NoSQL бази даних

NoSQL базы данных: понимаем суть. 27.09.2012 http://habrahabr.ru/post/152477/
Реляционные базы данных обречены? 28.01.2011, http://habrahabr.ru/post/103021/

Масштабування систем

Упродовж усього існування реляційних БД вони постійно пропонували найкращу суміш простоти, стійкості, гнучкості, продуктивності, масштабованості і сумісності в сфері управлінні даними. Але їх показники по кожному з цих пунктів не обов'язково вище, ніж у аналогічних систем, орієнтованих на якусь одну особливість.

Сьогодні серед особливостей на перше місце виходить масштабованість. Оскільки все більше застосувань працюють в умовах високого навантаження, наприклад, таких як веб-сервіси, їх вимоги до масштабованості можуть дуже швидко змінюватись, зокрема сильно зростати.

У розподілених системах планування потужності з похибкою менше двох порядків абсурдне. Створений вами сайт несподівано стає популярним і рахунок відвідувачів буквально йде на мільйони. Так само просто (і з такими ж катастрофічними наслідками) можливий і інший варіант: ви створюєте сайт, яким ніхто особливо не цікавиться. Дороге устаткування простоює, споживаючи гроші за електроенергию і адміністрування. У мережевому світі перехід сайту з одного стану в інший відбувається за хвилини.

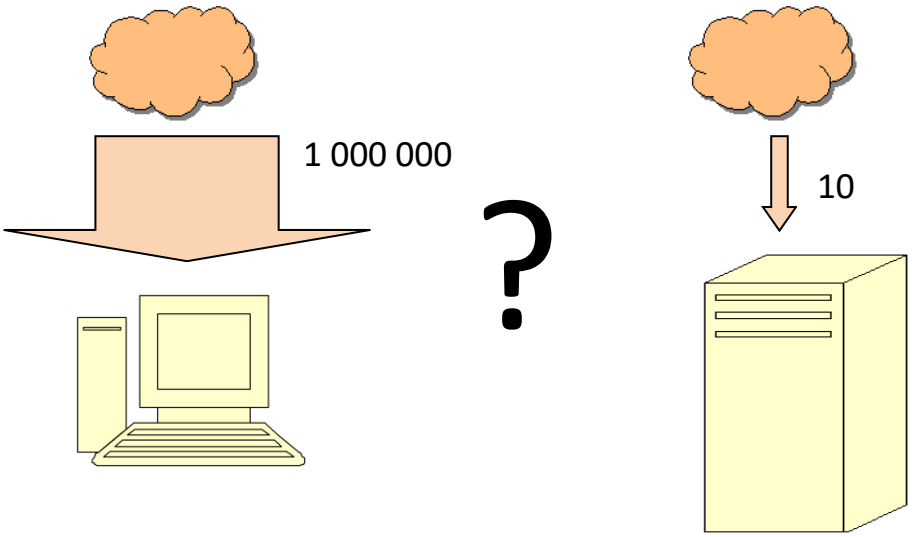


Рисунок 1. Проблема масштабування

Реляційні БД добре масштабуються тільки у тому випадку, якщо розташовуються на одному сервері. Коли ресурси цього сервера закінчаться, необхідно буде додати більше машин і розподілити навантаження між ними. І ось тут складність реляційних БД починає грати проти масштабованості. Якщо ви спробуєте збільшити кількість серверів не до декількох штук, а до сотні або тисячі, складність зросте на порядок.

Тому доводиться застосовувати інші типи баз даних, як-от NoSQL, які мають більш високу здатність до масштабування, хай і ціною відмови від інших можливостей, доступних в реляційних БД.

CAP теорема

У 2000 році професор Каліфорнійського університету в Берклі Брюера (Eric Brewer) запропонував так звану теорему CAP, у комп'ютерних науках також відому як теорема Брюера. Вона стверджує, що ***неможливо* для розподіленої комп'ютерної системи одночасно забезпечити всі три наступні гарантії :**

Consistency (*цілісність*, усі вузли бачать однакові дані в один час).

Availability (*доступність*, гарантія, що кожен запит отримує відповідь про те, чи він виконаний успішно, чи невдало).

Partition tolerance (*стійкість до розділення*, система продовжує працювати незважаючи на довільну втрату повідомлення або відмову частини системи).

У 2002 році Сет Джилберт і Ненсі Лінч з Массачусетського технологічного інституту (MIT) підібрали формальні моделі асинхронних і синхронних розподілених обчислень, у рамках яких показано виконання теореми CAP в умовах відсутності синхронізації (загального годинника) у вузлів розподіленої системи і принципову можливість компромісу в частково синхронних системах. Надалі практики стали посилатися на цю роботу як на доведення теореми CAP.

Наслідок: розподілені системи залежно від пари практично підтримуваних властивостей з трьох можливих розпадаються на три класи.

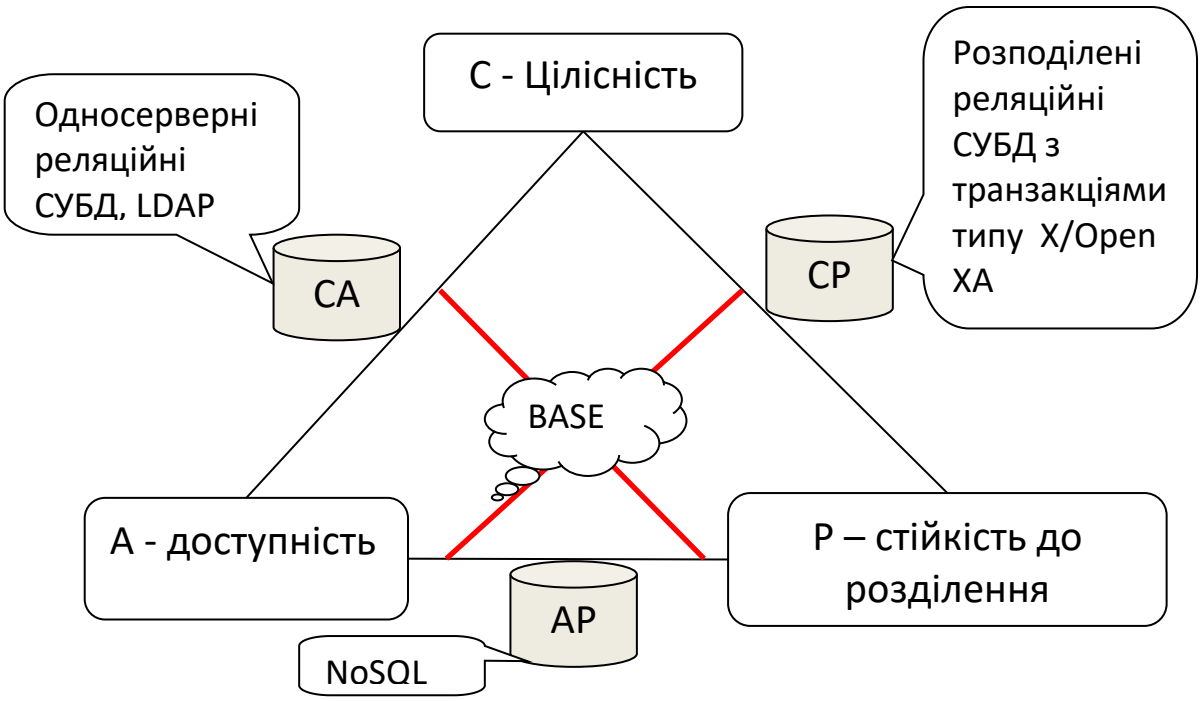


Рисунок 2. Три класи розподілених систем як наслідок теореми CAP

CA

Система, в усіх вузлах якої дані погоджені і забезпечена доступність, жертвує стійкістю до розпаду на секції. Такі системи можуть підтримувати транзакції в сенсі ACID. Приклади таких систем: односерверна система на основі реляційної СУБД; розподілена служба каталогів LDAP.

CP

Розподілена система, яка в кожний момент забезпечує цілісний результат і здатна функціонувати в умовах розпаду, може бути недоступною. Стійкість до розпаду на секції вимагає забезпечення дублювання змін в усіх вузлах системи, тому застосовуються розподілені песимістичні блокування. Тоді розподілені транзакції відповідають умовам ACID, але прийнятна тривалість відгуку не гарантується. Схоже на специфікацію обробки розподілених транзакцій X/Open XA в застосуванні до реплікації.

AP

Розподілена система, що відмовляється від цілісності результату. Хоча системи такого роду відомі задовго до формулювання принципу CAP (наприклад, розподілені веб-кеші або DNS), зростання популярності систем з цим набором властивостей зв'язується саме з поширенням теореми CAP. Так, більшість NoSQL- систем принципово не гарантують цілісності даних, і посилаються на теорему CAP як на мотив такого обмеження. Завданням при побудові AP- систем стає забезпечення деякого практично доцільного рівня цілісності даних, в цьому сенсі про AP- системи говорять як про "цілісних у підсумку"(англ. eventually consistent).

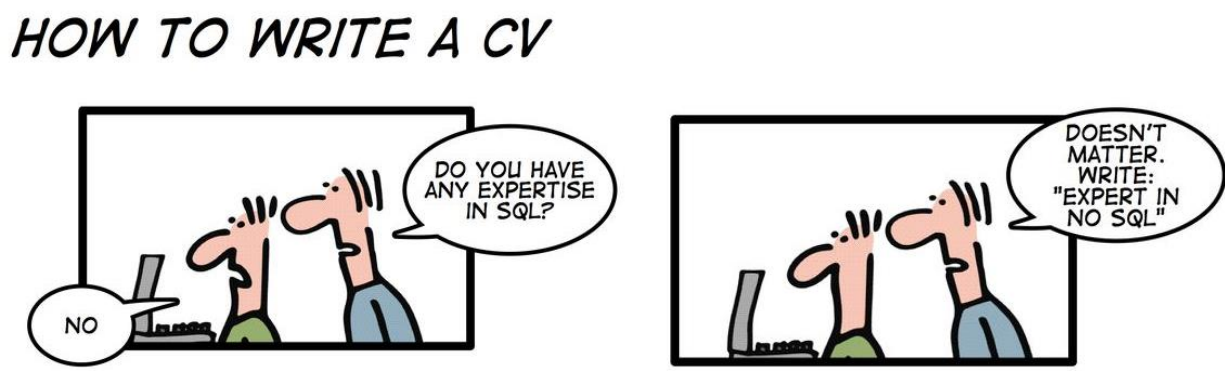
BASE-архітектура

Цей підхід сформульовано в другій половині 2000-х років. Вимоги цілісності і доступності тут забезпечуються не в повній мірі.

- BA – Basically Available, базова доступність. Збій в деяких вузлах призводить до відмови в обслуговуванні тільки для незначної частини сесій при збереженні доступності у більшості випадків.
- S – Soft-state, нестійкий стан. Не зберігається довго інформація стану сесій (проміжні результати виборок, інформація про навігацію, контекст), фіксується оновлення лише критичних операцій.
- E - eventually consistency, узгодженість у підсумку. Можливість суперечливості даних в деяких випадках, але при забезпеченні узгодження в практично осяжний час.

Сутність NoSQL

Термін "NoSQL" має стихійне походження і не має загальновизнаного визначення або наукової установи за спиною. Ця назва швидше характеризує вектор розвитку IT убік від реляційних баз даних. Розшифровується як Not Only SQL, хоча є прибічники і прямого визначення No SQL.



Інша розповсюджена назва – сховище типу ключ-значення. Цей тип БД можна зустріти в контексті :

- документо-орієнтованих БД,
- атрибутно-орієнтованих БД,
- шардованих упорядкованих масивів,
- розподілених хеш-таблиць.

Розглянемо властивості баз NoSQL, які є для них спільними, крім графових NoSQL баз.

Не використовується SQL

Мається на увазі ANSI SQL DML, хоча багато NoSQL СУБД намагаються використовувати схожі мови запитів. Тому NoSQL насправді слід розуміти буквально.

Неструктурованість (schemaless)

На відміну від реляційних баз структура даних не регламентована (або слабко типізована в термінах мов програмування). В окремому рядку або документі можна додати довільне поле без попередньої декларативної зміни структури усієї таблиці. Зміна даних виконується кодом застосунку.

Позитивний наслідок неструктурованості – ефективність роботи з розрідженими даними. Якщо в одному документі поле потрібно, а в другому ні, то це поле не буде зберігатись у другому документі.

Негативні наслідки неструктурованості:

- Застосунок має враховувати, що поле в документі не завжди є.
- Відсутність обмежень з боку бази (not null, unique, check constraint і т.п.).
- Виникають складності в розумінні і контролі структури бази при паралельній роботі з базою різних проектів. Втім, не виключено, що це перевага.

Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Color: Green Year:2005 Transmission: Auto

Рисунок 3 - Приклад типового домену (колекції) ключ-значення

Подання даних у вигляді агрегатів

На відміну від реляційної моделі, яка зберігає логічні бізнес-сутності застосування в різні фізичні таблиці в цілях нормалізації, NoSQL сховища оперують з цими сутностями як з цілісним об'єктом:

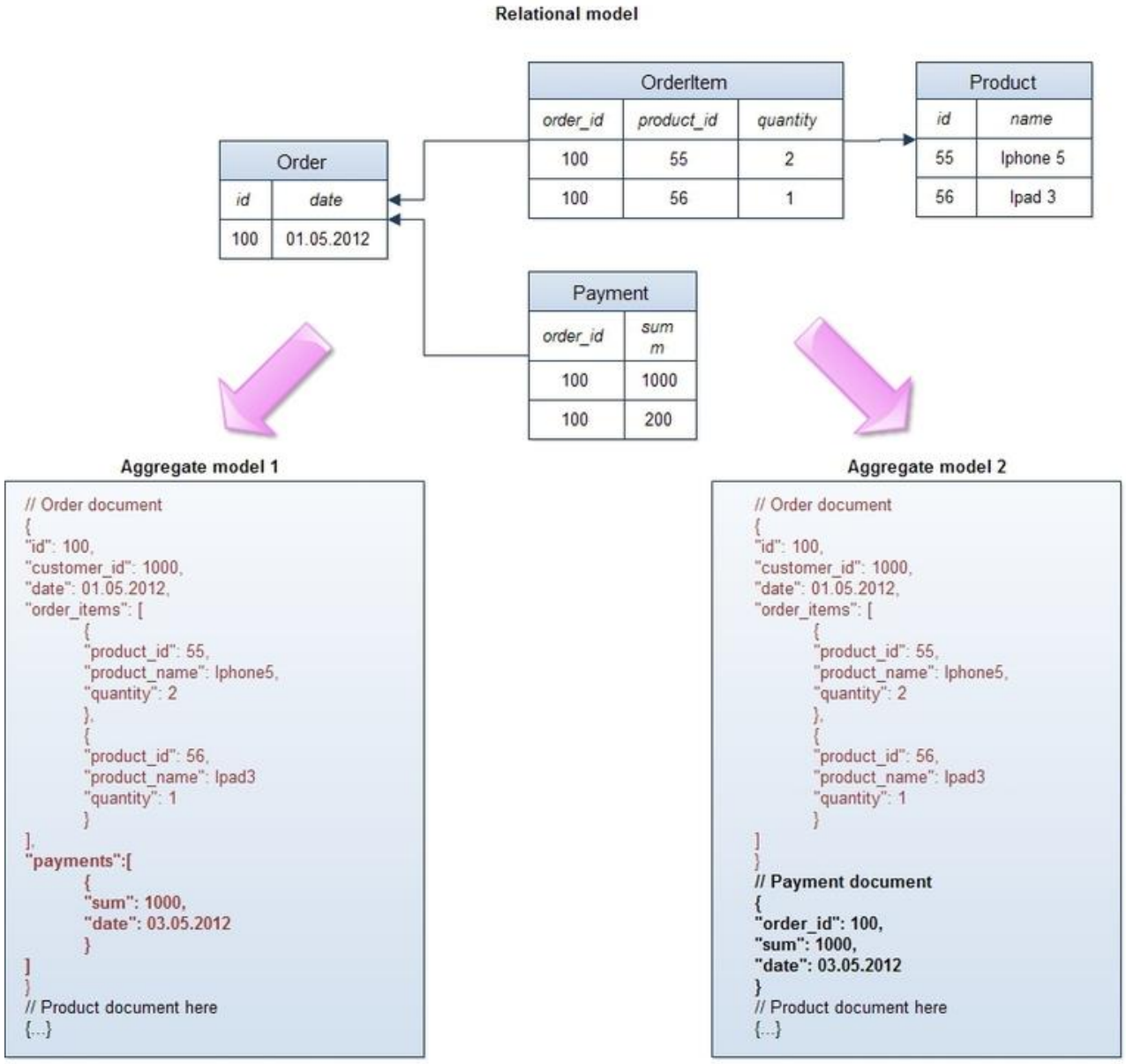


Рисунок 4 - Приклад моделей даних для NoSQL бази MongoDB в порівнянні з реляційною моделлю

В цьому прикладі продемонстровані агрегати для стандартної концептуальної реляційної моделі e-commerce "замовлення - позиції замовлення - платежі - продукт". У обох випадках замовлення об'єднується з позиціями в один логічний об'єкт, при цьому кожна позиція зберігає в собі посилання на продукт і деякі його атрибути, наприклад, назва (така денормалізація потрібна, щоб не просити об'єкт продукту при витяганні замовлення - головне правило розподілених систем - мінімум "JOIN" між об'єктами).

У одному агрегаті платежі об'єднані із замовленням і є складовою частиною об'єкту, в іншому - винесені в окремий об'єкт. Цим демонструється головне правило проєктування

структури даних в NoSQL базах - вона повинна підлягати вимогам застосування і бути максимально оптимізованою під найчастіші запити.

- Якщо платежі регулярно витягаються разом із замовленням - має сенс їх включати в загальний об'єкт,
- якщо ж багато запитів працюють тільки з платежами - краще їх винести в окрему сутність.

Робота з великими ненормалізованими об’єктами тягне проблеми в разі довільних запитів, що не вкладаються у структуру агрегатів. Наприклад, бізнес просить нас підрахувати, скільки одиниць певного продукту було продано минулого місяця. Доведеться витягати всі рядки кожного замовлення в NoSQL сховище, хоча потрібні рядки лише одного товару. Але така нормалізація, як у звичайній реляційній БД, з винесенням рядків замовлень в окрему таблицю, в розподіленій системі тягне ще більшу затримку.

Плюси і мінуси обох підходів зведемо в таблицю:

Нормалізовані дані	*Дані як агрегати*
• Цілісність при оновленні. Відсутні аномалії. • Орієнтованість на широкий спектр запитів	• Оптимізація лише під певний тип запитів • Складнощі при оновленні денормалізованих даних
• Неефективність в розподіленому середовищі • Низька швидкість при використанні об’єднань (JOIN) • Невідповідність об’єктної моделі застосування фізичній структурі даних	• Найвища швидкість читання в розподіленому середовищі • Можливість зберігати об’єкти в тому вигляді, в якому з ними працює застосунок • Природня підтримка атомарності на рівні записів (документів)

Слабкі ACID властивості

З виникненням величезних масивів інформації та розподілених систем стало ясно, що забезпечити для них транзакційність набору операцій з одного боку і отримати високу доступність і швидкий час відгуку з іншого - неможливо. Більше того, навіть оновлення одного запису не гарантує, що будь-який інший користувач вмить побачить зміни в системі, адже зміна може статися, наприклад, в головному вузлі, а репліка асинхронно скопіюється на підлеглий вузол, з яким працює інший користувач, і він побачить результат через якийсь проміжок часу. Це і є eventual consistency і це те, на що йдуть зараз усі найбільші інтернет-компанії світу, включаючи Facebook і Amazon. Останні з гордістю заявляють, що максимальний інтервал, впродовж якого користувач може бачити неконсистентні дані, триває не більше секунди. Приклад такої ситуації показаний на малюнку:

Рисунок 5 - Приклад наслідку «цілісності в підсумку»: двічі заброньований номер у готелі

Що ж робити системам, які класично пред'являють високі вимоги до атомарності-консистентності операцій і в той же час потребують швидких розподілених кластерів, - фінансовим, інтернет-магазинам і т.д? Практика показує, що ці вимоги вже давно неактуальні: ось що сказав один розробник фінансової банківської системи : «Якби ми дійсно чекали завершення кожної транзакції у світовій мережі ATM(банкоматів), транзакції займали б стільки часу, що клієнти втікали б геть в люті. Що відбувається, якщо ти і твій партнер знімаєте гроші одночасно і перевищуєте ліміт? - Ви обоє отримаєте гроші, а ми виправимо це пізніше.»

Інший приклад - бронювання готелів, показаний на рисунку 5. Онлайн-магазини, чия політика роботи з даними припускає eventual consistency, зобов'язані передбачити заходи на випадок таких ситуацій (автоматичне вирішення конфліктів, відкат операції, оновлення з іншими даними). На практиці готелі завжди намагаються тримати "пул" вільних номерів на непередбачений випадок і це може стати рішенням спірної ситуації.

Розподілені системи без сумісно використовуваних ресурсів (share nothing)

Можливо, це головна властивість, заради якої обирають NoSQL бази. Адже, нагадаємо, одночасно з лавиноподібним зростанням інформації у світі і необхідності її обробляти за розумний час зростання швидкості процесора зупинилося на 3.5 Ггц, швидкість читання з диска також росте тихими темпами, плюс ціна потужного сервера завжди більше сумарної ціни декількох простих серверів.

Поки що єдиний вихід з ситуації - горизонтальне масштабування, коли декілька незалежних серверів з'єднуються швидкою мережею і кожен володіє/обробляє тільки частину даних і/або тільки частину запитів на читання-оновлення. У такій архітектурі для підвищення потужності сховища (місткості, часу відгуку, пропускної спроможності) необхідно лише додати новий сервер в кластер - і все. Процедурами шардінгу, реплікації, забезпеченням відмовостійкості (результат буде отриманий навіть якщо один або декілька серверів перестали відповідати), перерозподілу даних у разі додавання вузла займається сама NoSQL база.

Властивості розподілених NoSQL баз:

Реплікація

Це копіювання даних на інші вузли при оновленні. Кожний вузол містить повну або часткову копію БД. Реплікація дозволяє як досягти більшої масштабованості, так і підвищити доступність і збереженість даних. Прийнято розглядати реплікацію двох видів: master - slave:

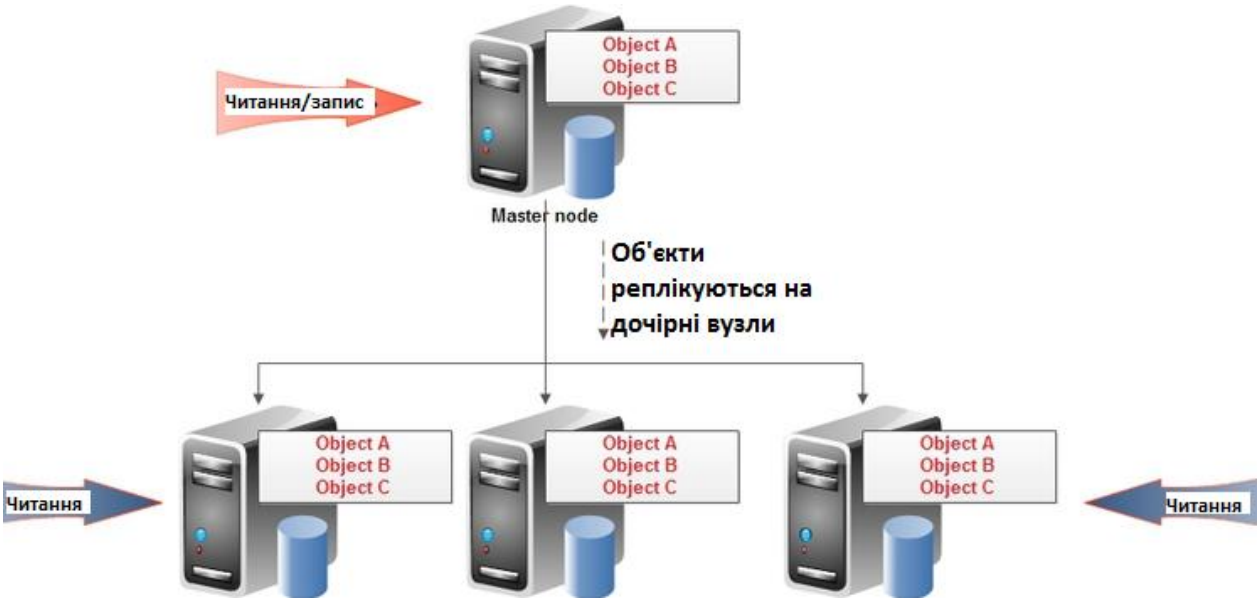


Рисунок 6 - Реплікація виду master - slave

Перший тип передбачає хорошу масштабованість на читання (може відбуватися з будь-якого вузла), але немасштабований запис (тільки у master-вузол). Також є особливості з забезпеченням постійної доступності (у разі падіння master або вручну, або автоматично на його місце призначається один з вузлів, що залишилися).

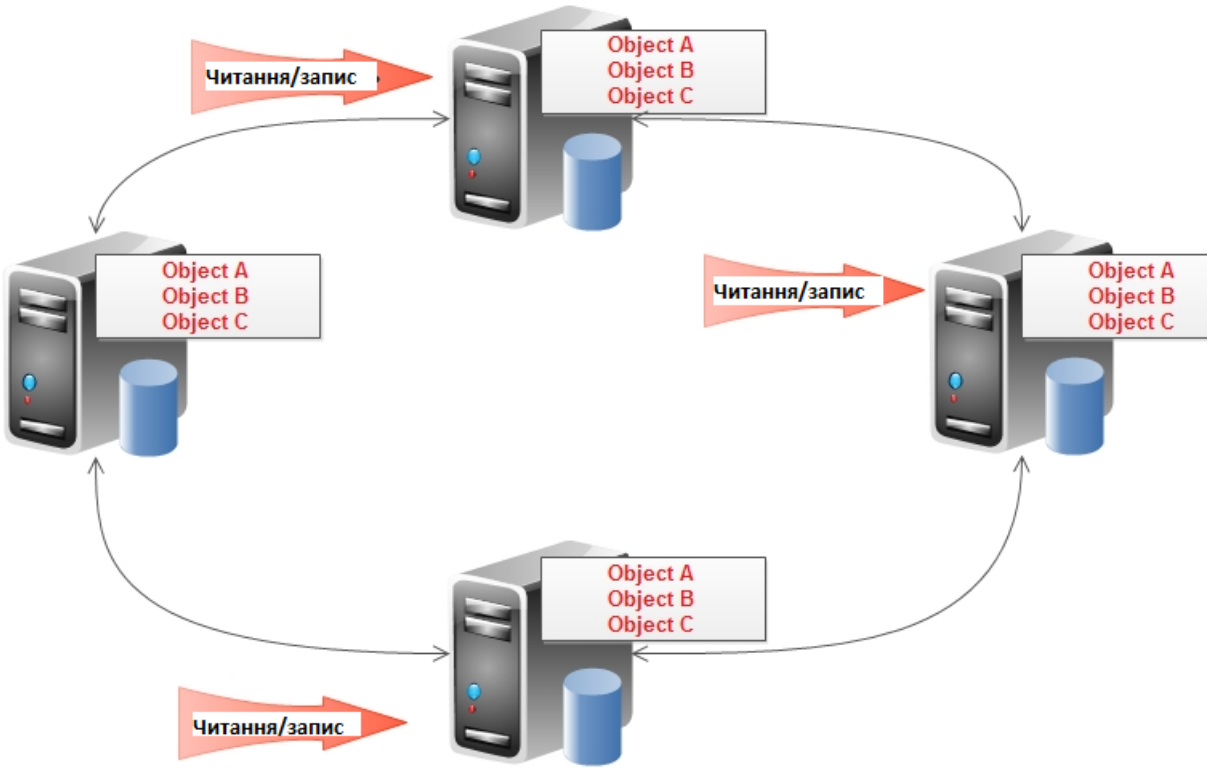


Рисунок 7 - Реплікація виду peer-to-peer

Для другого типу реплікації передбачається, що усі вузли рівні, містять повну копію БД і можуть обслуговувати як запити на читання, так і на запис. Цей принцип закладений також у технологію *блокчейн*.

Реплікація передбачена також і в реляційних БД.

Шардінг

Це вбудовані засоби для рознесення даних по ключовому полю на кластер з декількох машин, для створення ефективного балансування навантаження.

У цьому разі користувацький застосунок *розділяє* дані по декількох незалежних базах даних і при запиті відповідних даних користувачем звертається до конкретної бази. Наприклад, таблиця розділяється по діапазонах ключів і розміщується в різних БД. Шардінг часто використовується також в реляційних БД для збільшення швидкості і пропускної спроможності. В РСУБД подекуди це називається *секціонування*.

У NoSQL базах даних шардінг, як і реплікація, виконуються автоматично самою БД.

Рисунок 8 - Шардінг

Резюме

NoSQL бази набувають популярності шаленими темпами. Це не означає, що реляційні бази даних стають чимось архаїчним, вони використовуватимуться як і раніше активно. Ми вступаємо в еру, коли для різних потреб використовуватимуться різні сховища даних. Тепер немає монополізму реляційних баз, як безальтернативного джерела даних. Все частіше архітектори вибирають сховище виходячи з природи самих даних і того, як ми ними хочемо маніпулювати, які об'єми інформації очікуються.

Зокрема, ось причини, чому варто обрати NoSQL базу:

- Дані сильно документо-орієнтовані, і більше підходять для моделі даних ключ-значення, ніж для реляційної моделі. Тобто дані складаються з самодостатніх об’єктів.
- Прикладна область сильно об’єктно-орієнтована, тому використання сховища типу ключ-значення спростить код для перетворення даних в об’єкти і навпаки.
- Проблема №1 – необхідність високої масштабованості відносно головного запиту.

MongoDB

Ця БД є «типовою» не-графовою NoSQL базою. Всі розглянуті ознаки для неї дійсні.
У порівнянні з реляційними СУБД MongoDB має іншу термінологію:

Реляційні СУБД	*MongoDB*
База даних	База даних
Таблиця	Колекція
Рядок таблиці, запис, кортеж	Документ
Атрибут, колонка таблиці, поле запису	Поле
Секціонування	Шардінг

Документи надаються користувачу в BSON-форматі, який є надмножиною формату JSON – JavaScript Object Notation, тобто формат опису об'єктів згідно синтаксису JavaScript (рис. 4 – Приклад моделей даних). Призначення JSON (BSON - binary JSON) аналогічне XML – серіалізація документу, тобто перетворення його в послідовність бітів для запису в файл і передачі по мережі.

Порівняймо деякі операції в SQL та MongoDB.
SQL to MongoDB Mapping Chart

SQL Statements	MongoDB Statements
INSERT INTO people(user_id, age, status) VALUES ("bcd001", 45, "A")	db.people.insertOne({ user_id: "bcd001", age: 45, status: "A" })
SELECT * FROM people	db.people.find()
SELECT * FROM people WHERE status = "A"	db.people.find({ status: "A" })
SELECT * **FROM people** **WHERE status != "A"**	db.people.find({ status: { $ne: "A" } })
SELECT * **FROM people** **WHERE status = "A"** **AND age = 50**	db.people.find({ status: "A", age: 50 })
SELECT * **FROM people** **WHERE status = "A"** **OR age = 50**	db.people.find({ $or: [{ status: "A" } , { age: 50 }] })
SELECT * **FROM people** **WHERE age > 25** **AND age <= 50**	db.people.find({ age: { $gt: 25, $lte: 50 } })
SELECT * **FROM people** **WHERE status = "A"** **ORDER BY user_id ASC**	db.people.find({ status: "A" }).sort({ user_id: 1 })
SELECT COUNT(*) FROM people	db.people.count() *or* db.people.find().count()
SELECT COUNT(*) **FROM** people **WHERE** age > 30	db.people.count({ age: { $gt: 30 } }) *or* db.people.find({ age: { $gt: 30 } }).count()

Шардінг

MongoDB Documentation Release 2.6.4. https://ru.scribd.com/doc/242299191/MongoDB-manual-pdf
MongoDB Documentation Release 3.0. MongoDB Inc., February 08, 2016, https://docs.mongodb.com/v3.0/

Системи баз даних з великими наборами даних та високопродуктивними застосуваннями можуть вичерпати можливості одного сервера.

- Висока інтенсивність запитів може вичерпати ємність ЦП на сервері.
- Великі набори даних можуть перевищити можливості зберігання на дисках однієї машини.
- Нарешті, робочі набори мають розміри більші, ніж оперативна пам'ять системи, можуть вичерпати вхідний / вихідний потенціал дискових накопичувачів.

Для відповіді на ці прояви масштабування в системах баз даних є два основних підходи: **вертикальне масштабування** та **шардінг**.
Вертикальне масштабування передбачає додавання більшої кількості ресурсів процесора та пам’яті для збільшення пропускної спроможності. Масштабування за рахунок додавання можливостей має обмеження: високопродуктивні системи з великою кількістю процесорів і великою кількістю оперативної пам'яті **непропорційно дорожчі**, ніж невеликі системи. Крім того, постачальники хмарних сервісів можуть дозволити користувачам менші екземпляри БД. В результаті є практична верхня межа для вертикального масштабування, тобто розміру одної БД.
Шардінг або горизонтальне масштабування, навпаки, розподіляє набір даних і поширює дані по декількох серверах, або шардах. Кожен шард є незалежною базою даних, і колективно вони складають єдину логічну базу даних.
Для розподілу даних і трафіку застосування в шардованій колекції MongoDB використовує shard-ключі. Вибір правильного shard-ключа суттєво впливає на продуктивність. В реляційних базах аналогом є розподілення, або секціонування таблиць.

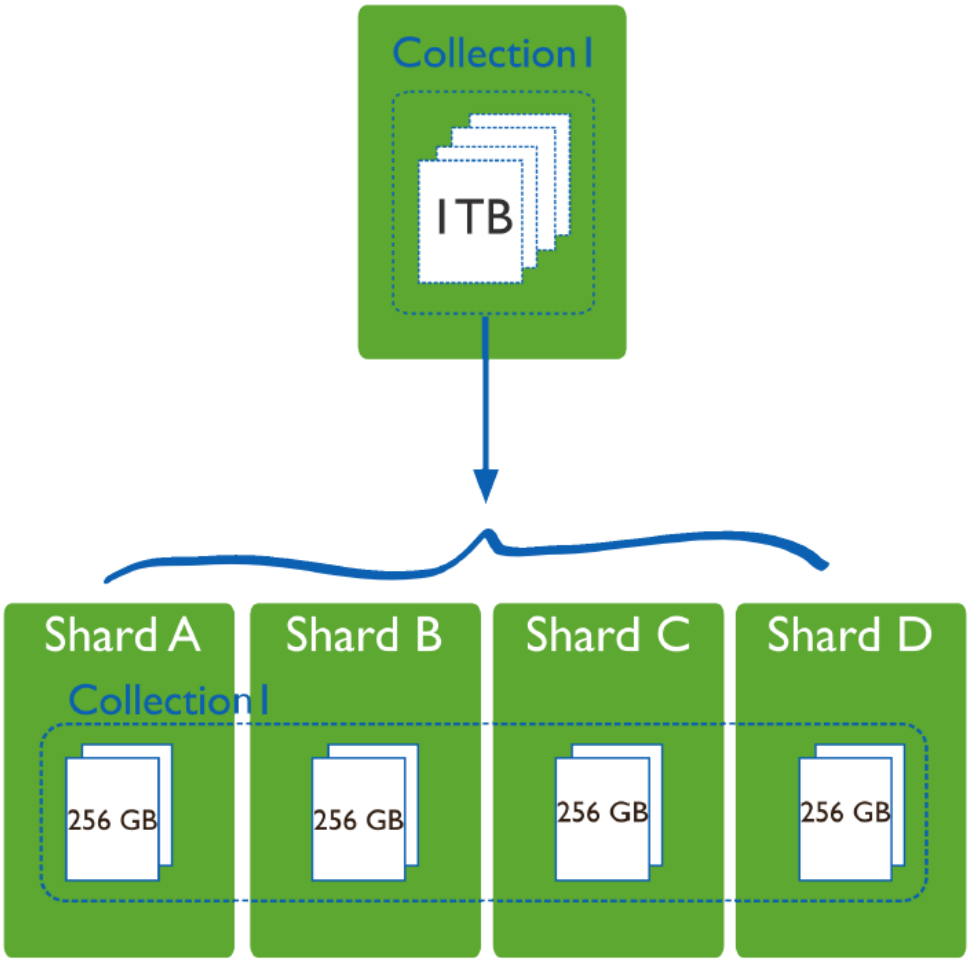


Рисунок 10 – Шардінг – горизонтальне масштабування

Шардінг задовольняє вимоги масштабування щодо підтримки високої продуктивності та великих наборів даних наступним чином:

- Шардінг зменшує кількість операцій на кожному шарді. У результаті кластер може збільшити пропускну спроможність та горизонтальну продуктивність. Наприклад, щоб вставити дані, програмі потрібно лише отримати доступ до шарду, відповідальному за цей запис.
- Шардінг зменшує кількість даних, які повинен зберігати кожен сервер. Кожний шард зберігає менше даних, в той час як кластер росте.

Наприклад, якщо в базі даних є 1 терабайт даних, і є 4 шарди, тоді кожен шард може мати лише 256 Гб даних. Якщо є 40 шардів, тоді кожен шард може мати лише 25 Гб даних.

Шардінг у MongoDB

MongoDB підтримує шардінг шляхом конфігурування *шардованих кластерів*.

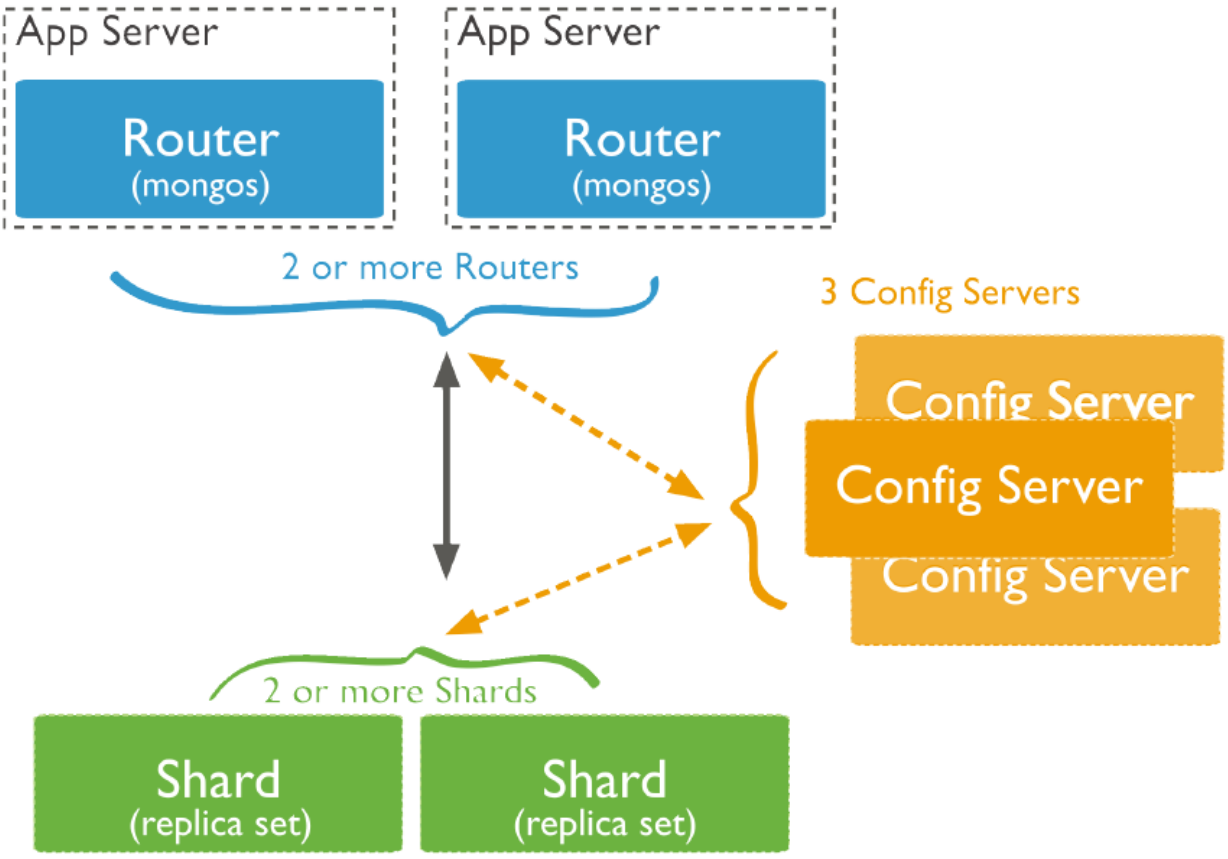


Рисунок 11 – Шардований кластер у MongoDB

Шардований кластер має наступні компоненти:

- Шарди;
- Маршрутизатори запитів (query routers);
- Сервери конфігурації (config servers).

Шарди зберігають дані. Для забезпечення високої продуктивності і цілісності даних у продуктивному шардованому кластері кожний шард є реплікою (replica set).

Маршрутизатор запитів, або екземпляр Mongos, є інтерфейсом між клієнтськими застосуваннями та прямими операціями до відповідного шарду або шардів. Протокол маршрутизатора обробляє і направляє операції на шарди, а потім повертає результати клієнтам. Шардований кластер може містити більше одного маршрутизатора запитів, щоб розділити завантаження запитами клієнтів. Клієнт надсилає запити на один маршрутизатор запитів. У більшості шардованих кластерів є багато маршрутизаторів запитів.

Сервери конфігурації зберігають метадані кластера. Ці дані містять відображення даних кластера, встановлених на шарди. Маршрутизатор запитів використовує ці метадані для спрямування операцій на конкретні шарди. Продуктивні шардовані кластери мають рівно 3 конфігураційні сервери.

Секціонування даних (Data Partitioning)

MongoDB розділяє дані, або шардує, на рівні колекції, за допомогою ключа шардування (shard key).

Ключі шардування

Щоб шардувати колекцію, потрібно вибрати ключ шардування. Це має бути або індексоване поле, або індексоване складене поле, яке існує в кожному документі колекції. MongoDB розподіляє значення ключів шардування на фрагменти (chunks) та рівномірно розподіляє фрагменти між шардами. Щоб розподілити значення шардованого ключа на фрагменти, MongoDB використовує розбиття на основі діапазонів значень ключа або на основі хешування значення ключа.

Шардування на основі діапазонів

MongoDB розділяє набір даних на діапазони, визначені значеннями ключа шардування. Увесь діапазон можливих значень ключа розділяється на менші діапазони, що не перетинаються, - фрагменти, і кожному фрагменту відповідає діапазон від мінімального до максимального значення ключа шардування.

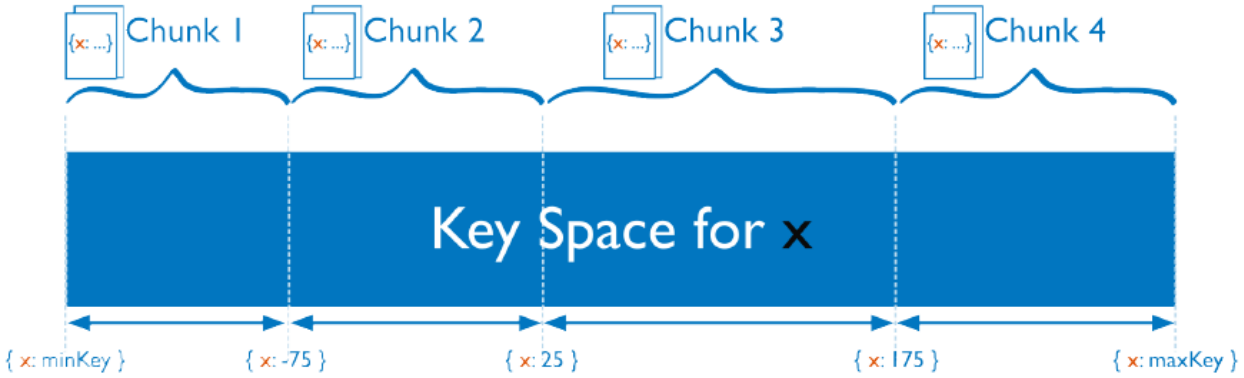


Рисунок 12 – Шардування на основі діапазонів

Шардування на основі хешування

MongoDB обчислює хеш-функцію від значення ключа шардування, та використовує це значення для створення фрагментів. З розбиттям на основі хешування два документи з "близькими" значеннями ключа навряд чи будуть частиною одного фрагменту. Це забезпечує більш рівномірний псевдовипадковий розподіл колекції в кластері.

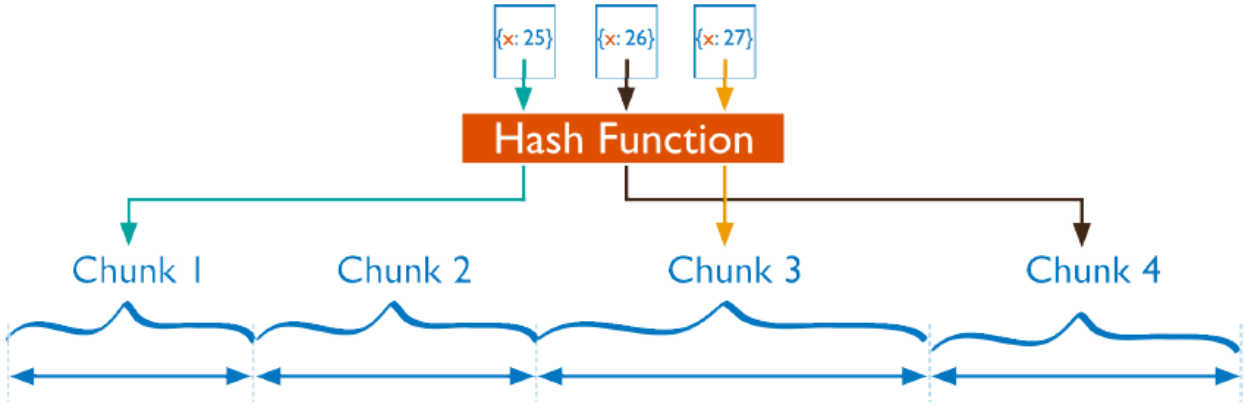


Рисунок 13 – Шардування на основі хешування

Відмінності в продуктивності між двома способами шардування

Розбиття на основі діапазону підтримує більш ефективні запити по діапазонах. Отримуючи запит по діапазону ключа шардування, маршрутизатор запитів може легко визначити, які фрагменти перекривають цей діапазон, і направляє запит лише на ті шарди, які містять ці фрагменти.

Однак розбиття на основі діапазону може призвести до нерівномірного розподілу даних, що може нівелювати деякі переваги шардування. Наприклад, якщо ключ шардування є лінійно зростаючим полем, таким як час, то всі запити за певний часовий проміжок будуть направлені до одного фрагменту і, таким чином, до одного шарду. У цій ситуації невелика кількість шардів може отримати більшість запитів, і система буде масштабована погано.
Хеш-шардування, навпаки, забезпечує рівномірний розподіл даних. Хешування значень ключів призводить до випадкового розподілу даних за фрагментами і внаслідок за шардами. Але випадковий розподіл робить більш ймовірним, що запит по діапазону ключа шардування не зможе піти до декількох фрагментів, але, швидше за все, буде запитувати кожний шард для повернення результату.