique et flexible

Couchbase Server n'impose pas de schéma uniforme.

- On peut utiliser un attribut nommé `type` (ou `_type`) pour indiquer la catégorie des données
- On peut regrouper tous les documents de même type dans une collection
- Les structures de documents peuvent varier, même entre documents de même type
- Les schémas sont entièrement définis et gérés par les applications

**Exemple:**

```
{
  "_type": "user",
  "userId": 123
}
```

#### 2.1.2 Structuration des données: deux approches

Avec une base de données de documents, il existe deux approches:

##### 2.1.2.1 1. Approche de séparation

Maintenir les données normalisées dans des documents séparés (similaire aux bases relationnelles).

**Exemple:**

```
key - invoice::1
{
  "BillTo": "Lynn Hess", "InvoiceDate": "2018-01-15",
  "InvoiceNum": "ABC123", "ShipTo": "Herman Trisler, 4189 Oak Drive" }

key - invoiceitem::1811cfcc-05b6-4ace-a52a-be3aad24dc52
{
  "InvoiceId": "1", "Price": "1000.00", "Product": "Brake Pad", "Quantity": "24" }

key - invoiceitem::29109f4a-761f-49a6-9b0d-f448627d7148
{
  "InvoiceId": "1", "Price": "10.00", "Product": "Steering Wheel", "Quantity": "5" }
```

Le champ `InvoiceId` est similaire à une clé étrangère, mais les bases de données de documents n'appliquent pas cette relation de la même manière que les bases relationnelles.

##### 2.1.2.2 2. Approche d'imbrication

Dénormaliser les données en les imbriquant dans leur document parent.

**Exemple:**

```
{
  "BillTo": "Lynn Hess",
  "InvoiceDate": "2018-01-15 00:00:00.000",
  "InvoiceNum": "ABC123",
  "ShipTo": "Herman Trisler, 4189 Oak Drive",
  "Items": [
    { "Price": "1000.00", "Product": "Brake Pad", "Quantity": "24" },
    { "Price": "10.00", "Product": "Steering Wheel", "Quantity": "5" },
    { "Price": "20.00", "Product": "Tire", "Quantity": "2" }
  ]
}
```

#### ⚠ Warning

Le champ `InvoiceId` n'est plus présent comme clé étrangère. Le contenu des `invoiceitems` est déjà imbriqué localement dans l'élément `Items`.

#### 2.1.3 Choix de modélisation: avantages et inconvénients

##### 2.1.3.1 Avantages de l'imbrication

**Vitesse d'accès:** Tout intégrer dans un seul document signifie qu'on n'a besoin que d'une seule recherche dans la base de données et qu'on n'a pas besoin de jointures.

**Tolérance aux pannes:** Dans une base de données distribuée, les documents non-imbriqués seraient sur plusieurs machines. En intégrant, nous introduisons moins d'opportunités pour des erreurs potentielles.

#### 2.1.3.2 Désavantages de l'imbrication

**Incohérence:** Comme les données ne sont pas normalisées, il peut y avoir de la redondance, et donc des problèmes connus liés à la mise-à-jour des données redondantes.

**Requêtes complexes:** Interroger les parties imbriquées des documents peut être plus complexe.

**Documents volumineux:** L'imbrication de beaucoup de données peut conduire aux documents volumineux composés de nombreuses données dupliquées.

#### 2.1.3.3 Avantages de séparation

**Cohérence:** On conserve une copie canonique des informations.

**Requêtes simplifiées:** L'interrogation des données non-imbriquées est plus simple (plus proche du relationnel).

**Meilleure utilisation du cache mémoire:** Les documents canoniques restent dans le cache plutôt que d'avoir plusieurs copies consultées moins fréquemment.

**Utilisation plus efficace du matériel:** L'imbrication crée des documents plus volumineux avec des données redondantes, alors que la séparation aide à réduire l'espace disque et la RAM nécessaire.

#### 2.1.3.4 Désavantages de séparation

**Recherches multiples:** Plusieurs recherches et jointures sont nécessaires pour lier les données séparées dans plusieurs documents.

**La cohérence est forcée:** Se référer à une version canonique signifie que des mises à jour seront reflétées dans chaque contexte où il est utilisé, ce qui peut ne pas être souhaitable.

#### 2.1.4 Règles empiriques

Si...	Considérer...
Relation 1:1 ou 1:N	Imbriqué
Relation M:N	Docs séparés
Lecture principalement des champs parents	Docs séparés
Lecture principalement des champs parents ou enfants (pas les deux)	Docs séparés
Lecture des champs parents et enfants (les deux)	Imbriqué
Lectures beaucoup plus nombreuses que les écritures	Imbriqué
Risque accepté de données incohérentes entre copies	Imbriqué
Optimisation de la vitesse d'accès	Imbriqué
Cohérence des données prioritaire	Docs séparés
Utilisation efficace du cache	Docs séparés
Version imbriquée serait lourde	Docs séparés

## 2.2 Modélisation physique

#### 2.2.1 Concepts de base

Le stockage des données dans Couchbase est différent des bases de données relationnelles.

Couchbase utilise des **items** pour stocker des clés-valeurs:

- Les **clés** (keys) sont des identificateurs uniques (au sein d'un bucket)
- Les **valeurs** peuvent être binaire ou document JSON
- Les **buckets** regroupent les items

#### 2.2.2 Correspondance avec le modèle relationnel

Serveur Couchbase	Base de données Relationnelle
-------------------	-------------------------------

Bucket	Database
Scope	—
Collection	Table
Item avec un attribut type	Table
Item (document ou clé-valeur)	Row

### 2.2.3 Collection et Scope

**Collection:** Organise les documents à l'intérieur d'un Bucket.

Exemple: Dans un Bucket contenant des informations de voyage, les documents des aéroports peuvent être dans une collection aéroports, tandis que les documents des hôtels sont dans une collection hôtels.

**Scope:** Mécanisme de regroupement de plusieurs collections. Les collections sont regroupées en fonction de type de contenu ou de la phase de déploiement (test, production, etc.)

#### ⚠ Warning

À la création d'un nouveau Bucket, une nouvelle Collection et un nouveau Scope, chacun nommé `_default` lui sont attribués.

### 2.2.4 Valeurs

Couchbase permet deux catégories de valeurs:

**Binaire:** Tout type de valeurs et toute forme de binaire est acceptable. Une valeur binaire ne peut pas être analysée, indexée ou interrogée: elle ne peut être récupérée que par clé.

**JSON:** JSON fournit une représentation riche pour les entités. Couchbase Server peut analyser, indexer et interroger les valeurs JSON.

#### 2.2.4.1 Exemple de valeur JSON

```
{
  "a1": 2,
  "a2": "a",
  "a3": {
    "b1": [ 1, 2, 3 ]
  },
  "a4": [
    { "c1": "simple", "c2": 10 },
    { "c1": "example", "c3": 10 }
  ]
}
```

Les attributs peuvent représenter:

- Des types de base: nombre, chaîne de caractères, booléen
- Des types complexes: documents et tableaux intégrés

Dans l'exemple: `a1` et `a2` sont des types simples, `a3` est un document imbriqué et `a4` est un tableau de documents imbriqués.

## 3 N1QL: Langage de requête

### 3.1 Introduction

**N1QL (Non-First Normal Form Query Language)** est le langage de requête pour Couchbase qui étend SQL pour supporter les documents JSON avec des structures imbriquées.

Structure d'une requête N1QL simple:

```
SELECT <projection>
FROM <bucket.scope.collection>
WHERE <conditions>
```

### 3.2 Découverte du schéma (INFER)

```
INFER `travel-sample`.inventory.route;
```

Retourne des métadonnées sur les documents :

- `#docs` : nombre de documents analysés
- `%docs` : pourcentage de documents avec un champ
- `samples` : exemples de valeurs
- `type` : type de données (string, number, object, array, etc.)

### 3.3 Opérations de base

#### 3.3.1 *SELECT avec \**

```
SELECT * FROM `travel-sample`.inventory.hotel;
```

Retourne tous les champs, avec un wrapper portant le nom de la collection.

#### 3.3.2 *SELECT avec RAW*

```
SELECT RAW city FROM `travel-sample`.inventory.airport
ORDER BY city LIMIT 3;
```

La clause `RAW` retourne les valeurs **sans imbrication** supplémentaire.

#### 3.3.3 *Accès aux champs imbriqués*

- Utiliser `.` pour accéder aux enfants : `hotel.name`
- Utiliser `[]` pour accéder aux éléments de tableau : `reviews[0].author`

#### 3.3.4 *Pattern matching*

```
WHERE name LIKE "%Medway%"
```

#### 3.3.5 *Métadonnées et clés (META)*

##### 3.3.5.1 *META()*

Accède aux métadonnées d'un document, notamment sa clé :

```
SELECT META(d).id AS docid
FROM `travel-sample`.inventory.hotel AS d
WHERE name LIKE "%Medway%";
```

##### 3.3.5.2 *USE KEYS*

Recherche directe par clé(s) :

```
-- Une clé
SELECT * FROM `travel-sample`.inventory.airport
USE KEYS "airport_1254";

-- Plusieurs clés
SELECT * FROM `travel-sample`.inventory.airport
USE KEYS ["airport_1254", "airport_1255"];
```

### 3.4 Opérateurs sur collections et tableaux

#### 3.4.1 IN / NOT IN

Teste si une valeur est **directement** contenue dans un tableau :

```
WHERE country IN ["United Kingdom", "France"]
```

#### 3.4.2 WITHIN / NOT WITHIN

Teste si une valeur est contenue **directement ou indirectement** :

```
WHERE "Walton Wolf" WITHIN hotel
```

#### 3.4.3 ANY / EVERY

Pour filtrer sur l'existence d'éléments dans des tableaux :

```
-- Au moins un élément satisfait la condition
WHERE ANY s IN schedule SATISFIES s.utc > "00:35" END

-- Tous les éléments satisfont la condition
WHERE EVERY s IN schedule SATISFIES s.utc > "00:35" END
```

#### 3.4.4 ARRAY

Construit un tableau à partir d'éléments filtrés :

```
SELECT RAW ARRAY v.flight FOR v IN schedule
  WHEN v.utc > "19:00" AND v.day = 5
END AS friday_flights
FROM `travel-sample`.inventory.route
```

#### 3.4.5 FIRST

Retourne le premier élément qui satisfait une condition :

```
SELECT FIRST v.flight FOR v IN schedule
  WHEN v.utc > "19:00" AND v.day = 5
END AS evening_flight
```

### 3.5 Agrégation et sous-requêtes

#### 3.5.1 GROUP BY

```
SELECT city, COUNT(DISTINCT name) AS LandmarkCount
FROM `travel-sample`.inventory.landmark
GROUP BY city
ORDER BY LandmarkCount DESC
LIMIT 4;
```

#### 3.5.2 HAVING

Filtre après agrégation :

```
SELECT city, COUNT(DISTINCT name) AS LandmarkCount
FROM `travel-sample`.inventory.landmark
GROUP BY city
HAVING COUNT(DISTINCT name) > 180;
```

#### 3.5.3 Sous-requêtes

Les sous-requêtes peuvent être utilisées dans les clauses WHERE, FROM, SELECT :

```
SELECT t1.city
FROM `travel-sample`.inventory.landmark t1
WHERE t1.city IN (
  SELECT RAW t2.city
  FROM `travel-sample`.inventory.airport t2
)
```

### 3.6 Gestion des valeurs NULL et MISSING

N1QL distingue trois états pour un attribut :

1. **VALUED** : l'attribut a une valeur

```
{"city": "Los Angeles"}
```

2. **NULL** : l'attribut est explicitement nul

```
{"city": null}
```

3. **MISSING** : l'attribut est absent

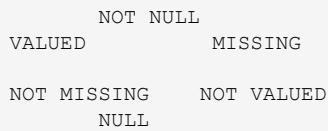
```
{}
```

#### 3.6.1 Tests de présence

```
-- Teste si un attribut a une valeur
WHERE city IS VALUED
WHERE city IS NOT MISSING

-- Teste si un attribut est NULL
WHERE city IS NULL
WHERE city IS NOT NULL
```

#### 3.6.2 Diagramme de Venn des états



### 3.7 Indexation

#### 3.7.1 Types d'index

1. **Index primaire** : sur la clé du document

```
CREATE PRIMARY INDEX travel_primary
ON `travel-sample`.inventory.airline;
```

2. **Index secondaire** : sur n'importe quel champ

```
CREATE INDEX travel_name
ON `travel-sample`.inventory.airline(name);
```

3. **Index composite** : sur plusieurs champs

```
CREATE INDEX travel_info
ON airline(name, id, icao, iata);
```

4. **Index couvrant** : contient tous les champs de la requête

- Évite la récupération des documents depuis les noeuds de données
- Améliore considérablement les performances

#### 3.7.2 Index sur éléments de tableau

```
CREATE INDEX travel_sched
ON `travel-sample`.inventory.route
(DISTINCT ARRAY v.day FOR v IN schedule END);
```

Permet d'accélérer les requêtes du type :

```
SELECT * FROM `travel-sample`.inventory.route
WHERE ANY v IN schedule SATISFIES v.day = 2 END;
```

### 3.7.3 Consultation des index

```
SELECT * FROM system:indexes
WHERE keyspace_id = "airline";
```

### 3.7.4 Avantages et inconvénients

#### Avantages :

- Accès beaucoup plus rapide aux données
- Meilleures performances pour WHERE, ORDER BY, JOIN

#### Inconvénients :

- Espace disque supplémentaire
- Ralentissement des INSERT, UPDATE, DELETE

## 3.8 Jointures et relations

### 3.8.1 ANSI JOIN

Couchbase recommande d'utiliser uniquement les jointures ANSI.

#### Types de jointures :

- INNER JOIN (par défaut)
- LEFT OUTER JOIN
- RIGHT OUTER JOIN

### 3.8.2 Exemple de JOIN

```
SELECT route.airlineid, airline.name,
       route.sourceairport, route.destinationairport
  FROM `travel-sample`.inventory.route
INNER JOIN `travel-sample`.inventory.airline
    ON route.airlineid = META(airline).id
 WHERE route.destinationairport = "SFO"
ORDER BY route.sourceairport;
```

### 3.8.3 Optimisation avec index couvrant

Sans index couvrant : la requête doit récupérer les documents depuis les nœuds de données.

Avec index couvrant :

```
CREATE INDEX idx_route_src_dst_airline
ON `travel-sample`.inventory.route
(sourceairport, destinationairport, airline);
```

La requête peut être résolue uniquement depuis l'index, sans accès aux documents.

### 3.8.4 NEST

Similaire à JOIN, mais imbrique les résultats dans un tableau au lieu de créer une ligne par correspondance.

#### 3.8.4.1 Exemple NEST vs JOIN

Avec JOIN :

```
SELECT a.callsign, r.destinationairport
  FROM `travel-sample`.inventory.airline a
JOIN `travel-sample`.inventory.route r
    ON r.airlineid = META(a).id
   LIMIT 3;
```

Résultat : une ligne par destination

```
[{"callsign": "MILE-AIR", "destinationairport": "HKB"}, {"callsign": "MILE-AIR", "destinationairport": "FAI"}, {"callsign": "SASQUATCH", "destinationairport": "BNA"}]
```

## Avec NEST :

```
SELECT a.callsign,
  ARRAY ro.destinationairport FOR ro IN r END AS dests
FROM `travel-sample`.inventory.airline a
NEST `travel-sample`.inventory.route r
  ON r.airlineid = META(a).id
LIMIT 3;
```

Résultat : une ligne par compagnie avec toutes ses destinations

```
[{"callsign": "MILE-AIR", "dests": ["HKB", "FAI"]}, {"callsign": "SASQUATCH", "dests": ["BNA", "AHN", ...]}]
```

### 3.8.5 UNNEST

Aplatit un tableau imbriqué en créant une ligne par élément.

#### 3.8.5.1 Exemple UNNEST

```
SELECT sched, META(r).id AS route_id
FROM `travel-sample`.inventory.route r
UNNEST r.schedule sched
WHERE sched.day = 1;
```

## Sans UNNEST :

```
{ "route_id": "route_10000", "sched": [ {"day": 1, "flight": "AF356", "utc": "12:40:00"}, {"day": 1, "flight": "AF480", "utc": "08:58:00"} ] }
```

## Avec UNNEST :

```
[ {"route_id": "route_10000", "sched": {"day": 1, "flight": "AF356", "utc": "12:40:00"}}, {"route_id": "route_10000", "sched": {"day": 1, "flight": "AF480", "utc": "08:58:00"}} ]
```

## 3.9 Fonctions sur les tableaux

### 3.9.1 ARRAY\_AGG

Agrège des valeurs dans un tableau :

```
SELECT city, ARRAY_AGG(name) AS hotels
FROM `travel-sample`.inventory.hotel
WHERE city IS NOT NULL
GROUP BY city
LIMIT 2;
```

Résultat :

```
[ { "city": "Aberdeen", "hotels": ["Youth Hostel Aberdeen", "Royal Hotel Aberdeen"] } ]
```

```

    }
]
```

### 3.9.2 ARRAY\_COUNT

Compte les éléments non-NUL d'un tableau :

```

SELECT h.name,
  ARRAY_COUNT(
    ARRAY r.ratings.Cleanliness FOR r IN h.reviews
    WHEN r.ratings.Cleanliness > 3 END
  ) AS high_cleanliness_reviews
FROM `travel-sample`.inventory.hotel h

```

**Note :** `ARRAY_LENGTH` est similaire mais compte aussi les NULL.

### 3.9.3 ARRAY\_MAX / ARRAY\_MIN

Retourne la valeur maximale/minimale d'un tableau :

```

SELECT h.name,
  ARRAY_COUNT(h.reviews) AS numberOfReviews,
  ARRAY_MAX(service_ratings) - ARRAY_MIN(service_ratings)
  AS ServiceDeviation
FROM hotel AS h
LET service_ratings = ARRAY r.ratings.Service FOR r IN h.reviews END
WHERE ARRAY_COUNT(h.reviews) > 3

```

### 3.9.4 Autres fonctions utiles

Voir la documentation complète : <https://docs.couchbase.com/cloud/n1ql/n1ql-language-reference/arrayfun.html>

## 3.10 Aspects avancés

### 3.10.1 Requêtes paramétrées

N1QL supporte les requêtes paramétrées pour :

- Améliorer la sécurité (prévention des injections SQL)
- Permettre la réutilisation des plans d'exécution

Documentation : <https://docs.couchbase.com/server/current/n1ql/n1ql-languagereference/prepare.html>

### 3.10.2 Portée des alias

Clause	Les alias créés peuvent être référencés dans
WITH	N'importe où dans le bloc de requête
FROM	N'importe où dans le bloc de requête
LET	N'importe où dans le bloc de requête
LETTING	HAVING, SELECT, et ORDER BY
SELECT	SELECT et ORDER BY
FOR	L'expression de collection locale

## 3.11 Bonnes pratiques

1. Toujours utiliser **EXPLAIN** pour comprendre le plan d'exécution
2. Créer des **index appropriés** pour les requêtes fréquentes
3. Privilégier les **index couvrants** quand possible
4. Utiliser **RAW** pour éviter l'imbrication inutile
5. Utiliser **DISTINCT** pour éliminer les doublons
6. Préférer **ANSI JOIN** aux autres types de jointures
7. Utiliser **NEST** quand on veut des résultats imbriqués
8. Utiliser **UNNEST** pour aplatisir les tableaux