

# Modèle clé-valeur

- **Principes**
- **Redis**
- **Dynamo**
- **HSTORE sous PostgreSQL**

# Base de données Clé-Valeur

- Similaire à table de hachage : Stockage des paires clé-valeur, identifiables par leur clé
- Similaire à une table relationnelle avec deux colonnes
- Très bonnes performances grâce à l'indexation sur la clé
- Le plus simple espace de stockage de type NoSQL
- 3 opérations principales : Récupérer/définir une valeur pour une clé, supprimer une clé

# Clé-Valeur : cas d'utilisation

## ■ Cas d'utilisation :

- Stockage des informations d'une session pour un site web
- Profils et préférences d'un utilisateur
- Panier d'achats sur un site de e-commerce : Stockage du panier d'achats actuel d'un utilisateur

## ■ Non applicable aux:

- Données liées à différentes clés
- Requêtes sur les valeurs

# Principaux moteurs Clé - Valeur

include secondary database models

68 systems in ranking, February 2023

Rank			DBMS	Database Model	Score		
Feb 2023	Jan 2023	Feb 2022			Feb 2023	Jan 2023	Feb 2022
1.	1.	1.	Redis	Key-value, Multi-model	173.83	-3.72	-1.96
2.	2.	2.	Amazon DynamoDB	Multi-model	79.69	-1.87	-0.67
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	36.51	-1.45	-3.45
4.	4.	4.	Memcached	Key-value	23.17	-0.58	-2.60
5.	5.	6.	Hazelcast	Key-value, Multi-model	9.11	-0.23	-0.39
6.	6.	5.	etcd	Key-value	8.51	+0.01	-3.19
7.	7.	10.	Aerospike	Multi-model	6.56	+0.09	+0.95
8.	8.	9.	Ehcache	Key-value	6.04	+0.04	-0.34
9.	10.	14.	Google Cloud Bigtable	Multi-model	5.92	+0.45	+1.75
10.	9.	8.	Ignite	Multi-model	5.50	-0.40	-1.00
11.	12.	11.	ArangoDB	Multi-model	5.29	+0.22	-0.11
12.	11.	7.	Riak KV	Key-value	5.14	-0.10	-1.78

# Redis (*REmote DIctionary Server*)

- Projet open-source écrit en ANSI C
- Lancé en 2009, puis développé par *Vmware* à partir de mars 2010 et par Redis Labs depuis juin 2015
- Similaire à une gigantesque *Hash Map* ou un dictionnaire Python
- Stocke les données en mémoire (*in memory DB*) - peut également servir de cache de données
- Possibilité de stocker des données sur disque (*snapshot*) : Au démarrage du service, dernier snapshot mis en mémoire
- Souvent utilisé pour augmenter la vitesse de réponse des sites web (Facebook, Twitter, Wikipedia) créés à partir de bases de données

# Redis : liens utiles

- Site web officiel Redis : <https://redis.io/>
- Console et un tutoriel interactif : <http://try.redis.io/>
- Liste des commandes Redis : <https://redis.io/commands/>
- Cours en ligne : <http://b3d.bdpedia.fr/redis.html>
- Vidéo sur les bases de Redis et sur son installation : <https://grafikart.fr/tutoriels/redis>
- Sur AWS : <https://aws.amazon.com/fr/redis/>
- Sur Microsoft Azure : <https://azure.microsoft.com/fr-fr/services/cache/>

# Redis : types de données

- Clé unique, *binary-safe*, pouvant aller jusqu'à 512MB – d'une chaîne de caractères, de préférence ASCII à une image JPEG
- Possibilité de donner une durée de vie à une clé
- Plusieurs types de valeurs possibles: chaîne (*string*), liste (*list*), ensemble non ordonné (*set*), ensemble ordonné (*Zset*) et table de hachage (*hash*),



# Redis : valeurs de type chaîne

Commandes pour manipuler les chaînes :

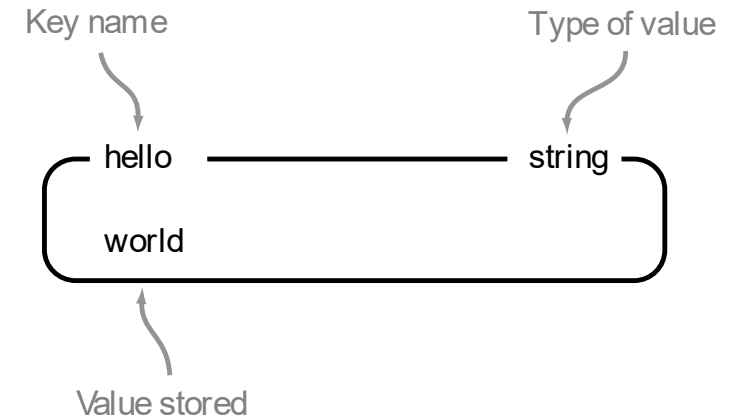
- SET ajoute une nouvelle chaîne
- GET récupère la valeur associée à une clé
- DEL supprime une clé du *store*

```
> SET ma:clé "ma valeur"
```

```
OK
```

```
> GET ma:clé
```

```
"ma valeur"
```



## Passenger information

```
'passenger:0:name' = 'Bryan Richard'  
'passenger:0:seat' = '12A'  
'passenger:0:flightCode' = 'U23211'
```

```
'passenger:1:name' = 'Charlie Dixon'  
'passenger:1:seat' = '12C'  
'passenger:1:flightCode' = 'U23211'
```

```
'passenger:2:name' = 'Melanie Brown'  
'passenger:2:seat' = '13B'  
'passenger:2:flightCode' = 'U23211'
```

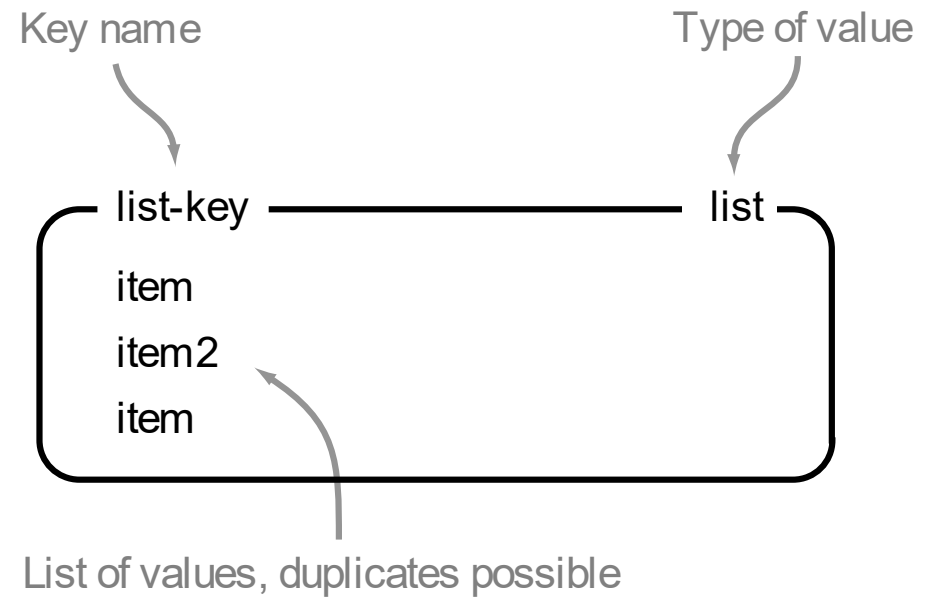
Image reprise de <https://techannotation.wordpress.com/2018/02/15/data-model-in-redis/> et <https://redis.com/fr/redis-enterprise/structures-de-donnees/>



# Redis : listes

Commandes pour manipuler les listes :

- LPUSH ajoute une entrée en début de la liste
- LPOP supprime le premier élément de la liste
- RPUSH ajoute une entrée en fin de la liste
- RPOP supprime de dernier élément de la liste
- LRANGE extrait une sous-liste depuis une liste



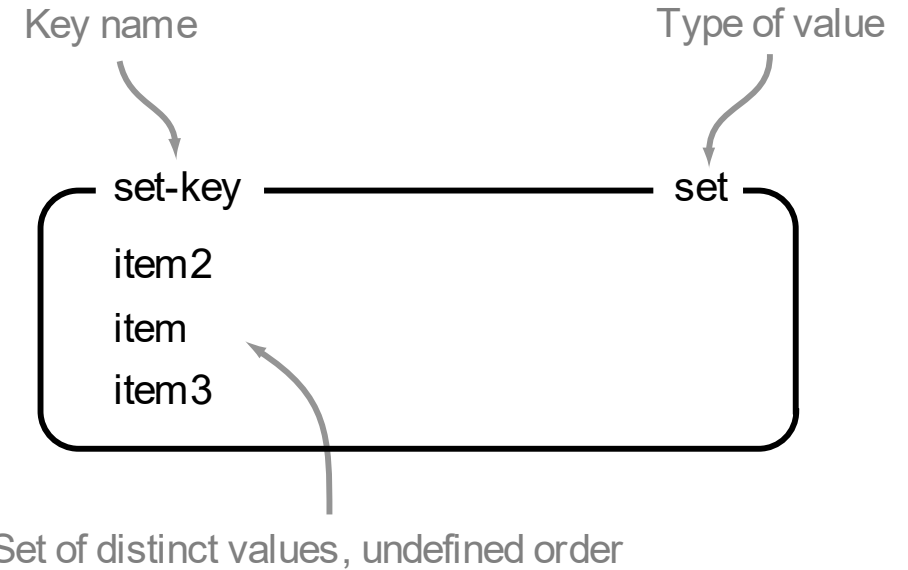
L

```
redis 127.0.0.1:6379> LPUSH LastWebVisitors Webuser:1
(integer) 1
redis 127.0.0.1:6379> LPUSH LastWebVisitors Webuser:2
(integer) 2
redis 127.0.0.1:6379> LPUSH LastWebVisitors Webuser:Temp
(integer) 3
redis 127.0.0.1:6379> LRANGE LastWebVisitors 0 -1
1) "Webuser:Temp"
2) "Webuser:2"
3) "Webuser:1"
```

# Redis : ensembles non ordonnés

Commandes pour manipuler les ensembles :

- SADD ajoute un élément dans l'ensemble
- SMEMBERS renvoie les éléments de l'ensemble
- SISMEMBER teste l'appartenance
- SREM supprime un élément
- SINTER, SUNION, SDIFF opérations ensemblistes



```
redis 127.0.0.1:6379> SADD Webuser:HasGMail Webuser:3
(integer) 1
redis 127.0.0.1:6379> SADD Webuser:HasGMail Webuser:2
(integer) 1
redis 127.0.0.1:6379> SADD Webuser:HasGMail Webuser:5
(integer) 1
redis 127.0.0.1:6379> SMEMBERS Webuser:HasGMail
1) "Webuser:2"
2) "Webuser:5"
3) "Webuser:3"
```

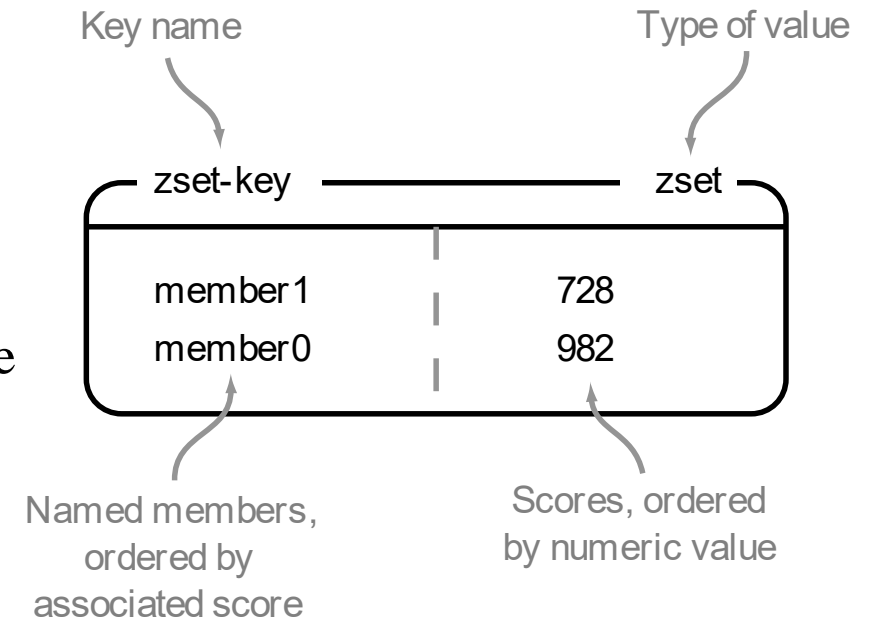
```
redis 127.0.0.1:6379> SMEMBERS Webuser:IsFrench
1) "Webuser:2"
2) "Webuser:9"
3) "Webuser:7"
4) "Webuser:5"
```

```
# Donne moi les utilisateurs Français avec GMail
redis 127.0.0.1:6379> SINTER Webuser:IsFrench Webuser:HasGMail
1) "Webuser:2"
2) "Webuser:5"
```

# Redis : ensembles ordonnés

Commandes pour manipuler les ZSET :

- ZADD ajoute un élément dans l'ensemble avec un certain score
- ZRANGE renvoie les éléments de l'ensemble
- ZRANGEBYSCORE renvoie les éléments de l'ensemble par score
- ZREM supprime un élément



```
redis rio.zaptravel.com:6379> ZADD DealsFranceByPrice 59 Trip:Lyon
(integer) 1
```

```
redis rio.zaptravel.com:6379> ZADD DealsFranceByPrice 259 Trip:Venise
(integer) 1
```

```
redis rio.zaptravel.com:6379> ZADD DealsFranceByPrice 130 Trip:Bordeaux
(integer) 1
```

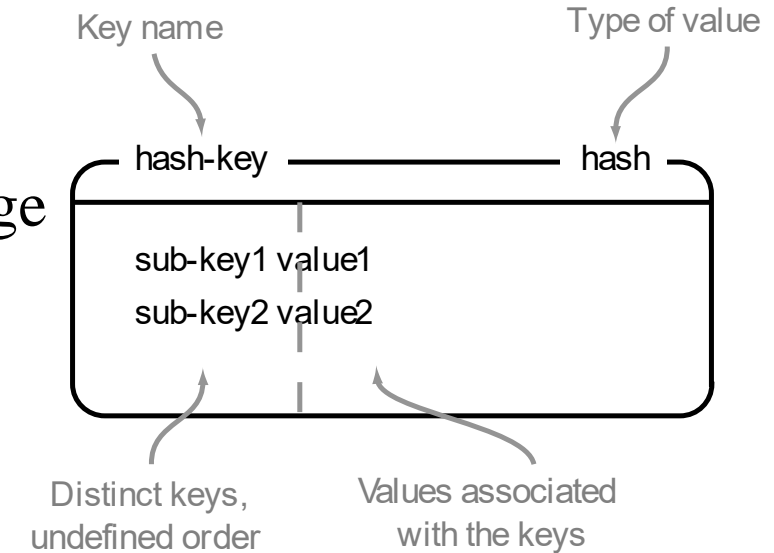
```
redis rio.zaptravel.com:6379> ZADD DealsFranceByPrice 990 Trip:Paris
(integer) 1
```

```
redis rio.zaptravel.com:6379> ZRANGEBYSCORE DealsFranceByPrice 0 200 WITHSCORES LIMIT 0 10
1) "Trip:Lyon"
2) "59"
3) "Trip:Bordeaux"
4) "130"
```

# Redis : table de hachage

Commandes pour manipuler les tables de hachage :

- HSET ajoute une entrée dans la table de hachage d'une clé
- HVALS récupère la table de hachage complète d'une clé
- HGET récupère la valeur d'une entrée d'une table de hachage
- HGETALL récupère les données associées à une clé
- HDEL supprime une entrée d'une table de hachage



```
> HSET utilisateur:1 nom "Thomas Bourgier"  
> HSET utilisateur:1 email "t.bourgier@exemple.com"  
> HSET utilisateur:1 mot_de_passe "1234"  
> HSET utilisateur:2 nom "Myriam Fois" email "m.fois@exemple.com"
```

# Redis: exemple de table de hachage

```
> HGET ALL utilisateur:1
```

```
1) "nom"
```

```
2) "Thomas Bourgier"
```

```
3) "email"
```

```
4) "t.bourgier@exemple.com"
```

```
5) "mot_de_passe"
```

```
6) "1234"
```

```
> HGET utilisateur:1 nom
```

```
"Thomas Bourgier"
```

```
> HSET utilisateur:1 visites 10
```

```
> HINCRBY utilisateur:1 visites 1
```

```
(integer) 11
```

# Redis : opération arithmétique

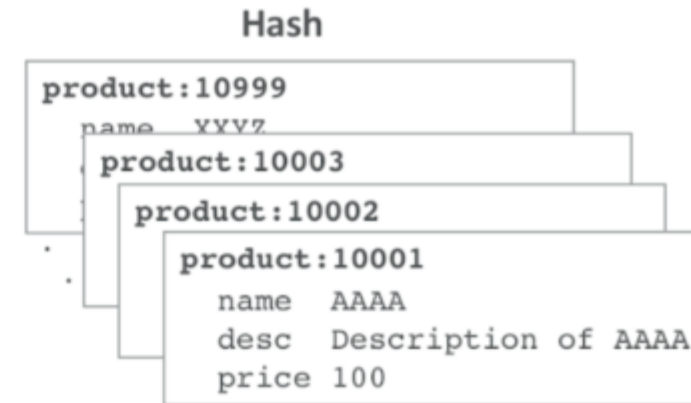
- Possibilité d'appliquer des opérations arithmétiques sur des entiers :
  - > SET mon:entier 1
  - OK
  - > INCR mon:entier
  - (integer) 2
  - > INCRBY mon:entier 10
  - (integer) 12
- Garantie par Redis que plusieurs appels simultanés à la fonction INCR seront exécutés de manière séquentielle

# Redis : Relationnel / Clé-valeur (exemple 1)

Transformer une base de données relationnelle en base clé-valeur

Table  
Products

ID	Name	Description	Price
10001	AAAA	Description of AAAA	100
10002	BBBB	Description of BBBB	200
10003	CCCC	Description of CCCC	200
.	.	.	.
.	.	.	.
10999	XXYZ	Description of XXYZ	500



Sorted Set

product_list	
10001	product:10001
10002	product:10002
10003	product:10003
.	.
.	.
10999	product:10999

Sorted Set

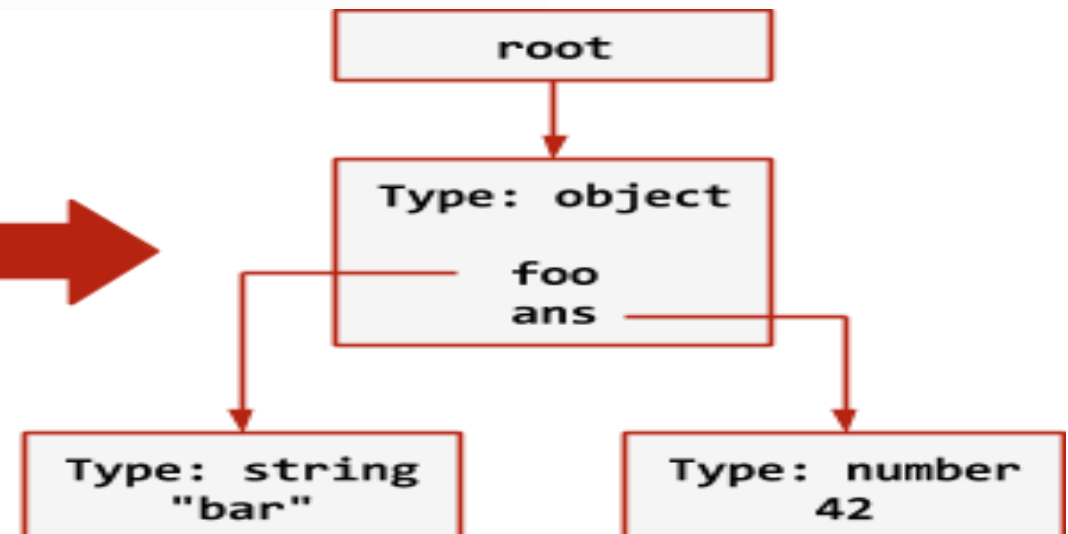
product_price	
100	product:10001
200	product:10002
200	product:10003
.	.
.	.
500	product:10999

# Redis : extension JSON

- Possibilité de gérer des documents JSON : module ReJSON

```
127.0.0.1:6379> JSON.SET object . '{"foo": "bar", "ans": 42}'  
OK  
127.0.0.1:6379> JSON.GET object  
"{\"foo\": \"bar\", \"ans\": 42}"  
127.0.0.1:6379> JSON.GET object .ans  
"42"
```

```
{  
  "foo": "bar",  
  "ans": 42  
}
```

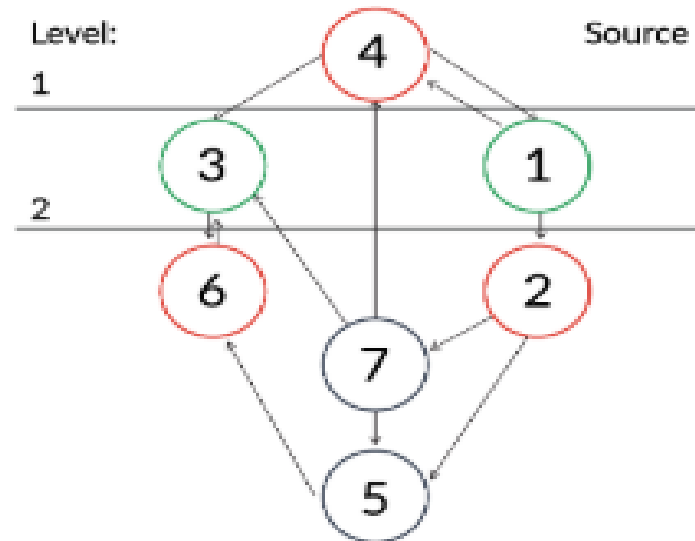




# Redis : extension Graphe

- Possibilité de gérer des graphes : module RedisGraph

	Source Nodes						
	1	2	3	4	5	6	7
1	0	0	0	1	0	0	0
2	1	0	0	0	0	0	0
3	0	0	0	1	0	1	1
4	1	0	0	0	0	0	1
5	0	1	0	0	0	0	1
6	0	0	1	0	1	0	0
7	0	1	0	0	0	0	0

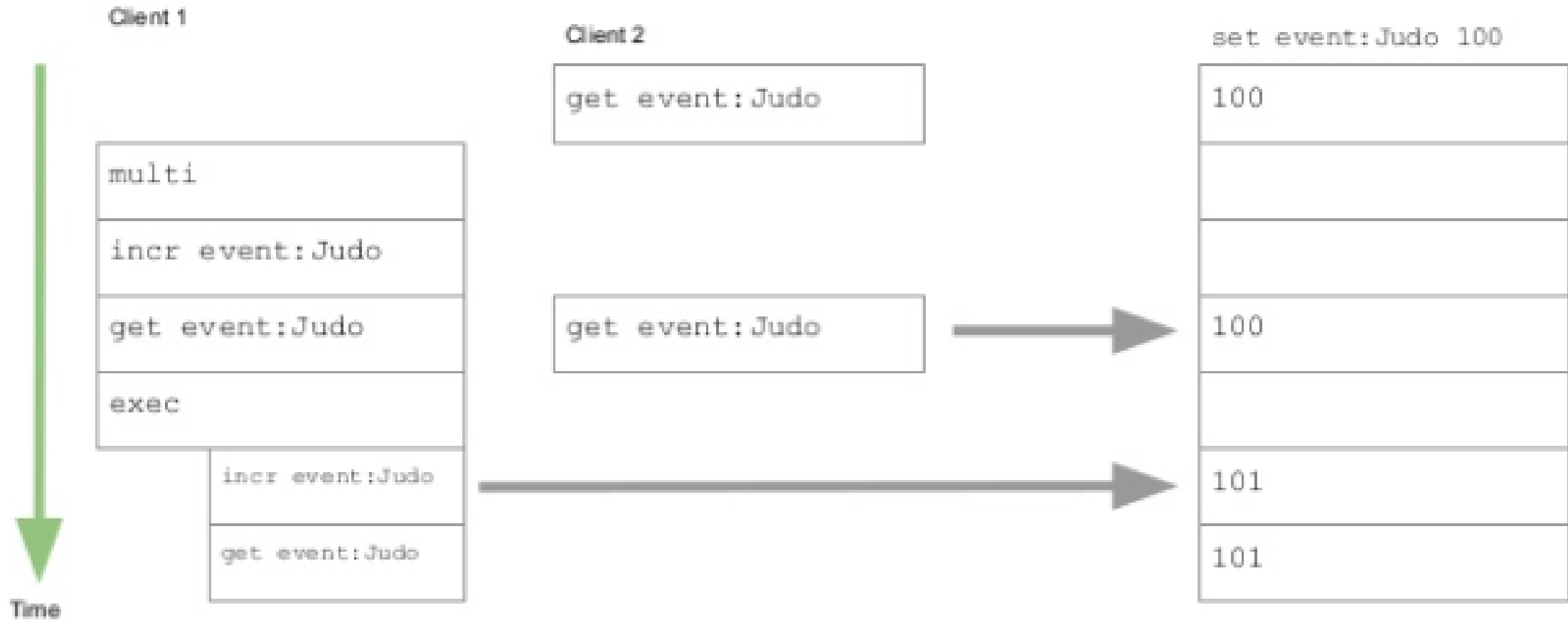


```
$ redis-cli
127.0.0.1:6379> GRAPH.QUERY MotoGP "CREATE (:Rider {name:'Valentino Rossi'})-[:rides]->(:Team {name:'Yamaha'}), (:Rider {name:'Dani Pedrosa'})-[:rides]->(:Team {name:'Honda'}), (:Rider {name:'Andrea Dovizioso'})-[:rides]->(:Team {name:'Ducati'})"
1) 1) Labels added: 2
2) Nodes created: 6
3) Properties set: 6
4) Relationships created: 3
5) "Query internal execution time: 0.399000 milliseconds"
```

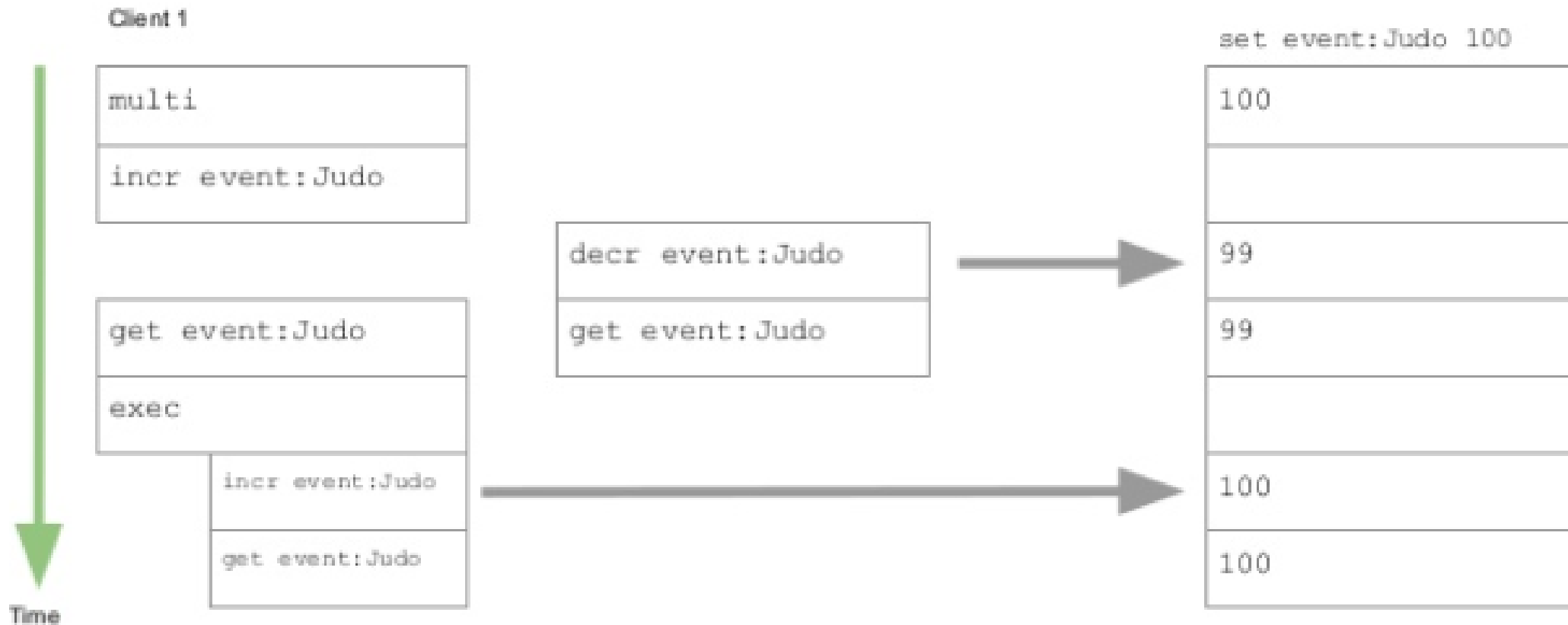
# Redis : « transactions »

- Transactions dans Redis non équivalentes aux transactions au sens bases de données SQL
- “Transaction” Redis : bloc d’instructions placées entre les commandes MULTI and EXEC (ou DISCARD pour *cancel*) = bloc atomique
- Commande MULTI  $\Rightarrow$  stockage des instructions dans une queue –
- Exécution des instructions stockées dans la queue (QUEUE) lors de la commande EXEC.
- Commande WATCH / UNWATCH  $\Rightarrow$  verrouillage optimiste.

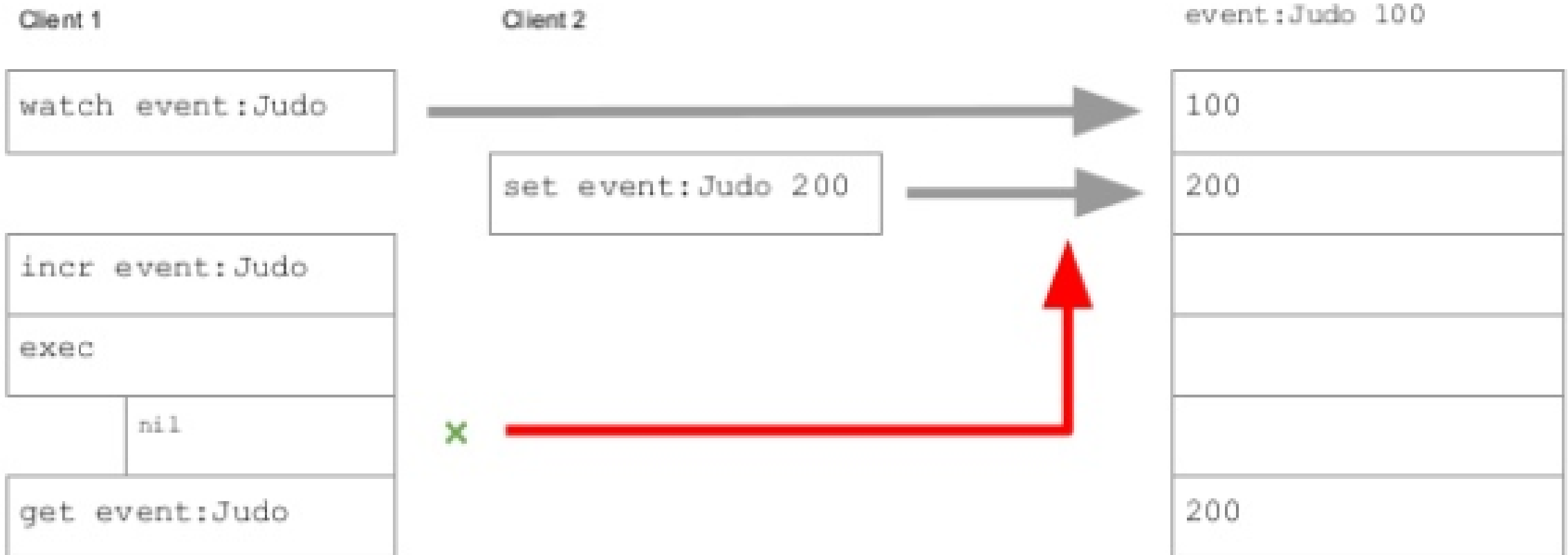
# Redis : instruction `multi exec` => « atomicité »



# Redis : instruction multi exec => pas de « verrouillage »



# Redis : gestion optimiste de la concurrence

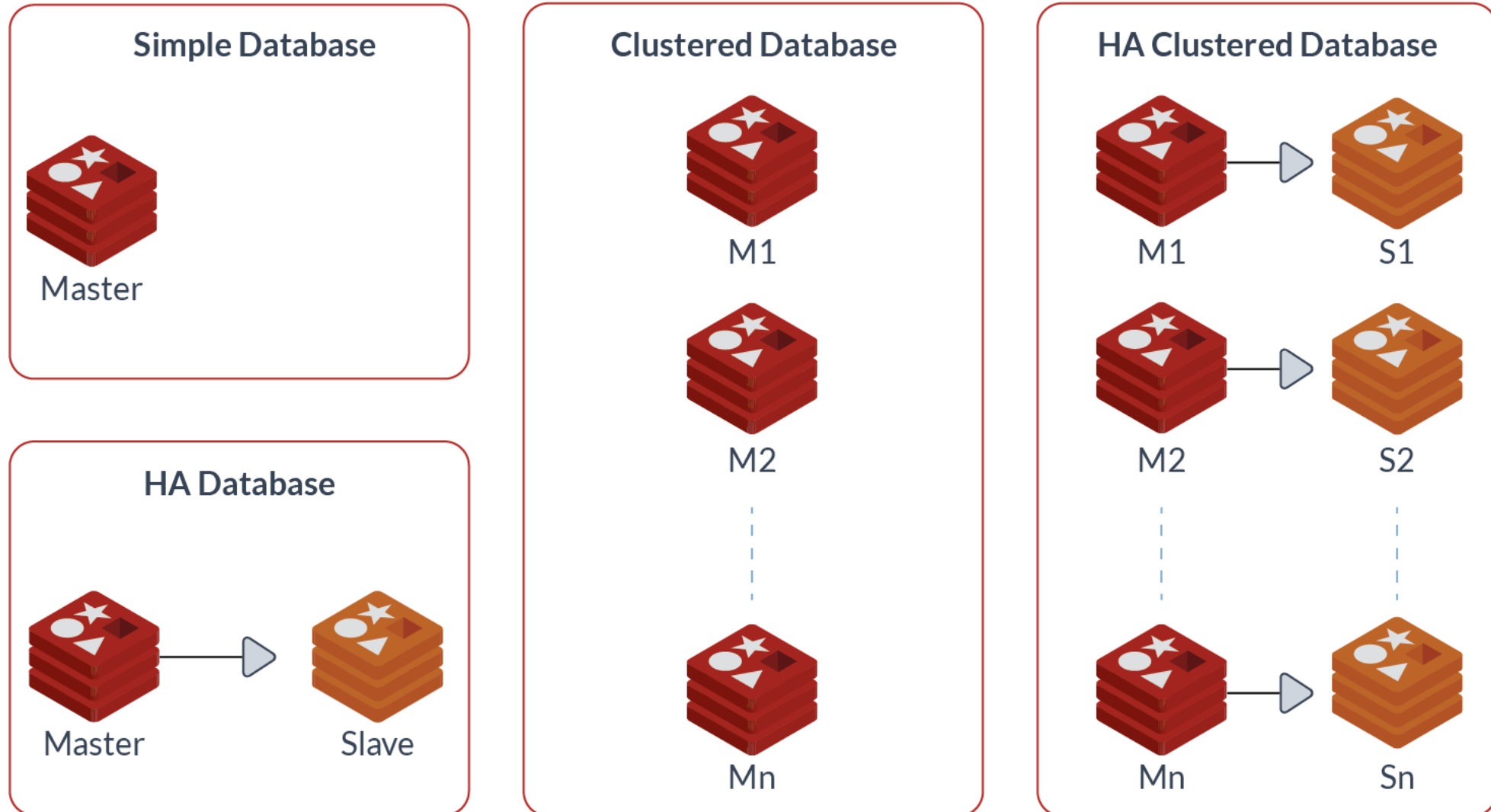


# Redis : exemple d'utilisation de MULTI ET WATCH

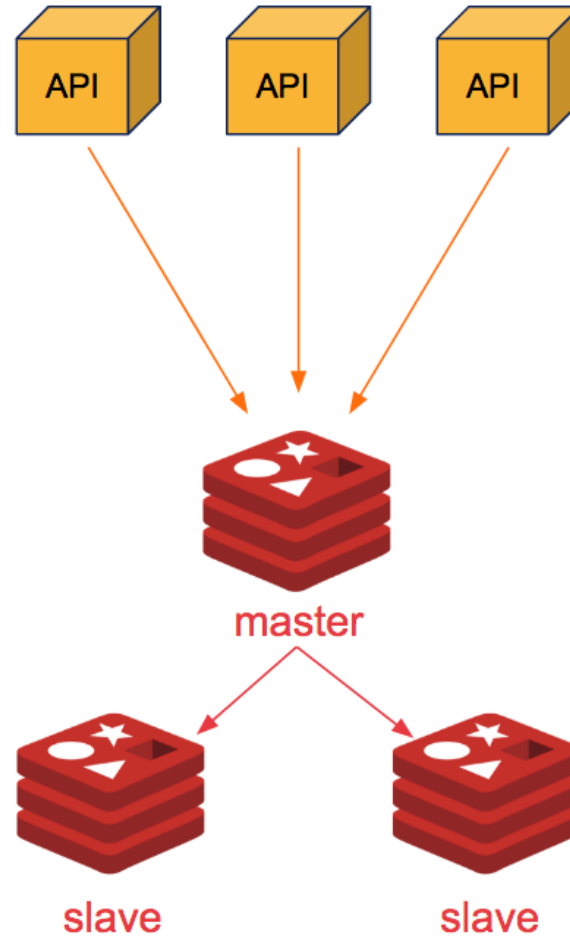
TERMINAL 1	TERMINAL 2
<pre># redis-cli 127.0.0.1:6379&gt; WATCH sample_key OK 127.0.0.1:6379&gt; MULTI OK 127.0.0.1:6379&gt; INCRBY sample_key 100 QUEUED</pre>	
<pre># redis-cli 127.0.0.1:6379&gt; WATCH sample_key OK 127.0.0.1:6379&gt; MULTI OK 127.0.0.1:6379&gt; INCRBY sample_key 100 QUEUED 127.0.0.1:6379&gt; EXEC (nil) 127.0.0.1:6379&gt; █</pre>	<pre>127.0.0.1:6379&gt; INCR sample_key (integer) 112 127.0.0.1:6379&gt; █</pre>

1. `sample_key` a pour valeur 111
2. Dans le Terminal 1 :
  - a. Exécution de `WATCH`
  - b. Exécution de `MULTI`
  - c. Stockage dans la `QUEUE` de l'incrément de `sample_key` dont la valeur passerait à 211
3. Dans le Terminal 2 : exécution de l'incrément de `sample_key` dont la valeur passe à 112
4. Dans le Terminal 1 : Exécution de la commande `EXEC` qui crée `(nil)` car la valeur de 211 ne peut pas être stockée

# Redis : Système de réplication basé sur différentes architectures maître-esclave



# Redis : 1 master et 3 réplicas





# Redis : communication et synchronisation entre le maître et ses esclaves

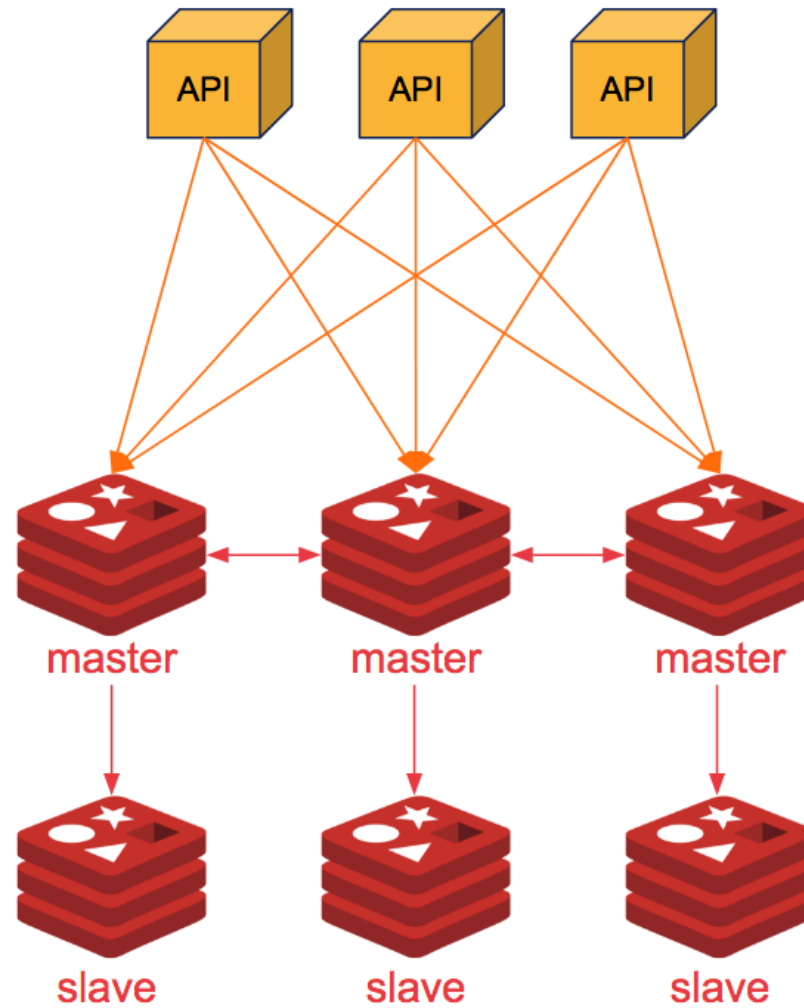
- Envoi d'un flux de commandes du maître aux esclaves pour répliquer les opérations effectuées sur le jeu de données du maître, quasiment en temps réel.
- En cas de rupture de la connexion entre maître et esclave est rompue ou de connexion trop longue : reconnexion de l'esclave, tentative de synchronisation partielle afin de ne récupérer que la parties du flux de commandes qu'il n'a pas pu obtenir lors de la coupure.
- En cas d'impossibilité de synchronisation partielle : demande de synchronisation complète par l'esclave demande une, une sauvegarde de toutes ses données par le maître, envoie à l'esclave, et reprise de l'envoi du flux de commande comme précédemment.

# Redis :

## réplication, sauvegardes périodiques/incrémentales et système de surveillance Sentinel

- Réplication asynchrone et donc non bloquante pour le maître et dans la plupart des cas elle est non bloquante également pour les esclaves.
- Possibilité pour les esclaves d'avoir des répliquions en cascade.
- Possibilité de combiner la réplication avec les sauvegardes périodiques (RDB) et les sauvegardes incrémentales (AOF).
- Recommandation de Redis : activer les deux modes de sauvegarde sur les maîtres et les esclaves afin d'assurer la meilleure reprise sur panne possible.
- Système Sentinel : pour surveiller les instances de Redis, et, en cas de panne d'un serveur maître, élire un nouveau serveur maître parmi les esclaves encore disponibles

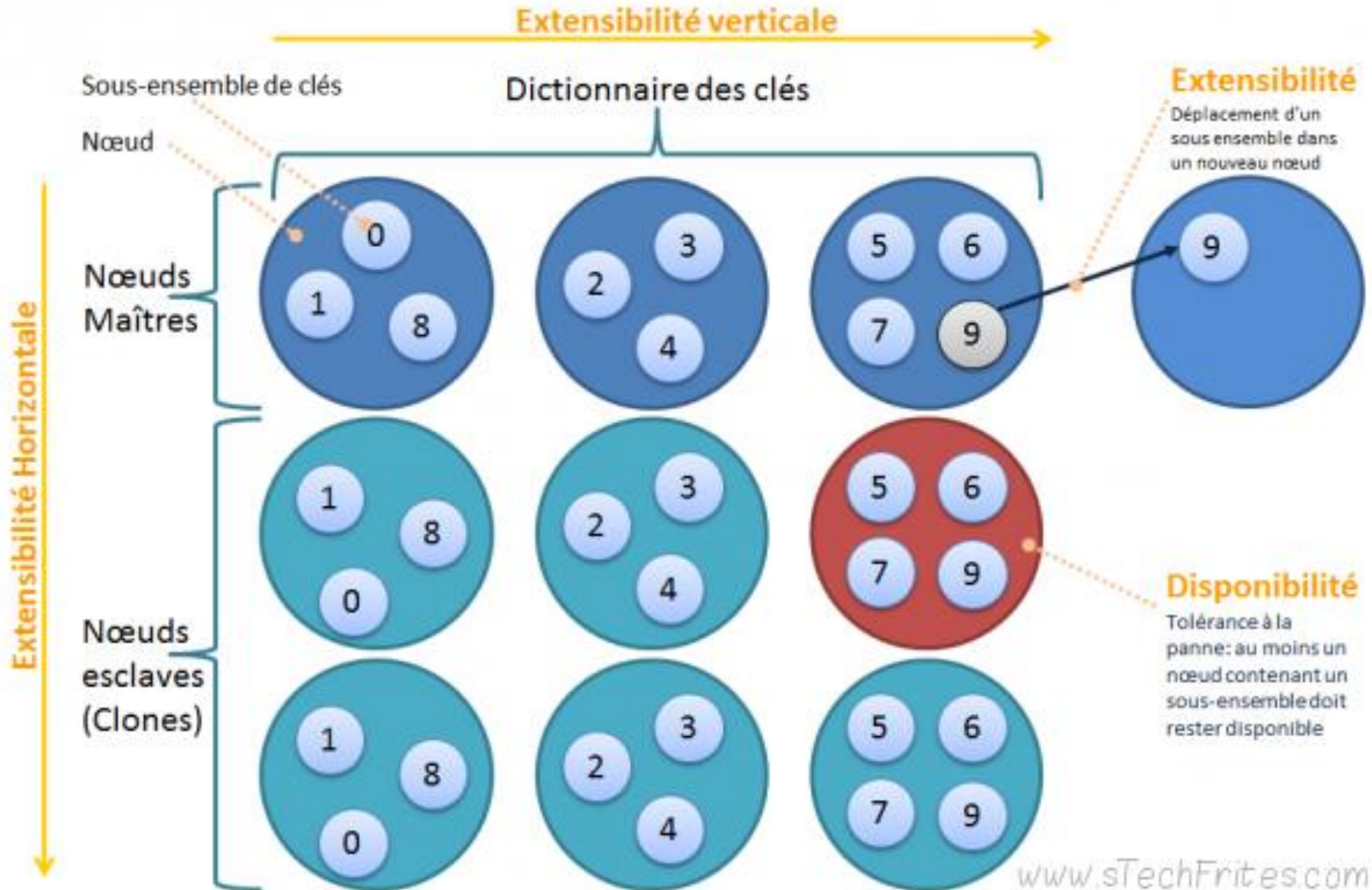
# Redis : Plusieurs maîtres



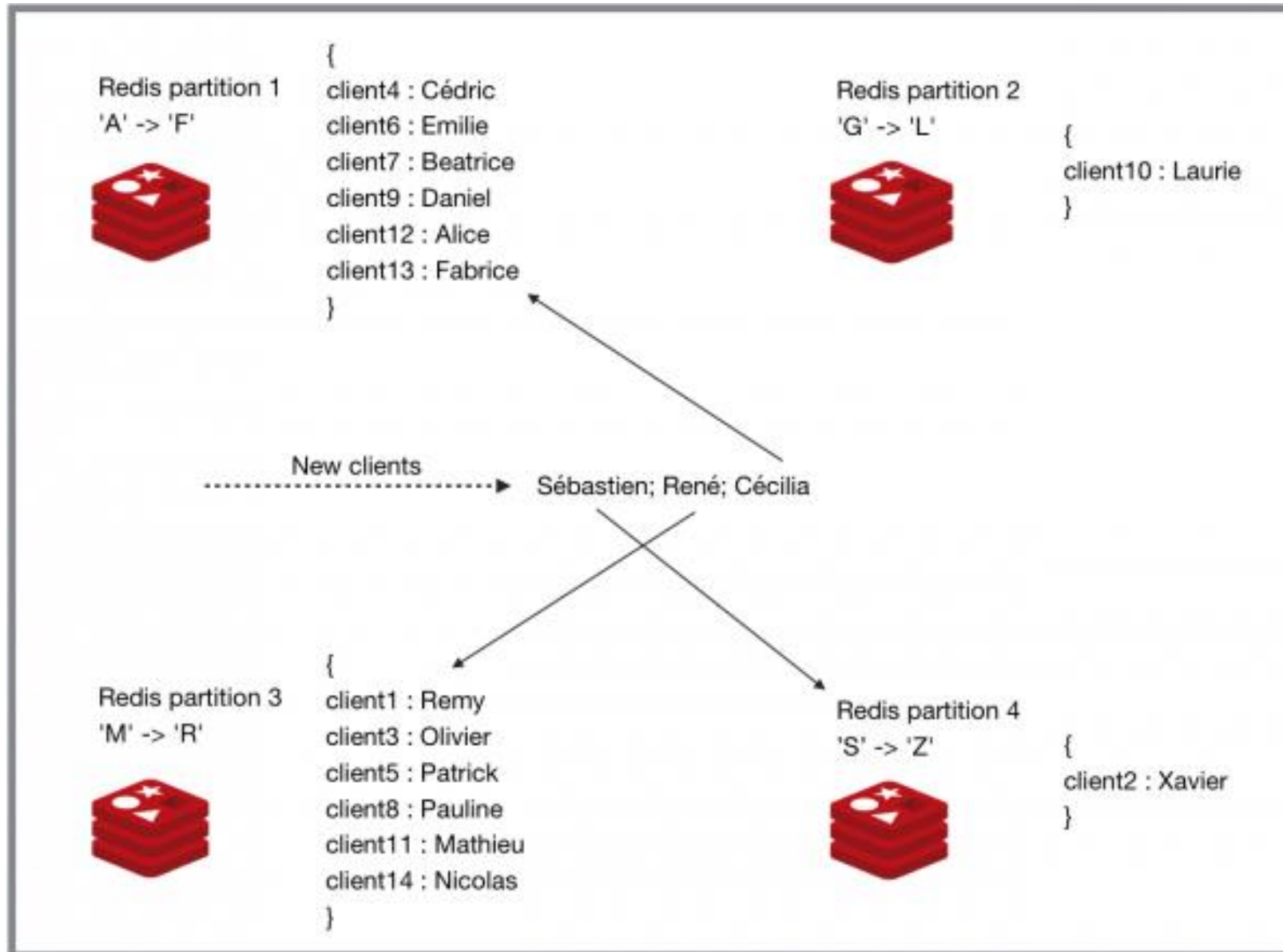
# Redis : Partitionnement horizontal / vertical

- Partitionnement horizontal : Distribution des clés à travers les différentes instances de Redis (aussi connu sous le nom de '*Sharding*') – la plus utilisée
- Partitionnement vertical : Distribution des valeurs des clés à travers les différentes instances de Redis

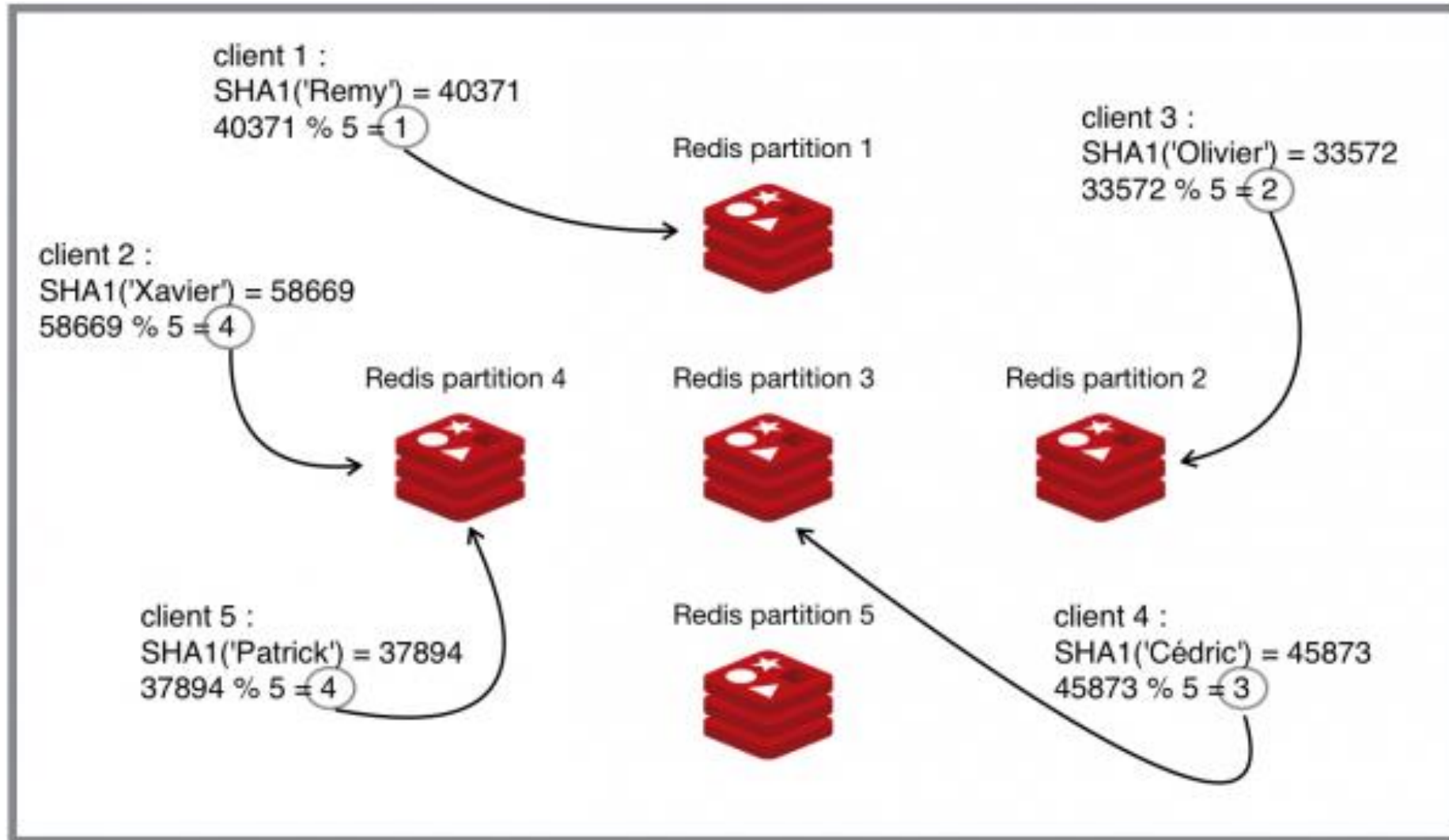
# Redis : Partitionnement



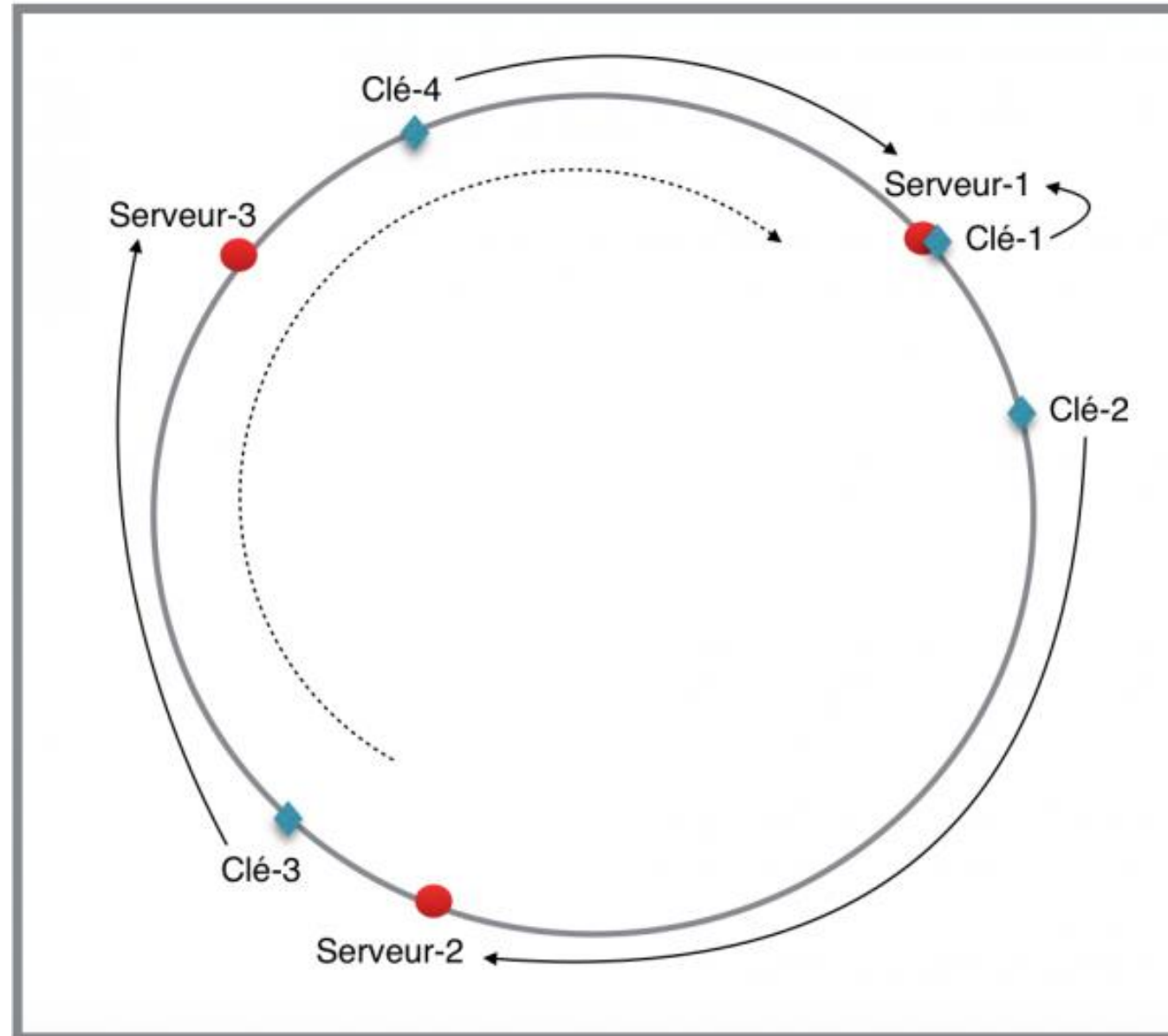
# Redis : *Range Partitioning*



# Redis : Hash Partitioning



# Redis : *Consistent hashing*





# Hachage cohérent (*Consistent hashing*) : pourquoi ?

- Hachage : basé sur une fonction  $h()$  qui distribue les valeurs de clé vers un intervalle  $[0, n-1]$ ,  $n$  correspondant au nombre de fragments

 modification de la fonction rend invalide la distribution existante

- **Hachage cohérent** : maintien de la même fonction tout en adaptant la règle d'affectation d'une donnée à un serveur selon l'évolution (ajout / suppression) de la grappe
- Principe du hachage cohérent : considérer dès le départ un intervalle immuable  $D = [0, n-1]$  pour le domaine d'arrivée de la fonction de hachage, où  $n$  est choisi assez grand pour réduire le nombre de collisions.

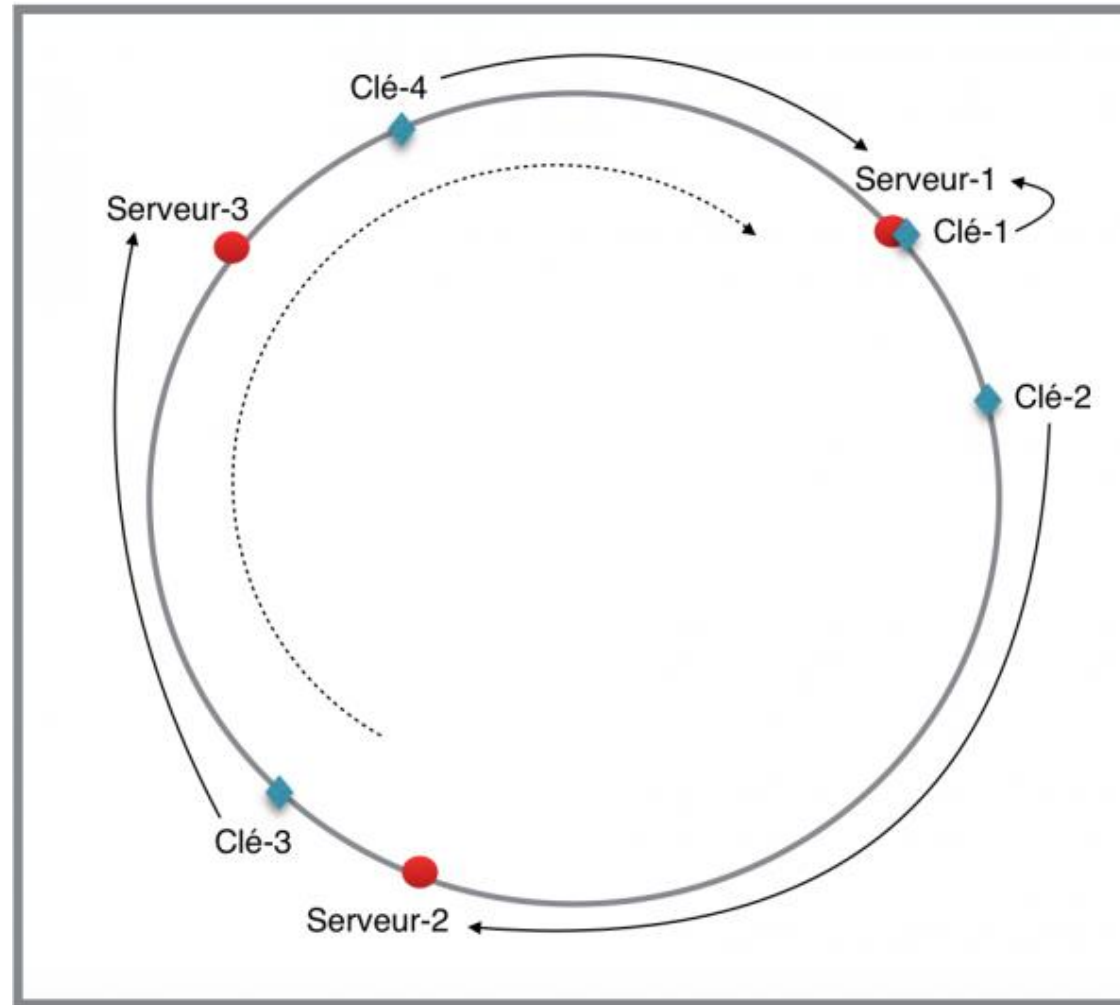
# Hachage cohérent (*Consistent hashing*) : principe

- Objectif : répartir les données de manière la plus équitable possible
- Système réparti vu comme un anneau (*hash ring*), où chaque nœud occupe une position dans l'anneau et se voit attribuer une valeur de hachage (rangée de valeurs hébergées par le nœud)
- *Hash code* calculé à partir de la clé et associé au nœud dont la valeur de hachage est immédiatement supérieure
- Chaque nœud contient toutes les clés inférieures à sa valeur de hachage et supérieures à celle de nœud précédent

# Hachage cohérent (*Consistent hashing*) : exemple

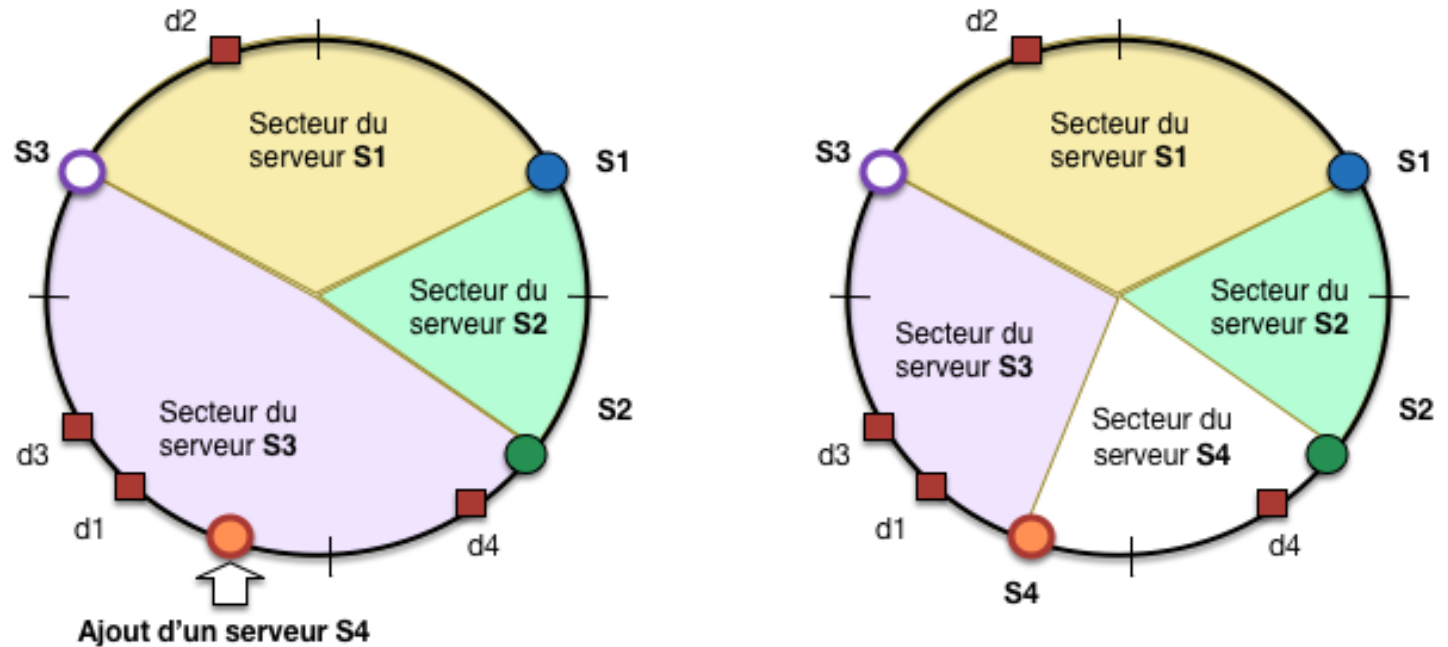
Exemple : 3 serveurs, 4 clés et une fonction de hachage MD5

MD5("serveur-1") = 3  
MD5("serveur-2") = 7  
MD5("serveur-3") = 11  
MD5("clé-1") = 3  
MD5("clé-2") = 4  
MD5("clé-3") = 8  
MD5("clé-4") = 12



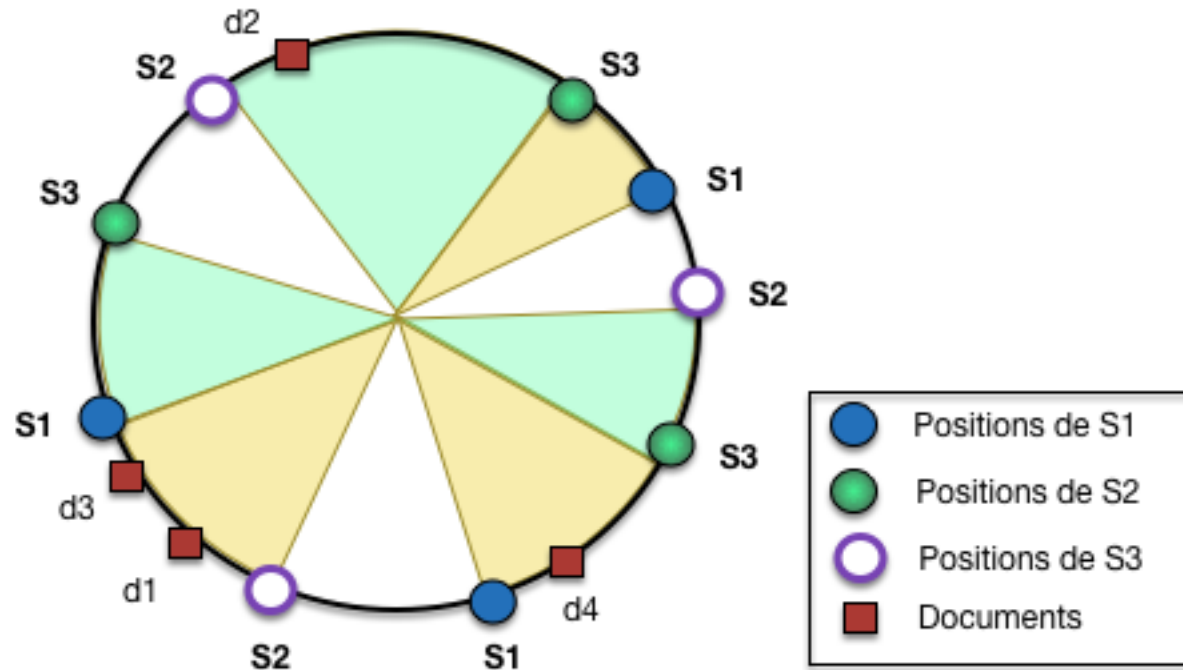
# Hachage cohérent (*Consistent hashing*) : 1 position par serveur sur l'anneau

- Interprétation du domaine  $D = [0, n-1]$  comme un anneau parcouru dans le sens des aiguilles d'une montre
- La fonction de hachage associe donc chaque serveur de la grappe à une position sur l'anneau



# Hachage cohérent (*Consistent hashing*) : plusieurs positions par serveur sur l'anneau

- Possibilité pour chaque serveur d'avoir plusieurs positions sur l'anneau, pour multiplier les arcs de cercles et, par un effet d'uniformisation, de rendre leurs tailles comparables

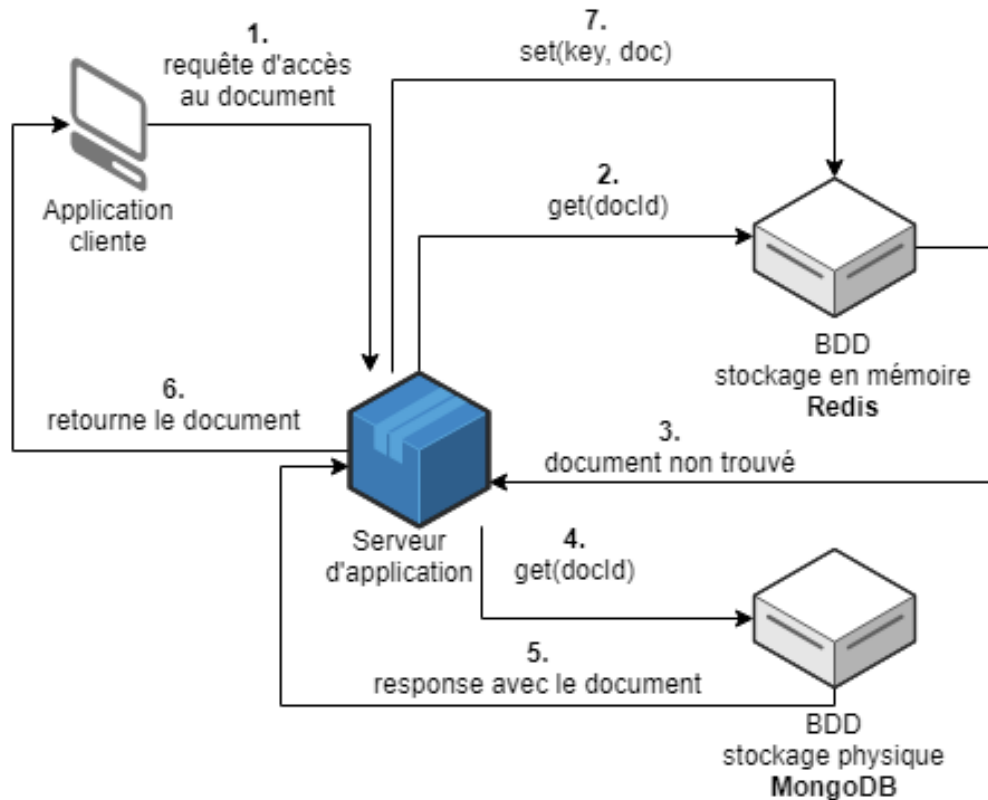


# Redis : exemple d'utilisation par Twitter

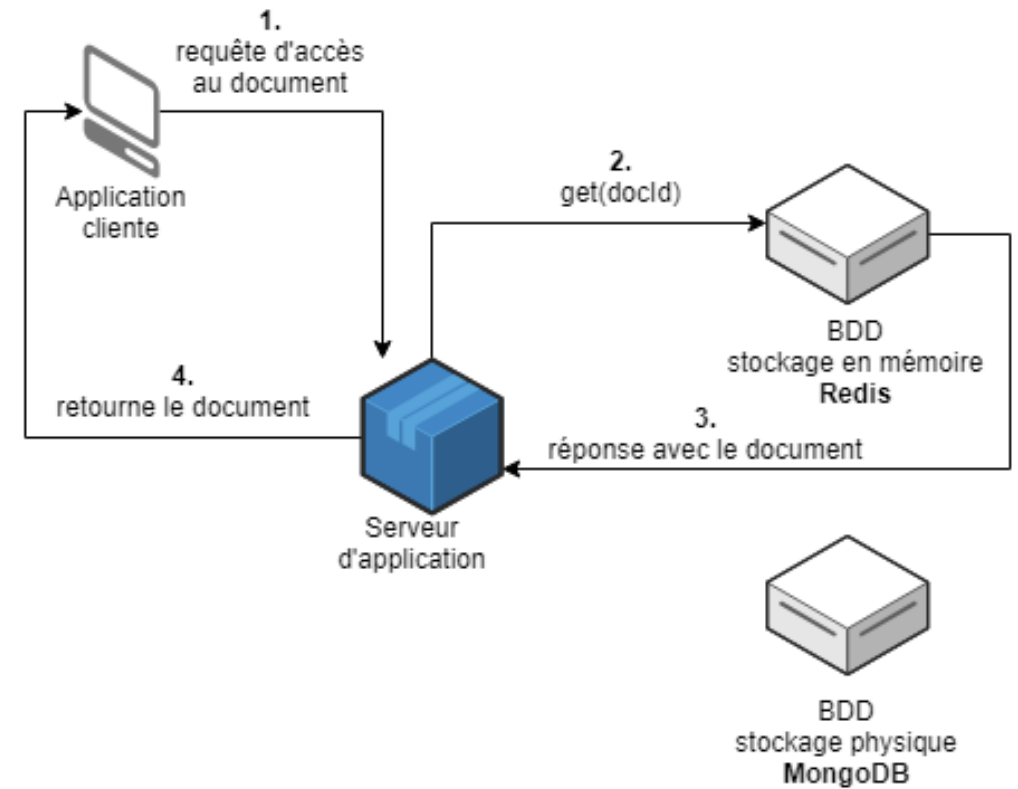
- Twitter : Réception d'environ 300 000 requêtes de lecture par secondes pour 6000 requêtes en écriture
- Pour pouvoir servir un volume de données aussi important : Récupération de données pré-compilées dans un cluster Redis chaque accès en lectures
- Stockage de tous les utilisateurs actifs ainsi que leurs *timelines* dans une machine Redis, configurée avec plusieurs terabytes de RAM

# Redis : exemple d'utilisation comme système de cache

A - Le document n'est pas encore dans le cache



B - Le document est disponible dans le cache



# Redis : exemple d'utilisation avec un SGBD relationnel

Utilisation de Redis comme index secondaire

Cust_ID	First_name	Last_name	City	Purchase
001	John	Smith	Boston	Laptop
002	Jane	Doe	Denver	Keyboard
003	David	Michaels	New York	Speaker
004	Susan	Andrews	San Jose	Laptop
005	Ben	Smith	Chicago	Mouse



Data table lookup

3. Return results

```
FT.SEARCH customer-idx @Purchase.Laptop
```

2. Secondary key query

```
FT.SEARCH customer-idx @Purchase.Laptop
"Cust_ID: 001"
1) "First_name"
2) "John"
3) "Last_name"
4) "Smith"
5) "City"
6) "Boston"
7) "Purchase"
8) "Laptop"
"Cust_ID: 004"
1) "First_name"
2) "Susan"
3) "Last_name"
4) "Andrews"
5) "City"
6) "San Jose"
7) "Purchase"
8) "Laptop"
```

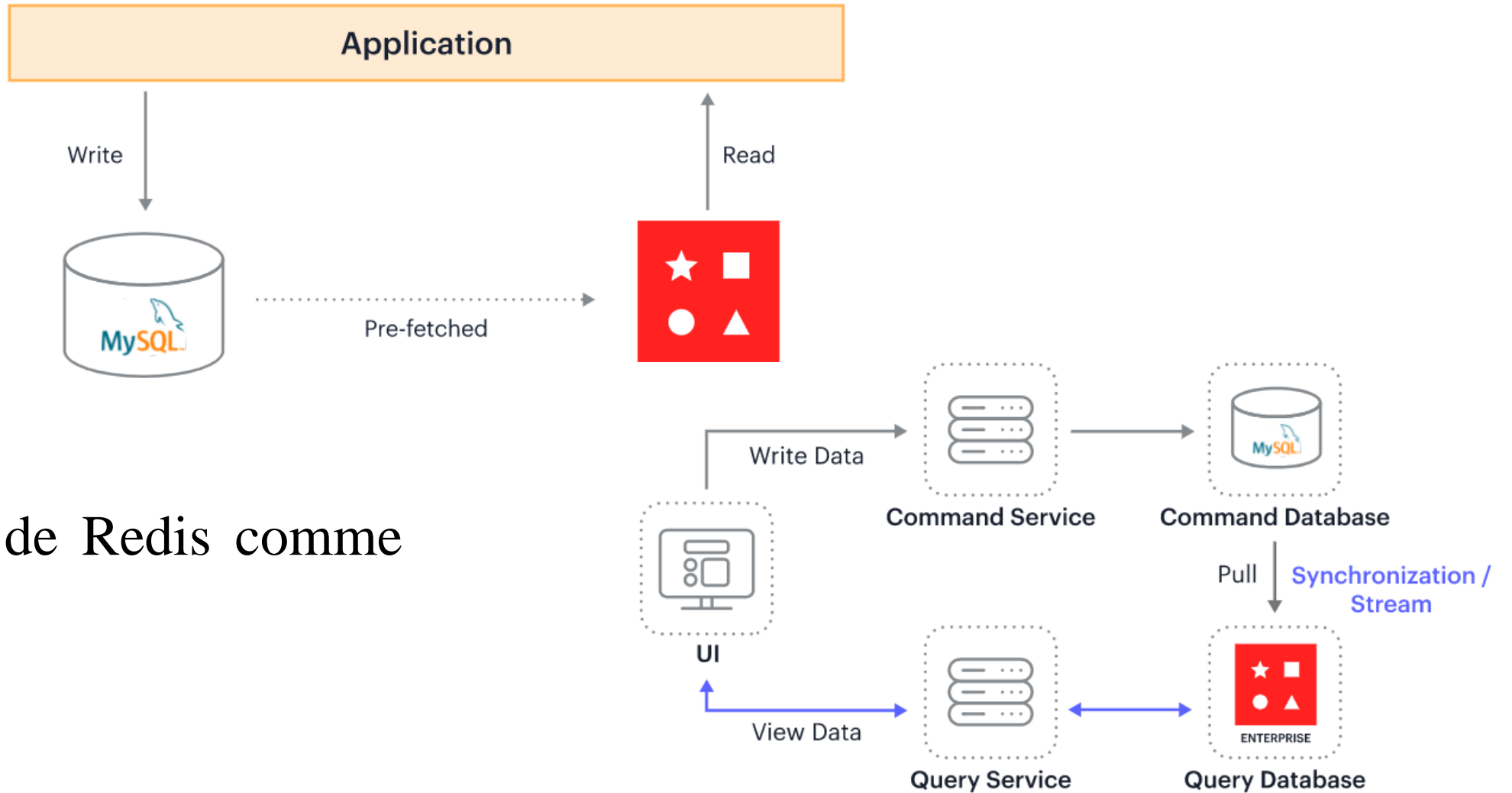
1. Created secondary index



```
>FT.CREATE customer-idx ON JSON PREFIX 1 Cust_ID: SCHEMA $.Purchase AS Purchase TEXT SORTABLE "OK"
```



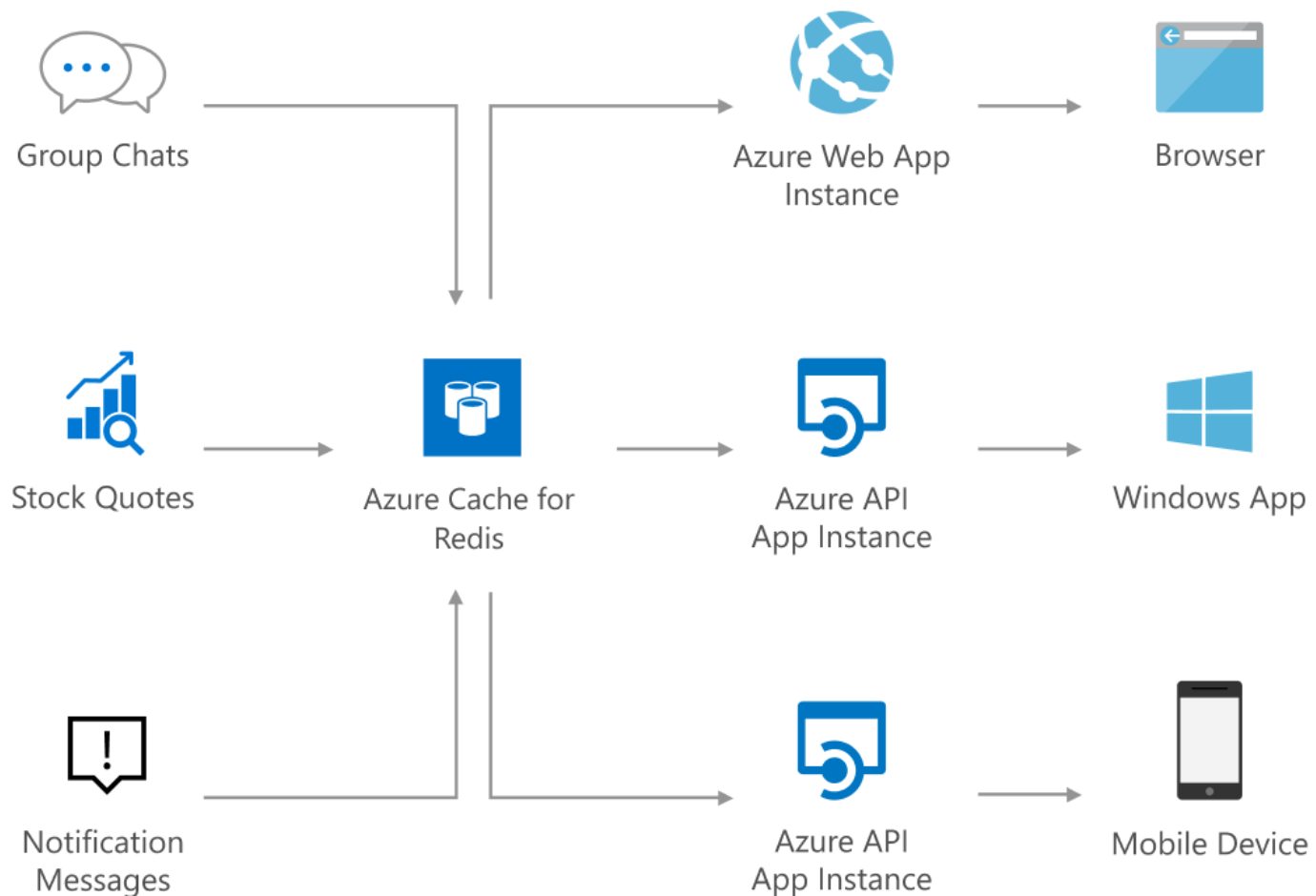
# Redis : exemple d'utilisation avec un SGBD relationnel










Utilisation de Redis comme cache

# Redis : exemple de file d'attente

Pour implémenter des architectures de publication/abonnement ou de file d'attente



# Redis : conclusion

-  **Accessibilité et performance**
-  **Types de données avancées**
-  **Sécuriser la persistance des données**
-  **Gestion d'écritures concurrentes**
  
-  **Sécurisation relative des échanges**
-  **Gestion des erreurs**
-  **pas une solution adaptée pour stocker les données de manière durable**

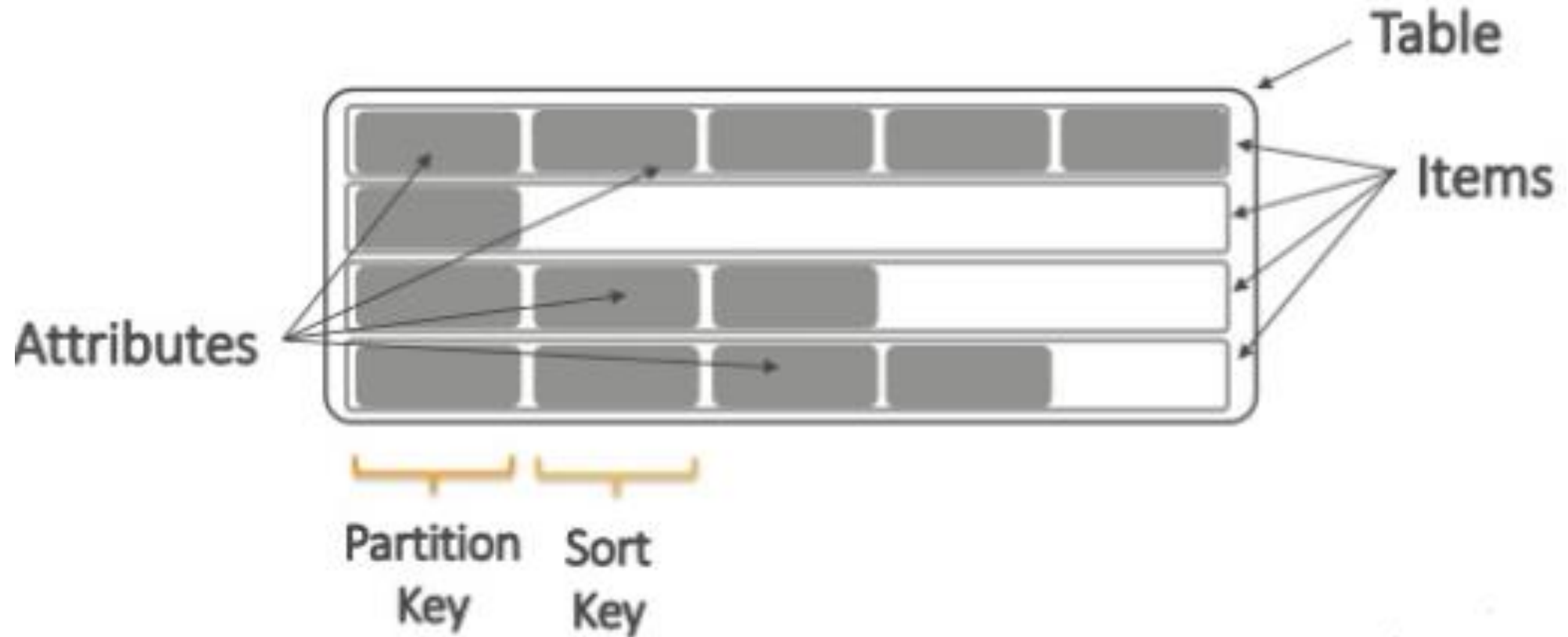
# DynamoDB

- Créée par Amazon en 2004
- Implémentation des modèles clés-valeurs et documents, en mode totalement distribué
- Article de recherche publié en 2007, « **Dynamo Paper** », qui a inspiré de nombreux moteurs NoSQL (ex. Riak, Cassandra ou Voldemort) : <https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>
- **Amazon DB** (2012) : base de données NoSQL sur Amazon Web Services
- Accès gratuit jusqu'à 100 Mo de stockage et une capacité en I/O de 5 écritures et 10 lectures par seconde – cf. <https://www.lemondeinformatique.fr/actualites/lire-dynamodb-une-base-de-donnees-nosql-sur-amazon-web-services-47471.html>
- Utilisée par Netflix, SnapShot, Nike etc.

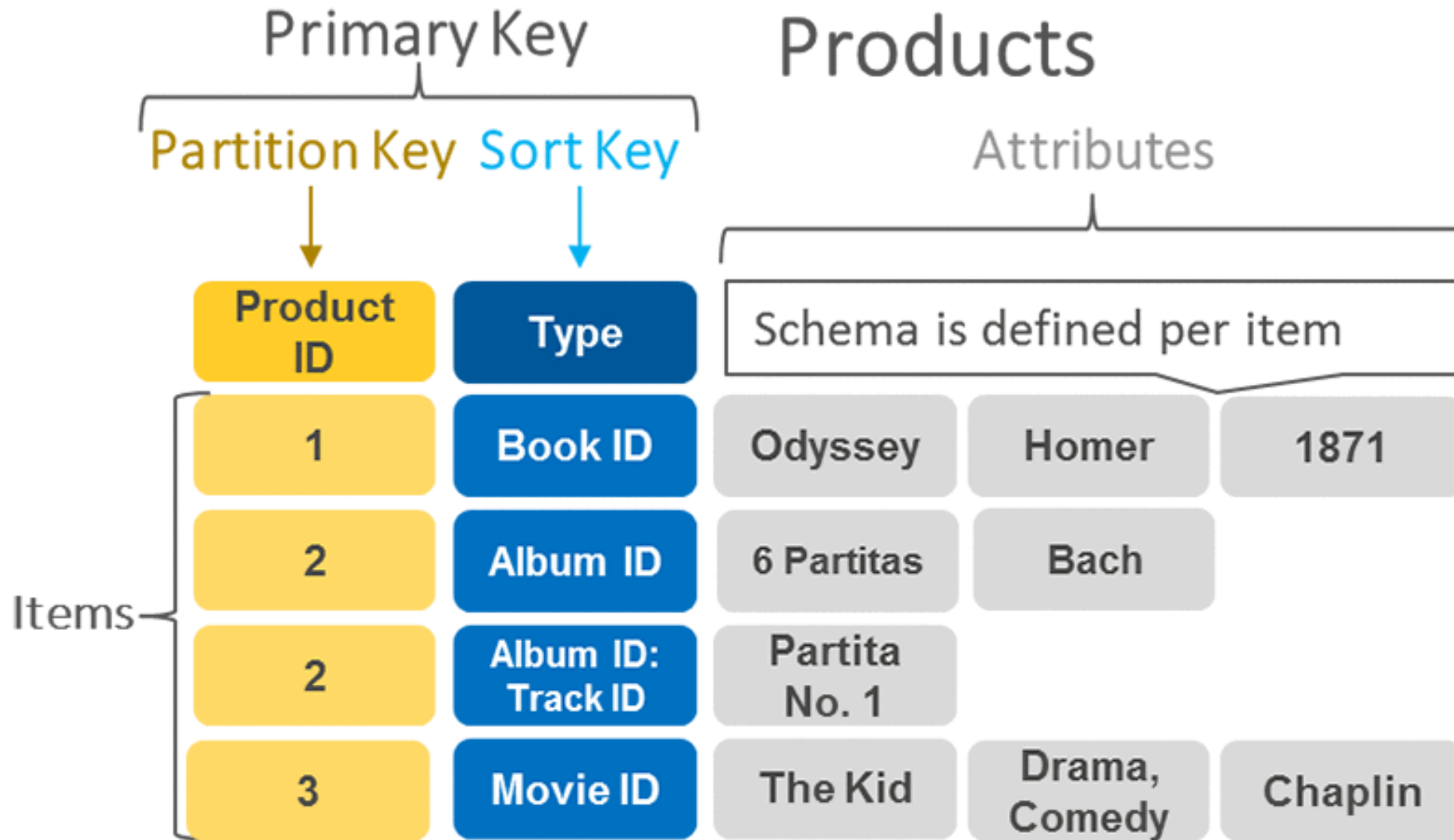
# DynamoDB : liens utiles

- Site officiel : <https://aws.amazon.com/fr/dynamodb/>
- Doc AmazonDB :  
[https://docs.aws.amazon.com/fr\\_fr/amazondynamodb/latest/developerguide/Introduction.html](https://docs.aws.amazon.com/fr_fr/amazondynamodb/latest/developerguide/Introduction.html)
- Article annonçant DynamoDB en 2012 :  
<https://www.allthingsdistributed.com/2012/06/amazon-dynamodb-growth.html>
- Transparents en ligne de Nicolas Travers, inspiré de Advait Deo :  
[http://chewbii.com/wp-content/uploads/2016/10/handout\\_Dynamo.pdf](http://chewbii.com/wp-content/uploads/2016/10/handout_Dynamo.pdf)

# DynamoDB : modèle de données



# DynamoDB : exemple



# DynamoDB : exemple de table

Primary Key		Attributes		
Actor (PARTITION)	Movie (SORT)			
Tom Hanks	Cast Away	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Chuck Noland	2000	Drama
	Toy Story	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Woody	1995	Children's
Tim Allen	Toy Story	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Nina Sayers	2010	Drama

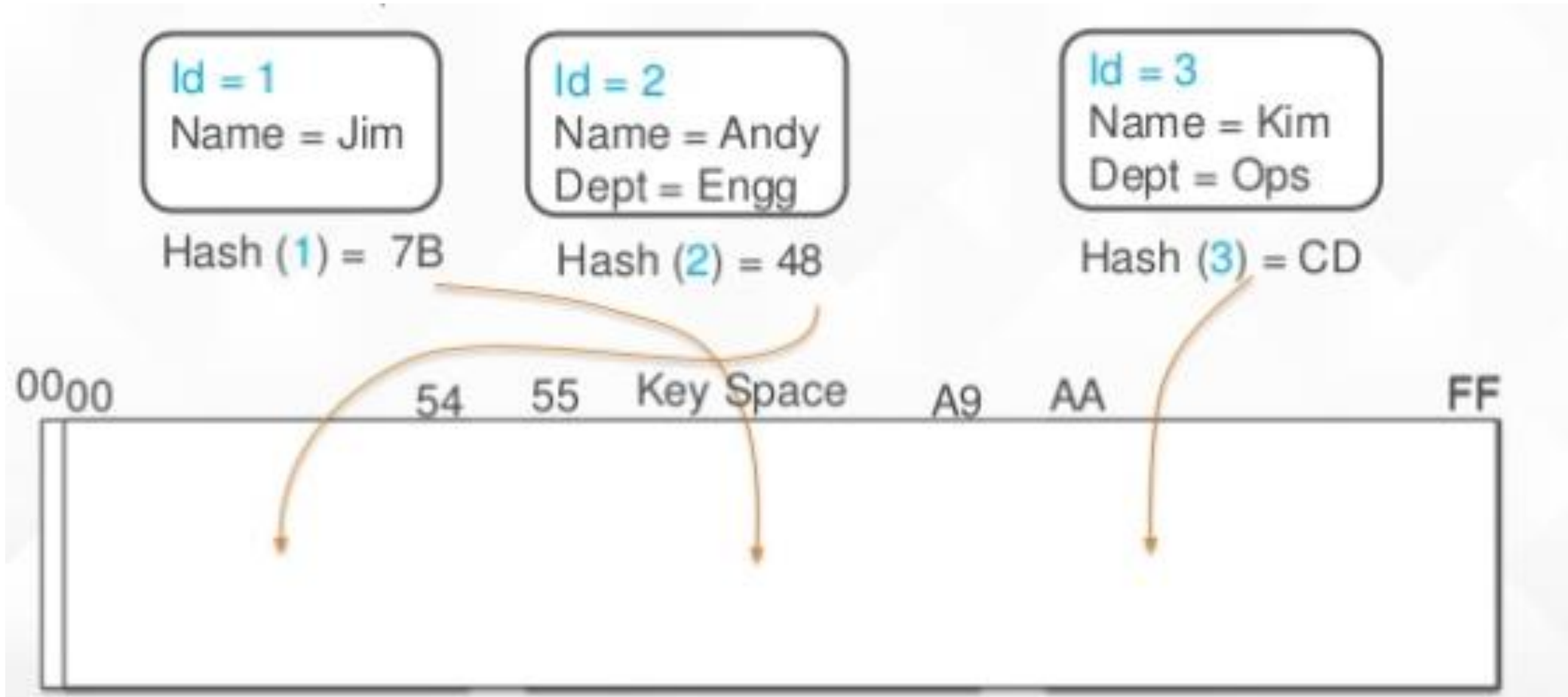


# DynamoDB : clé de partition / clé de tri

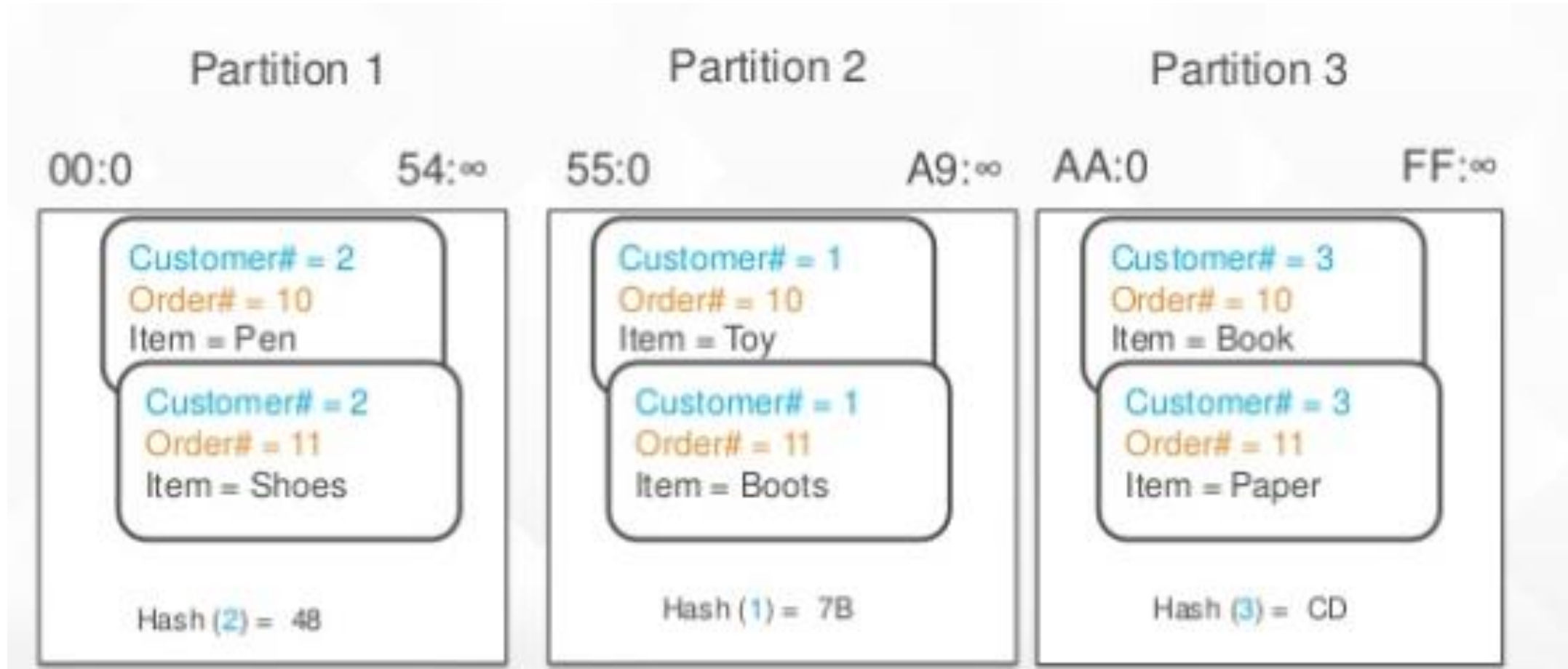
2 différents types de clés primaires :

- **Clé de partition** : clé primaire simple, composée d'un attribut
- **Clé de partition et clé de tri** : clé primaire composite, composée de deux attributs
- Valeur de la clé de partition utilisée comme entrée d'une fonction de hachage interne : *attribut de hachage*
- Chaque attribut de clé primaire doit être un scalaire et de type *string*, *number* ou *binary*

# DynamoDB : exemple de clé de partition



# DynamoDB : exemple de clé de partition et de clé de tri



# DynamoDB : principaux composants

- **Table** : ensemble d'éléments / données
- **Élément** : ensemble d'**attributs** (généralement scalaires)
- **Identifiant** unique pour chaque élément de la table
- Table **sans schéma**: pas de définition préalable des attributs ni des types de données
- Possibilité d'attributs imbriqués, jusqu'à 32 niveaux

```
{  
  "PersonID": 101,  
  "LastName": "Smith",  
  "FirstName": "Fred",  
  "Phone": "555-4321"  
}
```

```
{  
  "PersonID": 102,  
  "LastName": "Jones",  
  "FirstName": "Mary",  
  "Address": {  
    "Street": "123 Main",  
    "City": "Anytown",  
    "State": "OH",  
    "ZIPCode": 12345  
  }  
}
```

```
{  
  "PersonID": 103,  
  "LastName": "Stephens",  
  "FirstName": "Howard",  
  "Address": {  
    "Street": "123 Main",  
    "City": "London",  
    "PostalCode": "ER3 5K8"  
  },  
  "FavoriteColor": "Blue"  
}
```

# DynamoDB : identificateur à plusieurs attributs

- Possibilité d'avoir des identificateurs à plusieurs attributs

```
{  
  "Artist": "No One You Know",  
  "SongTitle": "My Dog Spot",  
  "AlbumTitle": "Hey Now",  
  "Price": 1.98,  
  "Genre": "Country",  
  "CriticRating": 8.4  
}
```

```
{  
  "Artist": "No One You Know",  
  "SongTitle": "Somewhere Down The Road",  
  "AlbumTitle": "Somewhat Famous",  
  "Genre": "Country",  
  "CriticRating": 8.4,  
  "Year": 1984  
}
```

```
{  
  "Artist": "The Acme Band",  
  "SongTitle": "Still in Love",  
  "AlbumTitle": "The Buck Starts Here",  
  "Price": 2.47,  
  "Genre": "Rock",  
  "PromotionInfo": {  
    "RadioStationsPlaying": [  
      "KHCR",  
      "KQBX",  
      "WTNR",  
      "WJHJ"  
    ],  
    "TourDates": {  
      "Seattle": "20150625",  
      "Cleveland": "20150630"  
    },  
    "Rotation": "Heavy"  
  }  
}
```

# DynamoDB : dénormalisation par duplication

Authors		
AuthorId	AuthorName	AuthorBirthdate
1	John Grisham	February 8, 1955
2	Stephen King	September 21, 1947
3	J.K. Rowling	July 31, 1965

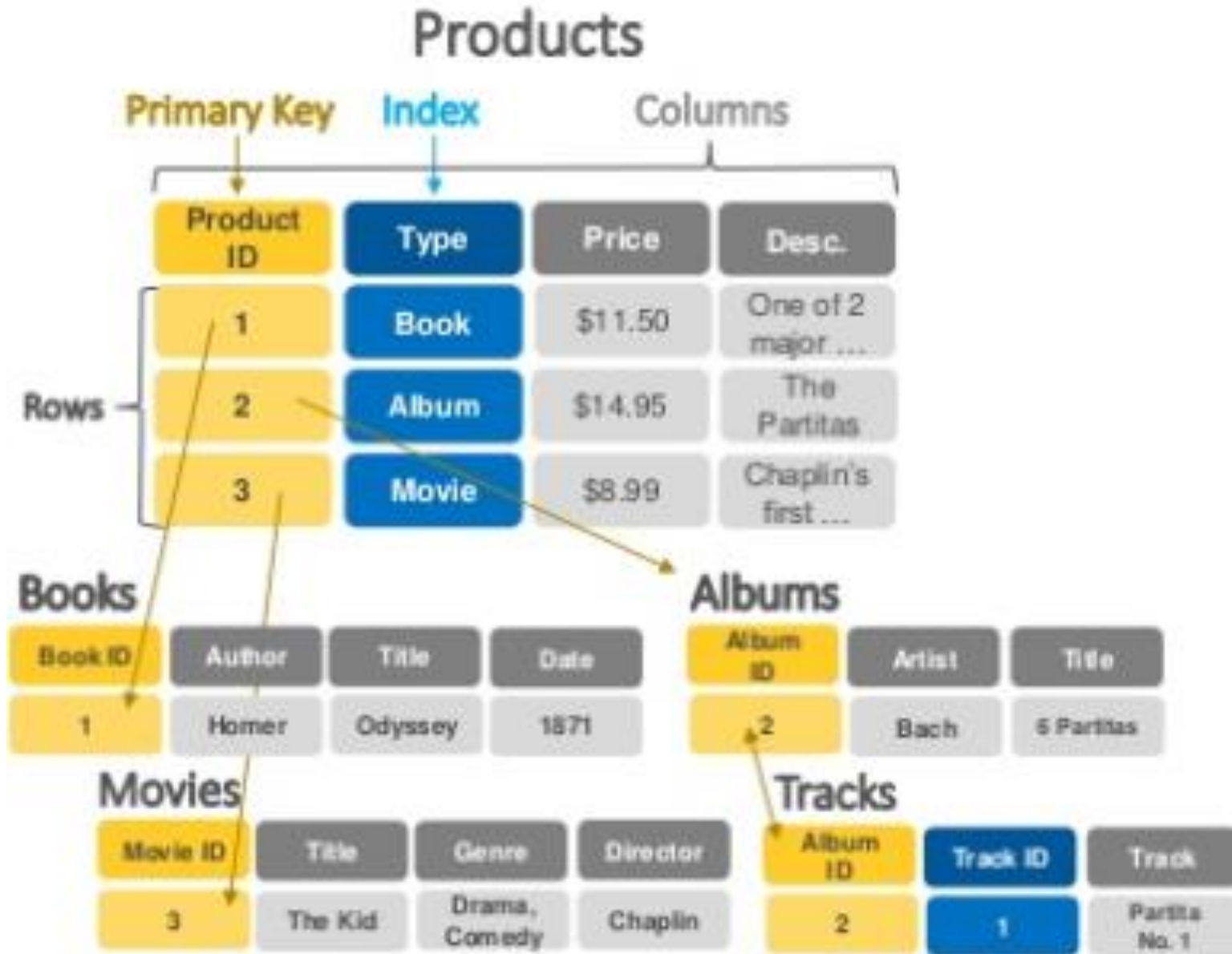
Books			
BookId	AuthorId	BookTitle	ReleaseYear
1	2	The Shining	1977
2	2	It	1986
3	3	Harry Potter and the Sorcerer's Stone	1997

**Modèle relationnel**  
modélisant des livres et des auteurs (chaque livre ayant un seul auteur)

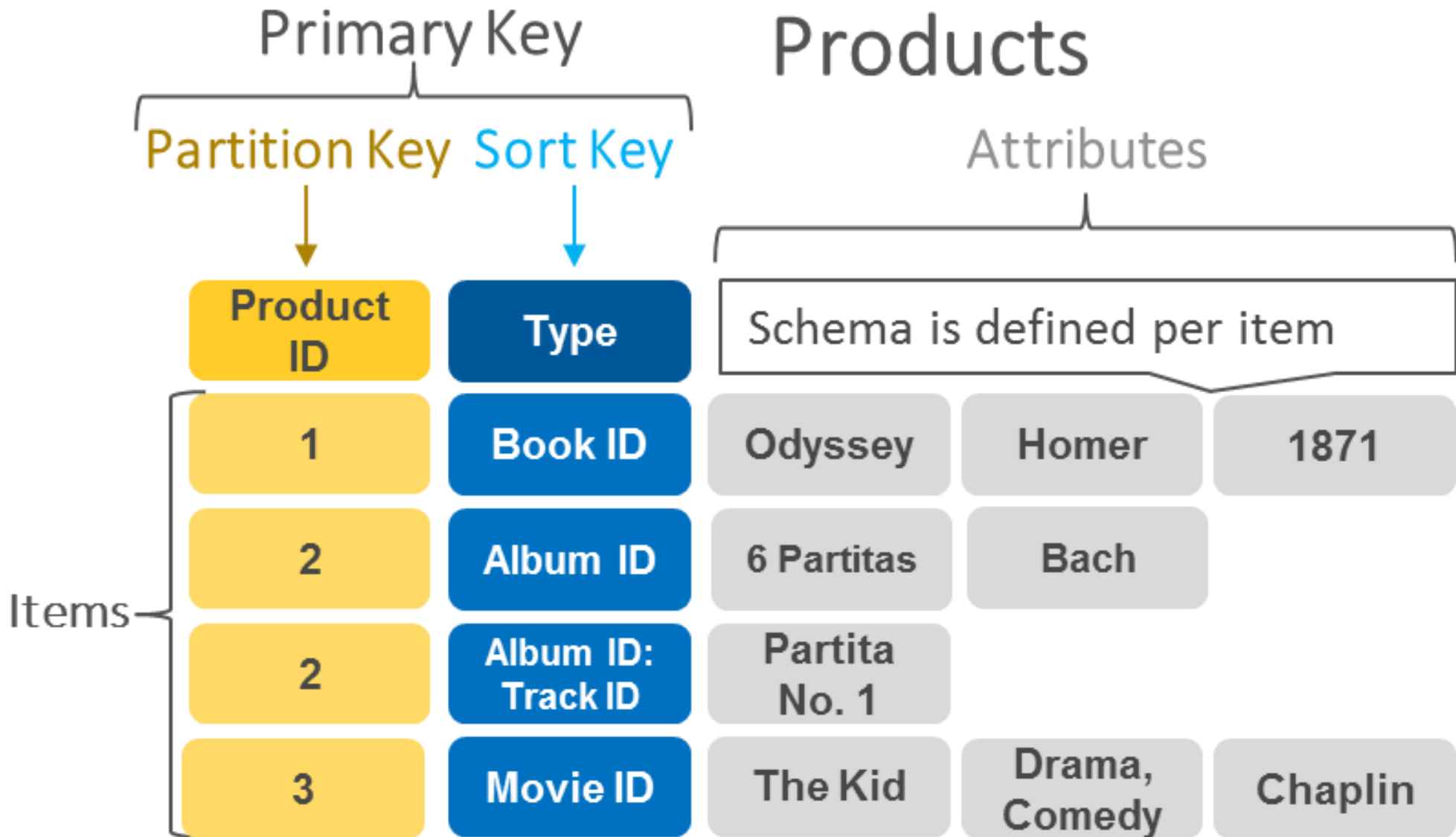
Primary key		Attributes	
Partition key: AuthorName	Sort key: BookName		
Stephen King	It	AuthorBirthdate	ReleaseYear
		September 21, 1947	1986
Stephen King	The Shining	AuthorBirthdate	ReleaseYear
		September 21, 1947	1977
J.K. Rowling	Harry Potter and the Sorcerer's Stone	AuthorBirthdate	ReleaseYear
		July 31, 1965	1997

**Collection d'*items* sous DynamoDB** : avec duplication de données

# Exemple d'un modèle relationnel



# Exemple du même modèle sous DynamoDB

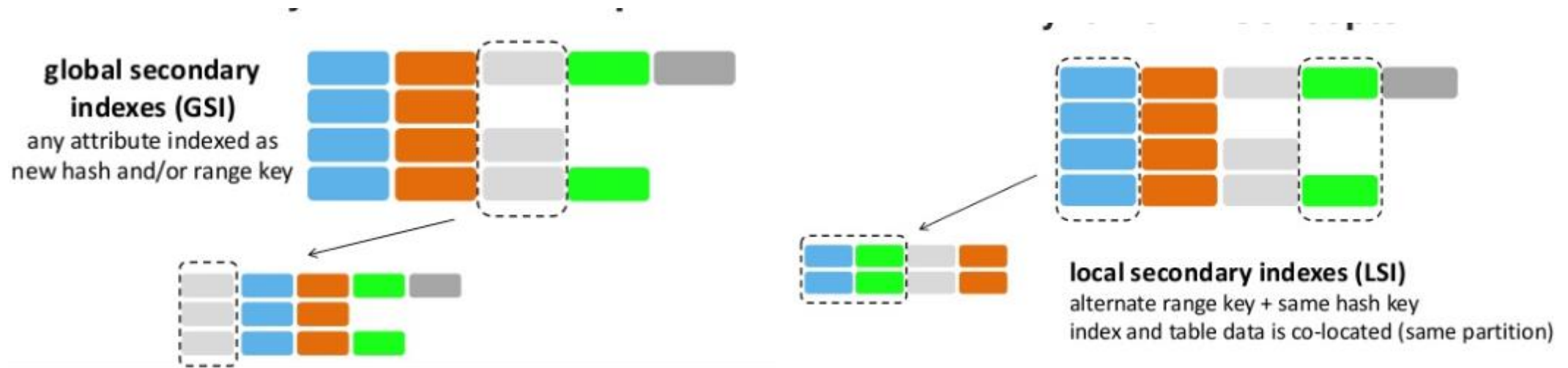




# DynamoDB : indexation

Possibilité de créer des index secondaires :

- **Global secondary index** : index avec une clé de partition et une clé de tri qui peuvent être différentes de celles de la table
- **Index secondaire local** : index avec la même clé de partition que la table, mais une clé de tri différente



# DynamoDB : exemple d'index global

Primary Key		Attributes		
Actor (PARTITION)	Movie (SORT)			
Tom Hanks	Cast Away	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Chuck Noland	2000	Drama
	Toy Story	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Woody	1995	Children's
Tim Allen	Toy Story	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Nina Sayers	2010	Drama

GSI (MoviesActorsIndex)		Attributes		
Movie (PARTITION)	Actor (SORT)			
Cast Away	Tom Hanks	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Chuck Noland	2000	Drama
Toy Story	Tom Hanks	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Woody	1995	Children's
Toy Story	Tim Allen	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Buzz Lightyear	1995	Children's
Black Swan	Natalie Portman	<b>Role</b>	<b>Year</b>	<b>Genre</b>
		Nina Sayers	2010	Drama

# DynamoDB : exemple d'index local

Primary Key		Attributes		
Actor (PARTITION)	Movie (SORT)	Role	Year	Genre
Tom Hanks	Cast Away	Chuck Noland	2000	Drama
	Toy Story	Woody	1995	Children's
Tim Allen	Toy Story	Buzz Lightyear	1995	Children's
	Black Swan	Nina Sayers	2010	Drama

LSI (ActorYearIndex)		Attributes		
Actor (PARTITION)	Year (SORT)	Role	Movie	Genre
Tom Hanks	2000	Chuck Noland	Cast Away	Drama
	1995	Woody	Toy Story	Children's
Tim Allen	1995	Buzz Lightyear	Toy Story	Children's
	2010	Nina Sayers	Black Swan	Drama

# DynamoDB : autre exemple d'index global

Music

GenreAlbumTitle

```
{  
  "Artist": "No One You Know",  
  "SongTitle": "My Dog Spot",  
  "AlbumTitle": "Hey Now",  
  "Price": 1.98,  
  "Genre": "Country",  
  "CriticRating": 8.4  
}
```

```
{  
  "Artist": "No One You Know",  
  "SongTitle": "Somewhere Down The Road",  
  "AlbumTitle": "Somewhat Famous",  
  "Genre": "Country",  
  "CriticRating": 8.4,  
  "Year": 1984  
}
```

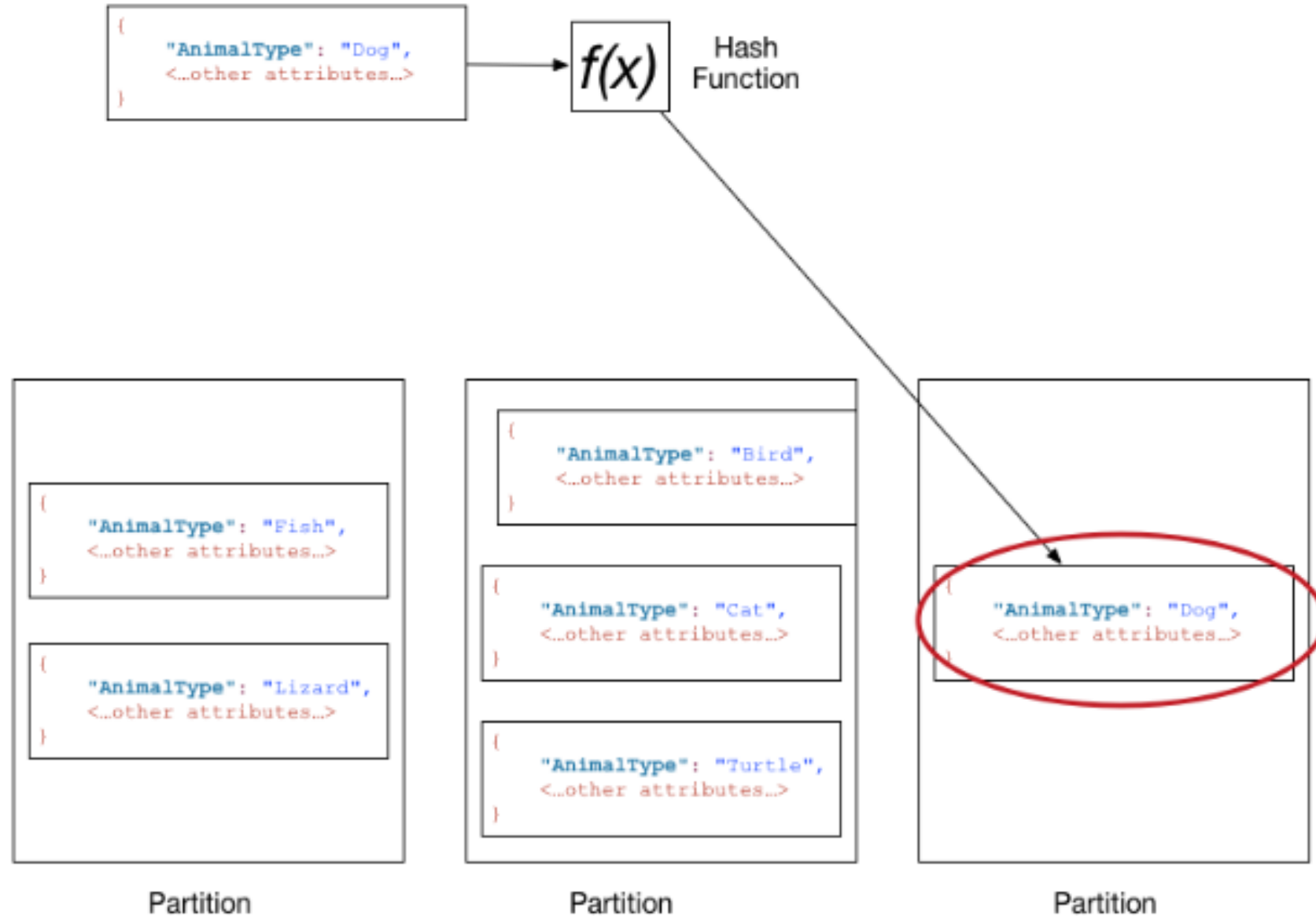
```
{  
  "Genre": "Country",  
  "AlbumTitle": "Hey Now",  
  "Artist": "No One You Know",  
  "SongTitle": "My Dog Spot"  
}
```

```
{  
  "Genre": "Country",  
  "AlbumTitle": "Somewhat Famous",  
  "Artist": "No One You Know",  
  "SongTitle": "Somewhere Down The Road"  
}
```

# DynamoDB : partitionnement

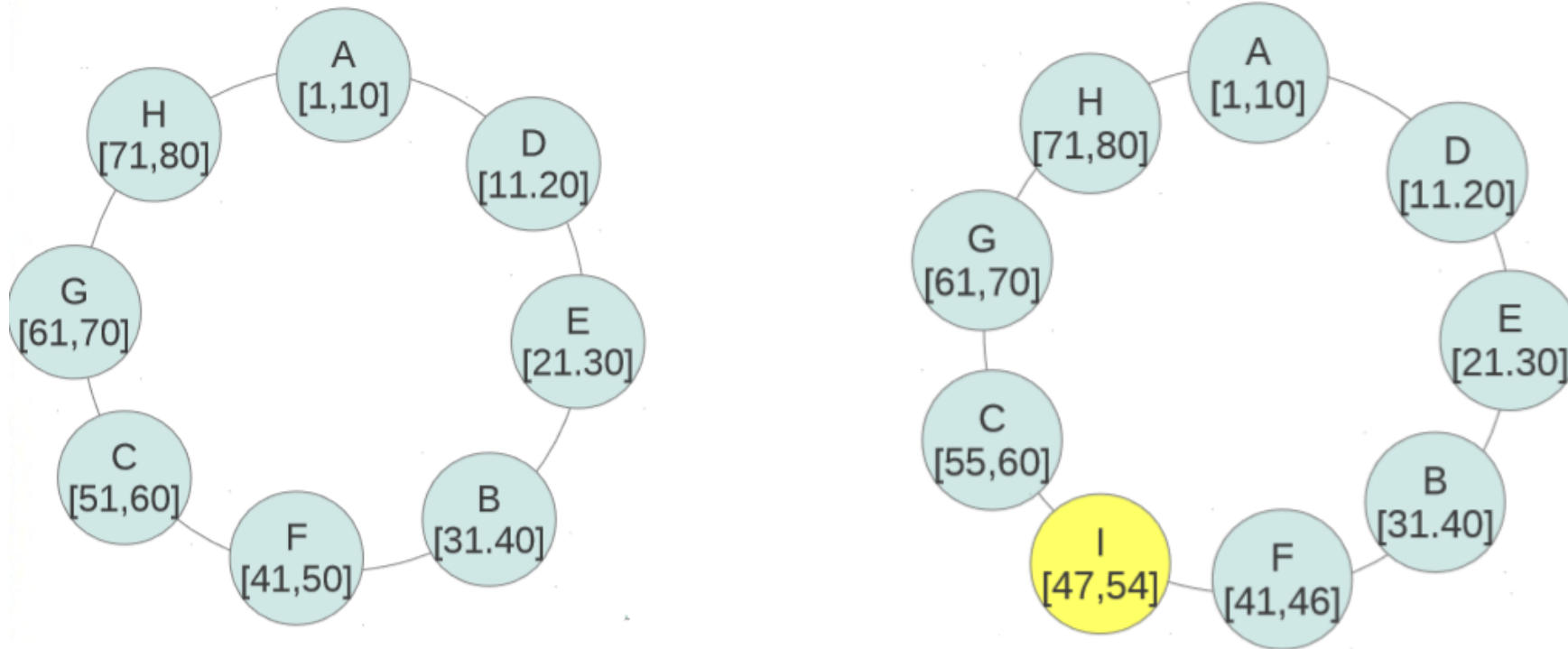
- Stocke des données dans des partitions
- Partition : allocation de stockage pour une table, automatiquement répliquée sur plusieurs zones de disponibilité
- Partition déterminée à partir valeur de sortie de la fonction de hachage appliquée à la valeur de la clé de partition
- Stockage de tous les éléments avec la même valeur de clé de partition physiquement proches les uns des autres, triés par la valeur de clé de tri

# DynamoDB : exemple de partitionnement



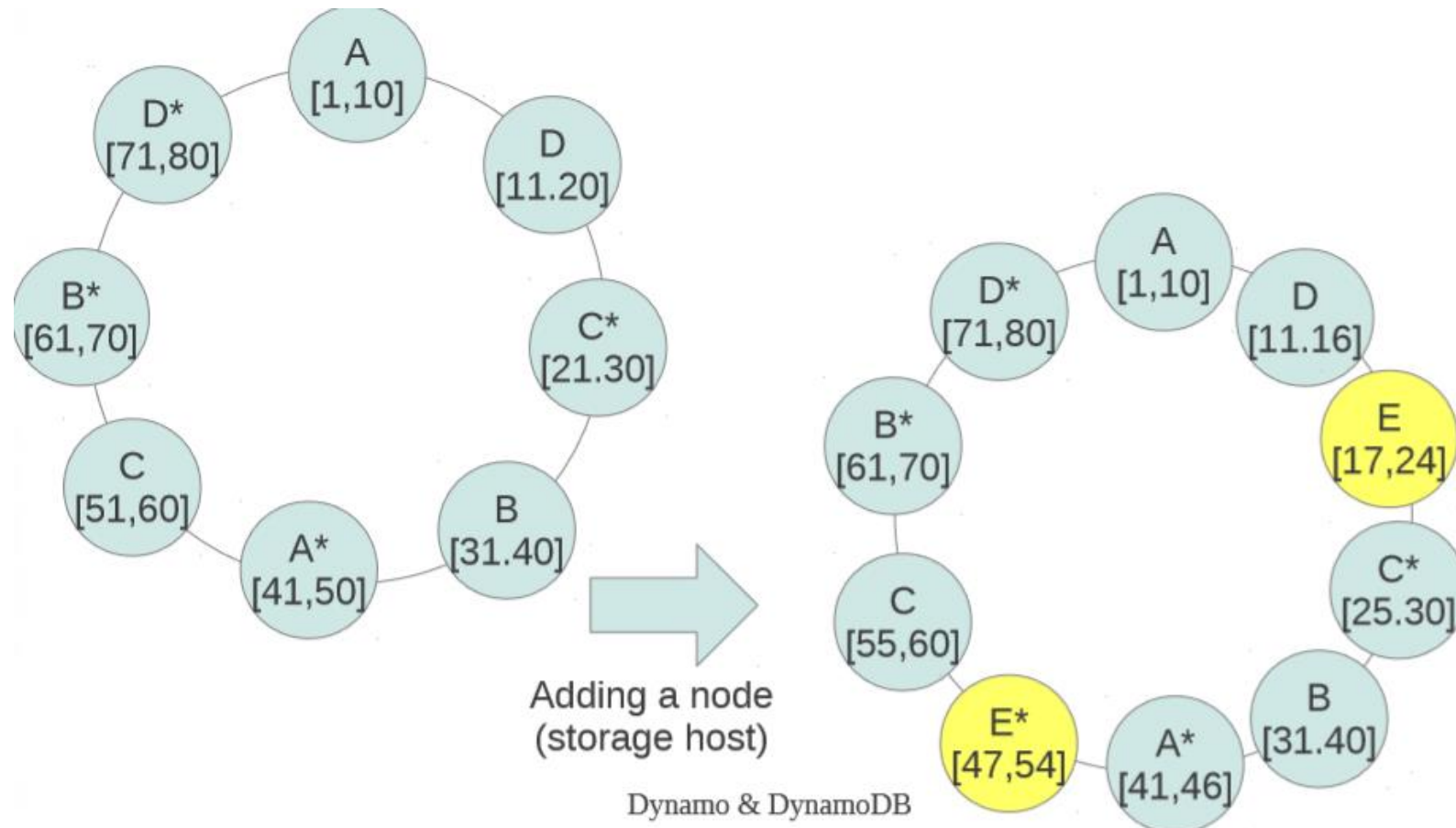
# DynamoDB : hachage cohérent

- Données partitionnées sur différents nœuds répartis sur un anneau (*ring*)



# DynamoDB : hachage cohérent avec nœuds virtuels

Utilisation du hachage cohérent avec des nœuds virtuels pour homogénéiser la répartition des données

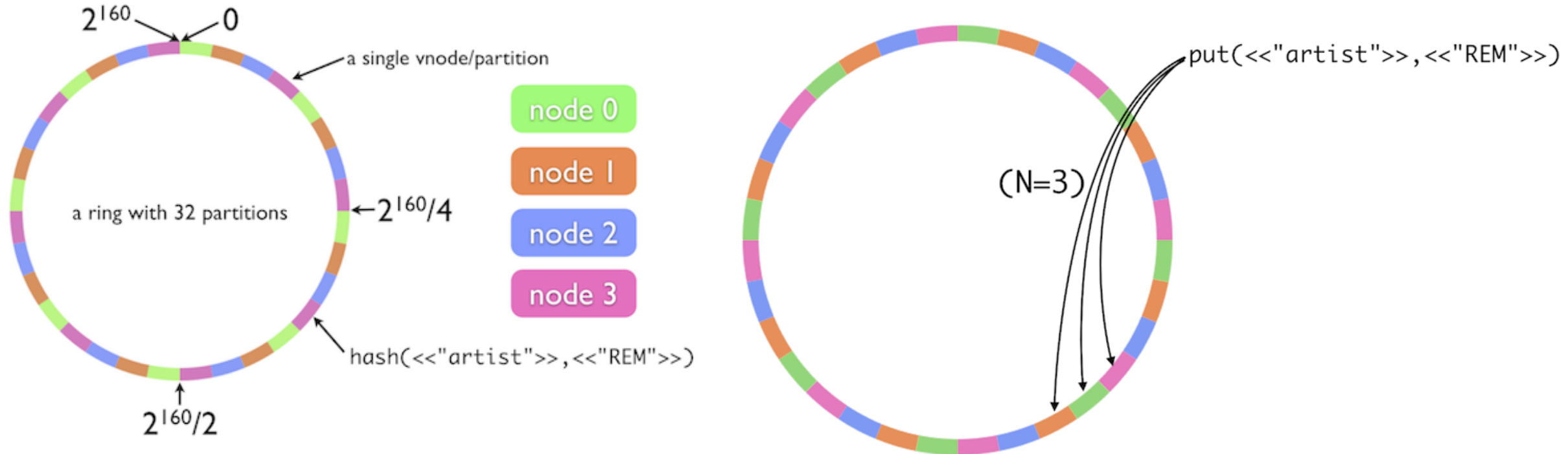




# DynamoDB : réplication

- Réplication de chaque item sur  $N$  (*replication factor*) nœud physique (pas virtuel)
- Association de chaque clé à un nœud coordinateur
  - Responsable des opérations de lecture et d'écriture pour la clé
  - Responsable du stockage en local des items associés à la clé et de leur réplication sur  $N-1$  nœuds voisins (enregistrés dans une liste de préférence)
- Facteur de réplication par défaut : 3

# DynamoDB : exemple de répartition



# Modèle de données DynamoDB : différence avec SQL

```
CREATE TABLE Music (  
  Artist VARCHAR(20) NOT NULL,  
  SongTitle VARCHAR(30) NOT NULL,  
  AlbumTitle VARCHAR(25),  
  Year INT,  
  Price FLOAT,  
  Genre VARCHAR(10),  
  Tags TEXT,  
  PRIMARY KEY(Artist, SongTitle)  
);
```

```
{  
  TableName : "Music",  
  KeySchema: [  
    {  
      AttributeName: "Artist",  
      KeyType: "HASH", //Partition key  
    },  
    {  
      AttributeName: "SongTitle",  
      KeyType: "RANGE" //Sort key  
    }  
  ],  
  AttributeDefinitions: [  
    {  
      AttributeName: "Artist",  
      AttributeType: "S"  
    },  
    {  
      AttributeName: "SongTitle",  
      AttributeType: "S"  
    }  
  ],  
  ProvisionedThroughput: { // Only specified if using provisioned mode  
    ReadCapacityUnits: 1,  
    WriteCapacityUnits: 1  
  }  
}
```

# DynamoDB : création d'une table via l'interface

## Create DynamoDB table

Tutorial



DynamoDB is a schema-less database that only requires a table name and primary key. The table's primary key is made up of one or two attributes that uniquely identify items, partition the data, and sort data within each partition.

**Table name\***  ⓘ

**Primary key\*** Partition key

String ⓘ

Add sort key

String ⓘ

### Table settings

Default settings provide the fastest way to get started with your table. You can modify these default settings now or after your table has been created.

Use default settings

- No secondary indexes.
- Provisioned capacity set to 5 reads and 5 writes.

# Requêtes d'interrogation DynamoDB : différence avec SQL (1/4)

```
/* Return all of the data in the table */  
SELECT * FROM Music;
```

```
// Return all of the data in the table  
{  
  TableName: "Music"  
}
```

# Requêtes d'interrogation DynamoDB : différence avec SQL (2/4)

```
/* Return all of the values for Artist and Title */  
SELECT Artist, Title FROM Music;
```

```
// Return all of the values for Artist and Title  
{  
  TableName: "Music",  
  ProjectionExpression: "Artist, Title"  
}
```

# Requêtes d'interrogation DynamoDB : différence avec SQL (3/4)

```
/* Return a single song, by primary key */  
  
SELECT * FROM Music  
WHERE Artist='No One You Know' AND SongTitle = 'Call Me Today';
```

```
// Return a single song, by primary key  
  
{  
  TableName: "Music",  
  KeyConditionExpression: "Artist = :a and SongTitle = :t",  
  ExpressionAttributeValues: {  
    ":a": "No One You Know",  
    ":t": "Call Me Today"  
  }  
}
```

# Requêtes d'interrogation DynamoDB : différence avec SQL (4/4)

```
/* Return all of the songs by an artist, with a particular word in the title...  
...but only if the price is less than 1.00 */
```

```
SELECT * FROM Music  
WHERE Artist='No One You Know' AND SongTitle LIKE '%Today%'  
AND Price < 1.00;
```

```
// Return all of the songs by an artist, with a particular word in the title...  
// ...but only if the price is less than 1.00
```

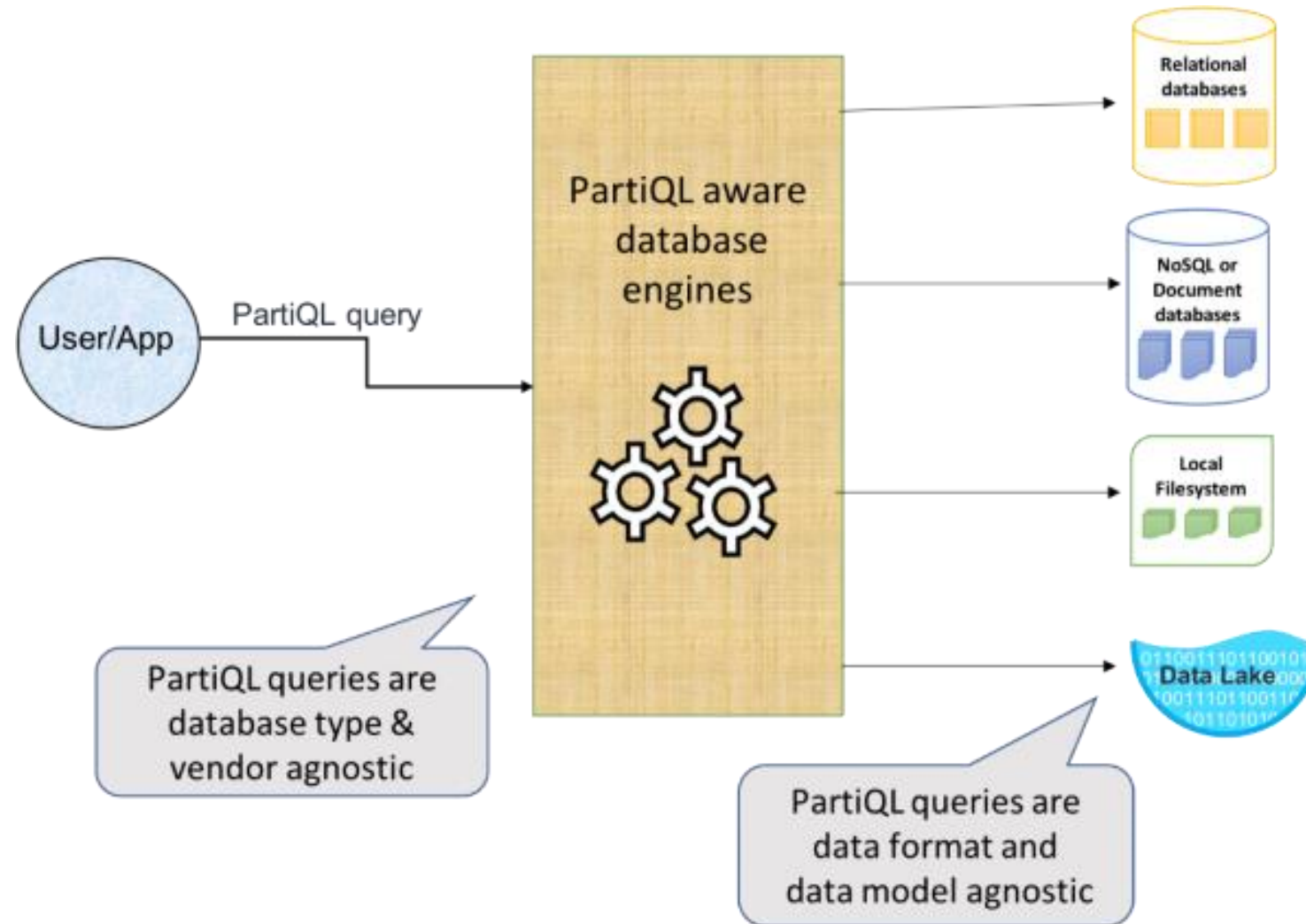
```
{  
  TableName: "Music",  
  KeyConditionExpression: "Artist = :a and contains(SongTitle, :t)",  
  FilterExpression: "price < :p",  
  ExpressionAttributeValues: {  
    ":a": "No One You Know",  
    ":t": "Today",  
    ":p": 1.00  
  }  
}
```



# DynamoDB + PartiQL

- Annonce en 2019 d'un un nouveau langage de requête compatible avec SQL : **PartiQL** – compatible avec DynamoDB depuis nov. 2020  
<https://aws.amazon.com/fr/about-aws/whats-new/2020/11/you-now-can-use-a-sql-compatible-query-language-to-query-insert-update-and-delete-table-data-in-amazon-dynamodb/>
- **PartiQL** : Langage entièrement open source sous licence Apache2.0, présenté comme « hautement flexible », censé faciliter la recherche et l'extraction efficace de grandes quantités et variétés de données, quel que soit le lieu ou le format dans lequel elles sont stockées
- <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-reference.html>
- <https://partiql.org/>

# PartiQL



PartiQL queries are database type & vendor agnostic

PartiQL queries are data format and data model agnostic

# DynamoDB + PartiQL : exemple

## query-documentAPI.js

```
const documentClient = new AWS.DynamoDB.DocumentClient();

async function queryWithDocumentClient({year, title}) {
  const params = {
    TableName: "Movies",
    ProjectionExpression: "#yr, title, info",
    KeyConditionExpression: "#yr = :yyyy and title = :title",
    ExpressionAttributeNames: {
      "#yr": "year"
    },
    ExpressionAttributeValues: {
      ":yyyy": year,
      ":title": title
    }
  };








  const results = await documentClient.query(params).promise();
}
```

## query-partiQL.js

```
const dynamoDB = new AWS.DynamoDB();

async function queryWithPartiQL({year, title}) {
  const statement = `SELECT * FROM Movies WHERE "year" = ${year} and "title" = '${title}'`;
  const results = await dynamoDB.executeStatement({Statement: statement}).promise();
}
```

# DynamoDB : conclusion

-  **Gestion clé-valeur et document**
-  **Scalabilité**
-  **Certaine « gestion de transactions ACID »**
-  **Pas d'infrastructure à gérer**
  
-  **Taille limitée pour les items et les requêtes**
-  **Plus on veut de fonctionnalité plus on paye**
-  **Pas de jointure entre tables**

# Clé-valeur : dans un moteur SQL

- HSTORE sous PostgreSQL

(cf. doc <https://www.postgresql.org/docs/current/hstore.html> )

```
1 CREATE EXTENSION hstore;
2 CREATE TABLE mytable (h hstore);
3
4 INSERT INTO mytable VALUES ('key1=>value1, key2=>value2');
5
6 SELECT * FROM mytable;
```

	h
	hstore
1	"key1"=>"value1", "key2"=>"value2"

```
8 SELECT h->'key1' AS valeur FROM mytable;
9
```

Query Query History

```
1 SELECT h['key1'] FROM mytable;
```

Data Output Explain Messages History

	valeur
	text
1	value1

	h
	text
1	value1

# Conversion d'une relation en ensemble de clé-valeur

```
CREATE EXTENSION HSTORE;
```

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
```

```
INSERT INTO test VALUES (123, 'foo', 'bar');
```

```
SELECT * FROM test;
```

col1	col2	col3
123	foo	bar

Transformation de la relation en couple (clé, valeur) :

```
SELECT hstore(t) FROM test AS t;
```

hstore

"col1"=>"123", "col2"=>"foo", "col3"=>"bar"

# Exemple d'attributs de type clé-valeur sous PostgreSQL

```
1 CREATE TABLE books (  
2     id serial primary key,  
3     title VARCHAR (255),  
4     attr hstore  
5 );  
6  
7 INSERT INTO books (title, attr) VALUES          INSERT INTO books (title, attr) VALUES  
8     ( 'PostgreSQL Tutorial',                    (  
9         "paperback" => "243",  
10        "publisher" => "postgresqltutorial.com",  
11        "language"  => "English",  
12        "ISBN-13"   => "978-1449370000",  
13        "weight"    => "11.2 ounces"  
14    );  
15  
16        'PostgreSQL Cheat Sheet',  
17        "paperback" => "5",  
18        "publisher" => "postgresqltutorial.com",  
19        "language"  => "English",  
20        "ISBN-13"   => "978-1449370001",  
21        "weight"    => "1 ounces"  
22    );
```

# Requête sur un attribut de type clé-valeur sous PostgreSQL

```
1 SELECT attr FROM books;
```

	attr
	hstore
1	"weight"=>"11.2 ounces", "ISBN-13"=>"978-1449370000", "language"=>"English", "paperback"=>"243", "publisher"=>"postgresqtutorial.com"
2	"weight"=>"1 ounces", "ISBN-13"=>"978-1449370001", "language"=>"English", "paperback"=>"5", "publisher"=>"postgresqtutorial.com"

```
1 SELECT attr -> 'ISBN-13' AS isbn
2 FROM books;
```

	isbn
	text
1	978-1449370000
2	978-1449370001



# Requête sur la valeur d'une clé particulière sous PostgreSQL

```
1 SELECT title, attr -> 'weight' AS weight
2 FROM books
3 WHERE attr -> 'ISBN-13' = '978-1449370000';
```

	title	weight
	character varying (255)	text
1	PostgreSQL Tutorial	11.2 ounces

```
1 SELECT title, attr -> 'publisher' AS weight
2 FROM books
3 WHERE attr -> 'publisher' LIKE 'postgres%';
```

	title	weight
	character varying (255)	text
1	PostgreSQL Tutorial	postgresqltutorial.com
2	PostgreSQL Cheat Sheet	postgresqltutorial.com

# Ajout d'une paire clé-valeur sous PostgreSQL

```
1 UPDATE books
2   SET attr = attr || '"freeshipping"=>"yes"' :: hstore;
3
4 SELECT title, attr -> 'freeshipping' AS freeshipping
5 FROM books;
```

	title	freeshipping
	character varying (255)	text
1	PostgreSQL Tutorial	yes
2	PostgreSQL Cheat Sheet	yes

```
1 UPDATE books
2   SET attr = delete(attr, 'freeshipping');
3
4 SELECT title, attr -> 'freeshipping' AS freeshipping
5 FROM books;
```

	title	freeshipping
	character varying (255)	text
1	PostgreSQL Tutorial	[null]
2	PostgreSQL Cheat Sheet	[null]

# Test d'existence d'une clé PostgreSQL

```
1 SELECT
2   title,
3   attr->'publisher' as publisher,
4   attr
5 FROM books
6 WHERE attr ? 'publisher';
```

	title	publisher	attr
	character varying (255)	text	hstore
1	PostgreSQL Tutorial	postgresqltutorial.com	"weight"=>"11.2 ounces", "ISBN-13"=>"978-14493...
2	PostgreSQL Cheat Sheet	postgresqltutorial.com	"weight"=>"1 ounces", "ISBN-13"=>"978-14493700...

```
1 SELECT title, attr->'publisher' as publisher, attr
2 FROM books
3 WHERE attr ? 'author';
```

	title	publisher	attr
	character varying (255)	text	hstore

# Test sur des paires clé-valeur PostgreSQL

```
1 SELECT title
2 FROM books
3 WHERE attr @> '"weight"=>"11.2 ounces"' :: hstore;
```

	title
	character varying (255)
1	PostgreSQL Tutorial

```
1 SELECT title
2 FROM books
3 WHERE attr ?& ARRAY [ 'language', 'weight' ];
```

	title
	character varying (255)
1	PostgreSQL Tutorial
2	PostgreSQL Cheat Sheet

```
1 SELECT akeys (attr)
2 FROM books;
```

	akeys
	text[]
1	weight,ISBN-13,language,paperback,publisher
2	weight,ISBN-13,language,paperback,publisher

# Conversion des paires clé-valeur PostgreSQL

```
1 SELECT title, hstore_to_json (attr) json
2 FROM books;
```

	title character varying (255)	json json
1	PostgreSQL Tutorial	{"language":"English","paperback":"243","ISBN-13":"978-1449370000","weight":"11.2 ounces","publisher":"postgresqtutorial.com..."}
2	PostgreSQL Cheat Sheet	{"language":"English","paperback":"5","ISBN-13":"978-1449370001","weight":"1 ounces","publisher":"postgresqtutorial.com"}

```
1 SELECT title, (EACH(attr) ).*
2 FROM books;
```

	title character varying (255)	key text	value text
1	PostgreSQL Tutorial	weight	11.2 ounces
2	PostgreSQL Tutorial	ISBN-13	978-1449370000
3	PostgreSQL Tutorial	language	English
4	PostgreSQL Tutorial	paperba...	243
5	PostgreSQL Tutorial	publisher	postgresqtutorial.com
6	PostgreSQL Cheat Sheet	weight	1 ounces
7	PostgreSQL Cheat Sheet	ISBN-13	978-1449370001
8	PostgreSQL Cheat Sheet	language	English
9	PostgreSQL Cheat Sheet	paperba...	5
10	PostgreSQL Cheat Sheet	publisher	postgresqtutorial.com

# Hstore sous PostgreSQL : conclusion

- 😊 Combinaison relationnel et clé-valeur
- 😊 Gestion des transactions
- 😊 Persistance des données
- 😞 Pas fait pour gérer les données en mémoire
- 😞 Pas les mêmes types de données et le même usage

Performance par rapport à Redis ?

cf. ex. de résultats expérimentations contradictoires <https://www.cybertec-postgresql.com/en/postgresql-vs-redis-vs-memcached-performance/>,

<https://www.peterbe.com/plog/redis-vs-postgres-blob-of-json> et <https://profil-software.com/blog/development/database-comparison-sql-vs-nosql-mysql-vs-postgresql-vs-redis-vs-mongodb/>