МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

К. О. Радченко,
А. В. Петрашенко

БАЗИ ДАНИХ NOSQL

КОНСПЕКТ ЛЕКЦІЙ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Інженерія програмного забезпечення
мультимедійних та інформаційно-пошукових систем»
спеціальності 121 Інженерія програмного забезпечення

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензент — *Клятченко Я. М.*, канд. техн. наук, доц. кафедри системного програмування і спеціалізованих комп'ютерних систем факультету прикладної математики НТУУ «КПІ ім. Ігоря Сікорського»

Відповідальний редактор — *Тесленко О. К.*, канд. техн. наук, доц. кафедри системного програмування і спеціалізованих комп'ютерних систем факультету прикладної математики НТУУ «КПІ ім. Ігоря Сікорського»

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № X від DD.12.2022 р.)
за поданням Вченої ради факультету прикладної математики
(протокол № X від DD.12.2022 р.)

Бази даних NoSQL. Конспект лекцій [Електронний ресурс]: навч. посіб. для студ. спеціальності 121 «Інженерія програмного забезпечення», освітньої програми «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем», які вивчають навчальну дисципліну «Програмне забезпечення інформаційно-пошукових систем 1. Бази даних NoSQL» / К. О. Радченко, А. В. Петрашенко; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 3,65 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2022. – 270 с.

У навчальному посібнику наведено матеріали, необхідні для практичної апробації теоретичних знань з організації баз даних NoSQL і використання їх у сучасних інформаційних системах. Подано блок навчально-методичного забезпечення, який включає перелік знань і вмінь по кожній темі, термінологічний словник, фрагменти навчального коду і завдання для самоконтролю студентами засвоєння знань.

Конспект лекцій призначено для студентів вищих навчальних закладів, що навчаються за спеціальністю спеціальності 121 «Інженерія програмного забезпечення», освітньої програми «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем» та вивчають навчальну дисципліну «Програмне забезпечення інформаційно-пошукових систем 1. Бази даних NoSQL».

Реєстр. № НП XX/XX-XXX. Обсяг 11,5 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© К. О. Радченко, А. В. Петрашенко
© КПІ ім. Ігоря Сікорського, 2022

Зміст

Вступ 7
Лекція 1. Основи концепції баз даних NoSQL 9
Перелік знань і вмінь 9
Сильні та слабкі сторони реляційних СУБД 10
Термінологічний словник 11
Теорема CAP 17
Висновок 19
Завдання для індивідуальної роботи 20
Лекція 2. Основи Riak 21
CRUD, посилання та типи MIME 22
REST і Riak 24
Збереження нового значення за допомогою дієслова PUT 25
Посилання 26
Типи MIME в Riak 30
Висновок 31
Завдання для індивідуальної роботи 31
Лекція 3. Riak. Mapreduce та кластери серверів 32
Вступ до Mapreduce 32
Функції Mapreduce в Riak 35
Слідування за посиланнями за допомогою mapreduce 39
Про узгодженість та довговічність 40
Кільце Riak 41
Вузли та операції читання-запису 43
Запис та довговічний запис 46
Висновок 47
Завдання для індивідуальної роботи 49
Лекція 4. Riak. Вирішення конфліктів та розширення Riak 50
Вирішення конфліктів за допомогою векторного годинника 50
Розширення Riak 56
Висновок 59
Завдання для індивідуальної роботи 60
Лекція 5. HBase 61
Введення в HBase 61
CRUD та адміністрування таблиць 62
Конфігурація HBase 63
Оболонка HBase 64
Створення таблиці 65

Додавання даних із програми....69
Робота з «великими даними». Імпорт даних, виконання скриптів....71
Механізм горизонтального масштабування HBase на диску....76
Висновок....84
Завдання для індивідуальної роботи....85
Лекція 6. Робота у хмарі з HBase....87
Встановлення Thrift....87
Розробка клієнтської програми....89
Введення в Whirr....90
Налаштування та запуск кластера....92
Висновок....95
Завдання для індивідуальної роботи....97
Лекція 7. Основи MongoDB....99
Mongo....99
Операції CRUD та вкладеність....100
JavaScript....102
Читання з Mongo....103
Оновлення....108
Посилання....109
Видалення....110
Функціональні критерії....111
Висновок....112
Завдання для індивідуальної роботи....112
Лекція 8. Індексування, угруповання, mapreduce у MongoDB....114
Індексування: коли швидкодії не вистачає....114
Агреговані запити....117
Команди на стороні сервера....119
Mapreduce (і Finalize)....122
Висновок....125
Завдання для індивідуальної роботи....125
Лекція 9. Набори реплік, сегментування, просторові дані, GridFS у MongoDB....126
Репліки....126
Сегментування....130
Просторові запити....133
GridFS....134
Висновок....135
Завдання для індивідуальної роботи....136
Лекція 10. Основи CouchDB....138
Інтерфейс Futon....139
Виконання операцій CRUD за допомогою REST-інтерфейсу та cURL....142

Реалізація представлень 145
Імпорт даних у CouchDB за допомогою програми на Ruby 152
Висновок 157
Завдання для індивідуальної роботи 157
Лекція 11. Редукція, Changes API й реплікація даних за допомогою CouchDB 159
Створення складніших представлень за допомогою редукторів 159
Відстеження змін у CouchDB 161
Реплікація даних у CouchDB 166
Висновок 170
Завдання для індивідуальної роботи 172
Лекція 12. Основи Neo4J 173
Графи, Groovy та операції CRUD 173
Neo4j та Gremlin 176
Конвеєри 179
Подальша взаємодія з графом 183
Предметно-орієнтовані кроки 187
Висновок 189
Завдання для індивідуальної роботи 190
Лекція 13. REST, індекси та алгоритми у Neo4j 191
REST-інтерфейс 191
Алгоритми Gremlin 197
Висновок 203
Завдання для індивідуальної роботи 203
Лекція 14. Розподіленість та висока доступність Neo4j 204
Транзакції 204
Висока доступність 205
HA-кластер 206
Резервне копіювання 211
Висновок 213
Завдання для індивідуальної роботи 214
Лекція 15. Основи Redis 215
Сховище сервера структур даних 215
Операції CRUD та типи даних 216
Транзакції 218
Простір імен 228
Висновок 229
Завдання для індивідуальної роботи 230
Лекція 16. Більш складні застосування, розподілені обчислення у Redis 231
Інтерфейс Redis 231
Налаштування Redis 234

Реплікація головний-підлеглий 238
Фільтри Блума 243
Висновок 246
Завдання для індивідуальної роботи 247
Лекція 17. Комбінування Redis з іншими базами даних 248
Служба багатостороннього зберігання 248
Заповнення даними 250
Сховище зв'язків 255
Веб-служба 257
Висновок 260
Завдання для індивідуальної роботи 262
Лекція 18. Порівняння різних NoSQL баз даних 263
Реляційні бази даних 263
Сховища ключів та значень 264
Стовпцеві бази даних 265
Документні бази даних 265
Графові бази даних 266
Висновок 267
Список використаної літератури 270

Вступ

Останні десятиліття характеризуються появою різного роду мобільних додатків, просторових інформаційних систем та інтернету речей, що призвело до вибухового зростання обсягів даних. Крім обсягів даних змінилися їх характеристики, виникла потреба в зберіганні та опрацюванні неструктурованої або слабоструктурованої інформації, яка є проблемною для реляційних баз даних. Новітнім напрямом розвитку баз даних є бази даних NoSQL, які призначені для вирішення зазначеної проблеми. NoSQL це спільна назва всіх СУБД, що можуть опрацьовувати великі обсяги розподілених неструктурованих чи слабоструктурованої даних. Назва NoSQL не заперечує реляційні СУБД , вона трактується як «Not Only SQL» тобто «не лише SQL».

Базами даних NoSQL не підтримується реляційна модель і не використовується мова запитів SQL. Характерним для баз даних NoSQL є відсутність регламентованої структури та жорстко заданої схеми, але при цьому підтримка гнучких динамічних схем з можливістю опрацювання агрегатів і ненормалізованих даних. Бази даних NoSQL підтримують горизонтальне масштабування на кластерах та роботу з Big Data. Ці бази даних на сьогодення є найпоширенішим і найоптимальнішим засобом зберігання неструктурованих даних, які широко застосовуються в інтернет і мобільних додатках.

Метою цього курсу є формування фундаментальних теоретичних знань і практичних навичок з організації нереляційних баз даних та оволодіння сучасною технологією роботи у середовищі сучасних систем керування базами даних NoSQL. Проблема масштабованості SQL була визнана інтернет-компаніями з величезними зростаючими потребами в даних і інфраструктурі, такими як Google, Amazon і Facebook. Вони й застосували свої власні рішення проблеми - такі технології, як BigTable, DynamoDB і Cassandra.

Успіх цих пропрієтарних систем поклав початок розробці ряду аналогічних систем баз даних з відкритим вихідним кодом і пропрієтарних систем, найбільш популярними з яких є Hypertable, Cassandra, MongoDB, DynamoDB, HBase і Redis.

Тому бази даних NoSQL добре підходять для багатьох сучасних додатків, наприклад, мобільних, ігрових, інтернет-додатків, коли потрібні гнучкі масштабовані бази даних з високою продуктивністю і широкими функціональними можливостями, здатні забезпечувати максимальну зручність використання:

Гнучкість. Як правило, бази даних NoSQL пропонують гнучкі схеми, що дозволяє здійснювати розробку швидше та забезпечує можливість поетапної реалізації. Завдяки використанню гнучких моделей даних БД NoSQL добре підходять для частково структурованих та неструктурованих даних.

Масштабованість. Бази даних NoSQL розраховані масштабування з використанням розподілених кластерів апаратного забезпечення, а чи не шляхом додавання дорогих надійних серверів. Деякі постачальники хмарних послуг проводять ці операції у фоновому режимі, забезпечуючи повністю керований сервіс.

Висока продуктивність. Бази даних NoSQL оптимізовані для конкретних моделей даних та шаблонів доступу, що дозволяє досягти вищої продуктивності порівняно з реляційними базами даних.

Широкі функціональні здатності. Бази даних NoSQL надають API та типи даних із широкою функціональністю, які спеціально розроблені для відповідних моделей даних.

Лекція 1. Основи концепції баз даних NoSQL

Перелік знань і вмінь

Вивчення даної теми дасть змогу опанувати основні терміни та визначення, які пов’язані з базами даних NoSQL (Not Only SQL):

— класифікацію баз даних;

— проблеми практичного використання реляційних баз даних при роботі з WEB-додатками та великими даними (BigData);

— поняття великих даних (BigData) і проблеми їх представлення в реляційних базах даних;

— передумови та причини виникнення концепції нереляційних баз даних;

— порівняльні характеристики реляційних і нереляційних баз даних;

— призначення масштабування даних і його практичне застосування;

— особливості вертикального та горизонтального масштабування даних і їх порівняльні характеристики;

— теоретичні засади баз даних NoSQL (теорема CAP);

— переваги та недоліки баз даних NoSQL;

— практичне застосування баз даних NoSQL.

Бази даних типу "ключ-значення" *Бази даних колоночного типу*

Документо-орієнтовані бази даних *Графо-орієнтовані бази даних*

Сильні та слабкі сторони реляційних СУБД

Сильних сторін у реляційної моделі багато – роки досліджень та промислової експлуатації практично у всіх галузях, де застосовуються комп'ютери, гнучкі засоби запитів, високий рівень несуперечності та довговічності даних. Для реляційних СУБД написано та перевірено у справі драйвери більшості мов програмування. Багато програмних моделей, у тому числі об'єктно-реляційне відображення (ORM), припускають, що в основі лежить реляційна СУБД. Головний плюс – гнучкість з'єднання. Не потрібно заздалегідь планувати, які запити будуть пред'являтися до моделі, оскільки завжди можна скористатися з'єднанням, фільтрацією, представленнями і індексами - більш ніж ймовірно, що ви зумієте витягти дані, що вас цікавлять. До того ж дані високою мірою однорідні і узгоджені зі структурованою схемою.

Водночас реляційна база даних – відмінний вибір, і зумовлено це насамперед гнучкістю запитів. Хоча реляційна модель вимагає проєктувати заздалегідь схему, вона не робить ніяких припущень про те, як дані будуть використовуватися. Якщо схема достатньо нормалізована, зокрема, в ній не зберігаються дублікати або обчислювані значення, то можете вважати себе у

всеозброєнні для пред'явлення будь-яких запитів. А якщо включити розроблені модулі, оптимально налаштувати ядро і побудувати потрібні індекси, то система працюватиме напрочуд швидко на терабайтних обсягах даних, споживаючи при цьому зовсім небагато ресурсів. І нарешті, для тих, хто на перше місце ставить безпеку, реляційна СУБД пропонує транзакції – повністю атомарні, несуперечливі, ізольовані та довговічні.

Але хоча реляційні бази даних, без сумніву, досягли найбільшого успіху в порівнянні з іншими видами СУБД, в деяких випадках вони – не найкраще рішення. Сегментування – не найсильніший бік реляційних баз. Якщо потрібно масштабування по горизонталі, а не по вертикалі (тобто кілька паралельних сховищ даних, а не одна суперпотужна машина чи кластер), краще пошукати в іншому місці. Якщо вимоги до даних настільки гнучкі, що не вкладаються в прокрустово ложе вимог до схеми, або вас відлякують витрати на експлуатацію повномасштабної СУБД, або завдання має на увазі дуже велику кількість операцій читання та запису значень ключа, або потрібно зберігати тільки великі двійкові об'єкти, то, бути може, краще підійде якесь інше сховище.

Термінологічний словник

Агрегат **—** це сукупність атрибутів, які інтерпретуються СУБД як єдине ціле. У таблицях реляційної бази дані зберігаються у вигляді атомарних атрибутів. Реляційні бази даних не підтримують агрегати, тому їх називають безагрегатними (aggregateignore). Підтримку агрегатних даних забезпечують бази даних NoSQL. Переважна більшість баз даних NoSQL (за виключенням графових) є агрегатно-орієнтованими. Агрегати зручні для роботи у кластерах, так як агрегат представляє собою одиницю реплікації і фрагментації.

Атомарні атрибути — це атрибути, які зберігають неподільне значення, яке не може бути списком чи певною множиною значень. Тобто це такі дані, розподіл яких на кілька елементів призводить до порушення їх

семантики.

Атомарність транзакції — трактується як «все або нічого», тобто успішно виконана транзакція вносить зміни до бази даних, у протилежному випадку змін не відбувається.

Багатоваріантна персистентність (Polyglot Persistence) — використання кількох СУБД для вирішення різних за технологією задач. Багатоваріантна персистентність може бути реалізована на різних рівнях, наприклад, по підприємству в цілому, або в рамках окремого IT-проєкту.

База даних (БД) — це поіменована, структурована сукупність даних, що характеризує окрему предметну область і перебуває під управлінням системи керування базами даних (СУБД).

База даних NoSQL — спільна назва для цілого класу баз даних відмінних від традиційних реляційних моделей, що підтримують неструктуровані та слабоструктуровані агрегатні дані без схем, не підтримують трансакції і не мають засобів мови SQL. Термін NoSQL («not only SQL» або «не лише SQL») був уперше запропонований у 2009 році і застосовується до баз даних, у яких вирішуються проблема горизонтального масштабування та роботи з Big Data.

База даних NewSQL — клас сучасних реляційних баз даних, які об’єднують у собі переваги NoSQL, підтримують трансакції та мову SQL. Термін був запропонований у 2011 році Метом Аслетом. Необхідність у базах даних NewSQL пояснюється потребою розподіленого, швидкого і надійного оброблення великих даних, у той час як бази NoSQL, не маючи механізму підтримки транзакцій, не відповідають вимогам надійності збереження та забезпечення цілісності й узгодженості даних.

Вертикальне масштабування (Vertical scaling) — спосіб збільшення обсягу БД шляхом нарощення кількості комп’ютерів і серверів БД, тобто за рахунок зростання кількості процесорів, дискової та оперативної пам’яті тощо. Цей тип масштабування, з одного боку, є найпростішим і не вимагає кардинальних змін на прикладному рівні (в існуючій системі усі слабкі

компоненти можна замінити на потужніші). З іншого боку, він є досить дорогим і має певну суто технічну межу, яка обмежена існуючою на конкретний момент часу максимально можливою потужністю сервера або фінансовими ресурсами.

Горизонтальне масштабування (Horizontal scaling) — спосіб збільшення обсягу БД шляхом розбиття її на окремі складові та переміщення їх на окремі сервери, що паралельно виконують одну і ту ж функцію, тобто має місце автоматичне розподілення даних між кількома серверами та підтримка кількох датацентрів з можливість підключення нових комп'ютерів, які об'єднуються в кластери. Цей тип масштабування можу потребувати змін на прикладному рівні.

BASE-вимоги — вимоги, що підтримуються базами даних NoSQL, зокрема: 1) базова доступність (*Basic Availability*) — вимога передбачає, що збій на деяких вузлах приводить до відмови в обслуговуванні лише незначної кількості звернень при збереженні доступності в переважній більшості випадків, тобто гарантується успішне чи безуспішне завершення кожного запиту; 2) гнучкий стан (*Soft state*) — вимога передбачає, що стан системи може змінюватись для досягнення узгодженості не залежно від того, були введені нові дані чи ні; 3) цілісність/узгодженість у підсумку (*eventual consistency*) — вимога передбачає, що в деяких випадках існує можливість суперечливості даних, але завжди досягається їх узгодженість (наприклад, пізніше під час читання).

Великі дані (Big Data) — набір структурованих і неструктурованих даних великих обсягів і значної різноманітності, що ефективно обробляються горизонтально масштабованими програмними інструментами. Основними характеристиками великих даних є три «V», а саме: *volume* — фізичний об'єм, *velocity* — швидкість зростання обсягів даних і швидкість їх оброблення, *variety* — різноманітність і можливість одночасного оброблення різних типів даних (структурованих і неструктурованих).

Ізольованість транзакції — властивість, яка характеризує локальне

виконання кожної окремої транзакції у БД. Ізольовані транзакції виконуються паралельно та не впливають одна на одну.

Комп'ютерний кластер або просто ***кластер* —** це кілька незалежних обчислювальних машин, що використовуються спільно і працюють як одна система для вирішення тих чи тих задач, наприклад, для підвищення продуктивності, забезпечення надійності, спрощення адміністрування тощо. Обчислювальний кластер потрібен для збільшення швидкості обрахунків за допомогою паралельних обчислень.

***Консистентність даних (data consistency чи data validity)* —** узгодженість і цілісність даних, їх внутрішня несуперечливість.

Кластер серверів (Server Cluster) — певна кількість серверів, об’єднаних у групу, які утворюють єдиний уніфікований обчислювальний ресурс. Кластери можуть використовувати просте апаратне забезпечення за рахунок цього вартість горизонтального масштабування є набагато меншою за вертикальне масштабування. Крім того, кластери є надійнішими, у разі виходу з ладу одного з комп’ютерів кластер продовжує функціонувати.

***Масштабування (scalability)* —** спосіб подолати збільшення навантаження на БД. Як правило, масштабування потрібне, коли не вистачає потужностей наявного сервера (або серверів), тоді дані розділяються на певні групи для розміщення їх на окремих серверах. Види масштабування — вертикальне та горизонтальне. Стратегії масштабування — партиціювання, шардинг, реплікації.

Модель даних — формальна теорія представлення та оброблення даних, яка визначає методи: опису типів даних і логічних структур, маніпулювання даними, опису та підтримки цілісності бази даних.

Надійність транзакції — властивість, яка означає, що зміни внесені до бази даних після успішного завершення транзакції гарантовано зберігаються, навіть у випадку збоїв обладнання чи програмного забезпечення.

***Партиціювання (partitioning)* —** це функціональний розподіл даних

на логічно завершені частини згідно певних критеріїв (функцій). Такий розподіл поділяє всі операції з оброблення на кілька незалежних і паралельних процесів, що суттєво прискорює роботу СУБД. Наприклад, на сайтах новин усі дані розподіляються за датою їх публікації, так як до свіжих новин більше звернень ніж до архівних.

Прозорі кластери (Transparent cluster) — системи NewSQL для масштабування поверх традиційних СУБД , які забезпечують проміжний шар (middleware), для автоматичного розподілу баз даних на кілька вузлів. При цьому база даних зберігається у своєму початковому форматі та забезпечується можливість прозорого її підключення до кластера, щоб забезпечити масштабованість.

Персистентні структури даних (persistent data structure) - це структури даних, які при внесенні до них змін зберігають усі свої попередні стани та доступ до них

Реляційна база даних (Relational Data Base Management System — RDBMS) — це модель бази даних, в основу якої покладено математичне поняття відношення, фізичним представленням якого є поіменована двовимірна плоска таблиця, що складається з рядків (кортежів) і стовпчиків (атрибутів). Реляційні бази даних — це чітко структуровані дані, що характеризують певну предметну область і представляються у вигляді логічно пов’язаних поіменованих двовимірних плоских таблиць, що знаходяться під управлінням системи керування базами даних (СУБД).

Реплікація — синхроне чи асинхроне копіювання даних між кількома серверами. Ведучий сервер називають майстром (master), а підпорядковані сервери — слейвами (slave). Майстри використовуються для зміни даних, а слейви — для читання. У класичній схемі реплікацій, як правило, використовується один майстер і кілька слейвів, так як у переважній більшості вебпроєктів привалюють операції читання над операціями запису. Проте є системи, що підтримують кілька майстерів. Репліка може включати всю базу даних (повна репліка), чи певну її підмножину.

Теорема CAP (Consistency, Availability, Partition tolerance), відома як **теорема Брюєра** — евристичне твердження про те, що у будь-якій розподіленій системі (це стосується і розподіленої БД) можливо забезпечити не більше двох з трьох таких властивостей:

• ***узгодженість*** (*consistency*) — в усіх вузлах мережі в один і той же момент часу дані не протирічать один одному;

• ***доступність*** (*availability*) — будь-який запит до розподіленої БД завершується коректною відповіддю;

• ***толерантність до розподілу*** (*partition tolerance*) — розщеплення розподіленої системи на кілька ізольованих секцій не приводить до некорректного відгуку кожної секції.

Транзакції — основа забезпечення несуперечливості даних у реляційних БД. Транзакція гарантує виконання всіх логічно пов'язаних команд, що входять до однієї трансакції. Якщо хоч одна команда завершується помилкою, то всі команди відкочуються назад, так, ніби вони і не виконувалися. Трансакції характеризуються такими властивостями ACID: атомарність (Atomicy), узгодженість (Consistency), ізольованість (Isolation), надійність (Durability).

Узгодженість транзакції — властивість, яка означає, що транзакція буде успішно завершена лише у випадку, якщо зміни не порушують цілісності бази даних.

Шардінг (sharding) — це розподіл даних між різними серверами. Шардінг є подальшим розвитком партиціювання і дозволяє виконати розподіл даних на групи, що, в ідеалі, повинно гарантувати, що дані, до яких виконується одночасне звернення, розміщуються на одному сервері. Процес шардингу виконує розподіл даних між окремими шардами з використанням ключа шардингу. Ключ шардингу — це параметр, згідно якого виконується розподіл даних між окремими серверами, наприклад ідентифікатор клієнта чи організації. На одному вузлі розміщується один шард. При цьому слід враховувати максимальний взаємозв'язок даних в одному шарді.

Теорема CAP

Розуміти особливості різних жанрів баз даних важливо для прийняття рішення про вибір, але жанр – не єдиний критерій. Ми неодноразово будемо згадувати теорема CAP, яка розкриває непривабливу правду про те, як розподілені системи поводяться в умовах нестабільної мережі.

CAP стверджує, що можна створити розподілену систему, яка буде узгоджена (consistent) (тобто операції запису атомарні, і всі наступні операції читання бачать нове значення), доступною (available) (база даних повертає значення, поки працює хоча б один сервер) і стійкою до втрати зв'язності (partition tolerant) (система продовжує функціонувати, навіть якщо зв'язку між серверами тимчасово відсутня), але одночасно можна гарантувати лише дві з цих трьох властивостей.

Іншими словами, можна створити розподілену систему, яка буде узгодженою та стійкою до втрати зв'язності, систему, яка буде доступною та стійкою до втрати зв'язності, або систему, яка буде узгодженою та доступною (але при цьому не буде стійкою до втрати зв'язності, тобто, по суті, не буде розподіленою). Але неможливо створити розподілену базу даних, яка була б одночасно узгодженою, доступною та стійкою до втрати зв'язності.

Теорема CAP істотна при проєктуванні розподіленої бази даних, тому що ви повинні вирішити, чим готові пожертвувати. Яку б базу даних ви не обрали, вона не зможе гарантувати доступність або узгодженість. Стійкість до втрати зв'язності – суто архітектурне рішення (від нього залежить, чи система взагалі буде розподіленою). Важливо розуміти сенс теореми CAP, щоб реалістично оцінювати можливості. Компроміси, прийняті у різних описаних у книзі базах даних, спираються саме на цю теорему.

Розподілена база даних має бути стійкою до втрати зв'язності, тоді як вибір між доступністю і узгодженістю іноді зробити складно. Однак, хоча теорема CAP і стверджує, що, обравши доступність, ви втрачаєте справжню узгодженість, узгодженість зрештою забезпечити все-таки можна.

Ідея узгодженості зрештою полягає в тому, що кожен вузол завжди доступний для відповіді на запити. А компроміс у тому, що зміни даних поширюються інші вузли у фоновому режимі. Це означає, що в будь-який момент часу система може виявитися неузгодженою, але, за великим рахунком, дані в ній точні.

Найвідомішим прикладом системи, узгодженої зрештою, є служба доменних імен в Інтернеті (DNS). Після реєстрації домену може пройти кілька днів, перш ніж зміна дійде до кожного сервера в Інтернеті. Однак будь-якої миті будь-який DNS-сервер доступний (за умови, звичайно, що ви можете до нього підключитися).

Деякі стійкі до втрати зв'язності бази даних можна налаштувати так, що вони будуть більш менш узгоджені або доступні на рівні окремого запиту. Так працює база Riak, яка дозволяє клієнту на момент запиту вказати, який рівень узгодженості йому необхідний. Інші бази даних займають той чи інший куточок трикутника компромісів CAP.

Redis, PostgreSQL та Neo4J узгоджені та доступні (CA); але дані у них не розподіляються, тому про стійкість до втрати зв'язності можна не думати (для нерозподілених систем теорема CAP не має особливого сенсу, хоча на цю тему можна посперечатися). MongoDB і HBase, взагалі кажучи, узгоджені та стійкі до втрати зв'язності (CP). У разі втрати зв'язності мережі вони можуть перестати відповідати на деякі запити (наприклад, у наборі реплік Mongo для операцій читання slaveok прапор скидається в false). На практиці апаратний збій обробляється так, що функціональність знижується поступово, - інші вузли, що ще знаходяться в мережі, можуть підмінити сервер, що вийшов з ладу, - але, строго кажучи, в сенсі теореми CAP вони недоступні. Нарешті, CouchDB доступна та стійка до втрати зв'язності (AP). Хоча два і більше серверів CouchDB можуть реплікувати дані між собою, CouchDB не може гарантувати узгодженість даних на будь-яких двох серверах.

Варто відзначити, що більшість цих баз даних можна налаштувати,

змінивши тип CAP (Mongo може бути зроблети CA, CouchDB – CP), але зараз ми говорили про поведінку, яка передбачається за умовчанням.

Але з урахуванням теореми CAP проєктування розподіленої системи баз даних не вичерпується. Наприклад, для багатьох архітекторів головним питанням є низька затримка (тобто висока швидкодія). Наприклад у системі Amazon Dynamo приділено увагу не лише забезпечення доступності, а й вимогам Amazon до затримок. Для певного класу додатків навіть невелика зміна затримки може призвести до великих фінансових втрат. Знаменита база даних Yahoo PNUTS жертвує і доступністю при експлуатації у звичайному режимі, і узгодженістю при втраті зв'язності задля того, щоб звести затримки до мінімуму. При розгляді розподілених баз даних брати до уваги теорему CAP важливо, але не менш важливо пам'ятати про те, що на ній теорія розподілених баз не закінчується.

Висновок

Бази даних NoSQL спеціально створені для певних моделей даних і мають гнучкі схеми, що дозволяє розробляти сучасні програми. Бази даних NoSQL набули широкого поширення у зв'язку з простотою розробки, функціональністю та продуктивністю за будь-яких масштабів. Вони використовують різноманітні моделі даних для доступу до даних та управління ними. Бази даних таких типів оптимізовані для додатків, які працюють з великим обсягом даних, потребують низької затримки та гнучких моделей даних. Усе це досягається шляхом пом'якшення жорстких вимог до несуперечності даних, притаманних інших типів БД.

У реляційній базі даних запис про певний об'єкт, париклад книгу часто поділяється на кілька частин («нормалізується») і зберігається в окремих таблицях, відносини між якими визначаються обмеженнями первинних та зовнішніх ключів. У цьому прикладі таблиці «Книги» є стовпці «ISBN», «Назва книги» та «Номер видання», у таблиці «Автори» – стовпці «ІД автора» та «Ім'я автора», а таблиці «Автор–ISBN» – стовпці «Автор» та «ISBN». Реляційна модель створена таким чином, щоб забезпечити

цілісність даних посилань між таблицями в базі даних. Дані нормалізовані для зниження надмірності та в цілому оптимізовані для зберігання.

У базі даних NoSQL запис про книгу зазвичай зберігається як документ JSON. Для кожної книги або елемента значення «ISBN», «Назва книги», «Номер видання», «Ім'я автора та «ІД автора» зберігаються як атрибути в єдиному документі. У такій моделі дані оптимізовані для інтуїтивно зрозумілої розробки та горизонтальної масштабованості.

Завдання для індивідуальної роботи

1. Побудувати таблицю порівняльних характеристик основних ознак реляційних і баз даних NoSQL.
2. Вивчіть матеріали ресурсу https://db-engines.com/en/ranking, проаналізуйте та дослідить тенденції розвитку баз даних NoSQL за останні 3 роки.
3. Охарактеризуйте причини створення та розвитку концепції баз даних NoSQL.
4. Дайте визначення агрегата та які моделі бази даних можуть їх підтримувати.
5. Які переваги мають бази даних NoSQL при роботі у веб-середовищі?
6. Сформулюйте теорему CAP та її застосування в розподілених системах.
7. Охарактеризуйте основні підходи до розподілу даних.
8. Що представляє собою масштабування та які є його види?
9. Що представляє собою реплікація та її види в базах даних NoSQL?
10. Охарактеризуйте BASE-вимоги до баз даних NoSQL.
11. Охарактеризуйте клас баз даних NewSQL.
12. Охарактеризуйте ACID-властивості транзакцій. Чому бази даних NoSQL не мають механізму підтримки транзакцій?
13. Охарактеризуйте основні сфери застосування баз даних NoSQL.

Лекція 2. Основи Riak

Riak– це розподілене сховище ключів та значень, у котором значенням може бути будь-що– простий текст, документ JSON або XML, зображення або відеокліп. Для доступу до сховища надається простий та одноманітний HTTP-інтерфейс. З якими б даними ви не працювали, Riak зможе зберегти їх.

Riak також може похвалитися відмовостійкістю. Будь-який сервер може бути зупинений або запущений у будь-який момент, точка загальної відмови не існує. Кластер продовжує працювати при видаленні, додаванні або аварійній відмові серверів. Riak дозволить відмовитись від нічних чергувань, тому що відмова одного вузла – не критична ситуація.

Однак подібна гнучкість потребує компромісів. У Riak немає хорошої підтримки довільних запитів, а сховища ключів та значень, за своєю природою, погано зв'язуються один з одним (іншими словами, поняття зовнішнього ключа відсутнє).

Riak пристосований до мови веб краще, ніж всі інші, що розглядаються у цьому курсі бази даних (на другому місці, ненабагато відставши, стоїть CouchDB). Ви надаєте запит за допомогою URL, заголовків та дієслів HTTP, а Riak повертає ресурси та стандартні коди відповіді HTTP.

Оскільки завдання цього курсу – познайомитися з різноманітними технологіями організації баз даних та покладеними в їх основу ідеями, а не навчити програмування конкретною мовою, ми по можливості будемо намагатися не використовувати нові мови. Riak надає REST-інтерфейс поверх HTTP, тому ми звертатимемося до нього за допомогою утиліти cURL. У виробничому середовищі ви, швидше за все, користуватиметеся драйвером своєї улюбленої мови. Використання cURL дозволить нам вивчити базовий API, не вдаючись до конкретного драйвера або мови програмування.

Riak – чудовий вибір для центрів обробки даних – таких, як Amazon, –

які мають обслуговувати багато запитів із низькою затримкою. У разі, коли кожна зайва мілісекунда очікування означає втрату потенційного клієнта, скласти конкуренцію Riak важко. Вона проста в налаштуванні та адмініструванні і може зростати разом із вимогами. Якщо вам доводилося працювати з веб-службами Amazon, наприклад SimpleDB або S3, то ви легко помітите деяку подібність у формі та функціонуванні. Це не випадковий збіг - основою Riak лягли ідеї, описані в статті Amazon про систему Dynamo.

CRUD, посилання та типи MIME

Можна скачати і встановити пакет Riak, запропонований компанією Basho, яка фінансує розробку, але краще збирати продукт самим, тому що у вихідному дистрибутиві є приклади деяких готових конфігурацій. Якщо вам зовсім не хочеться займатися збиранням, то можете встановити готову версію, а потім завантажити вихідний код і знайти в ньому приклади налаштування сервера. Для запуску Riak знадобиться також мова Erlang (версії не нижче R14B03).

Для складання Riak з вихідного коду необхідні три речі: Erlang (він знадобиться також при вивченні CouchDB), сам вихідний код і стандартні засоби складання Unix типу Make. Процедура налаштування Erlang нескладна, хоча і займає деякий час. Вихідний код ми брали з репозиторію (посилання є на сайті Basho). Усі приклади у цій лекції тестувалися для версії 1.0.2.

У тому ж каталозі, де збирався Riak, виконайте таку команду:

```
$ make devrel
```

Після її завершення ви виявите три налаштовані сервери. Залишилось тільки запустити їх:

```
$ dev/dev1/bin/riak start
$ dev/dev2/bin/riak start
$ dev/dev3/bin/riak start
```

Якщо якийсь сервер не запуститься, тому що порт вже зайнятий, не панікуйте. Просто змініть номер порту, навіщо відкрийте файл etc/app.config у каталозі невдалого сервера і пропишіть новий порт у рядку виду:

```
{http, [ {“127.0.0.1”, 8091 } ]}
```

Теперу нас має бути три процеси Erlang, у яких виконується програма

beam.smp. Кожен процес представляє один вузол Riak (примірник сервера), який нічого не знає про існування інших вузлів. Щоб створити кластер, вузли необхідно об'єднати, запустивши утиліту riak-admin у каталозі кожного сервера та вказавши в команді join адресу будь-якого іншого вузла.

```
$ dev/dev2/bin/riak-admin join dev1@127.0.0.1
```

Який сервер вказати неважливо - в Riak всі вузли рівноправні. Включивши в кластер вузли dev1 та dev2, ми можемо вказати на будь-який з них з dev3.

```
$ dev/dev3/bin/riak-admin join dev2@127.0.0.1
```

Переконаємося, що всі сервери працюють, перевіривши їхній стан у браузері: http://localhost:8091/stats. Ця команда може запропонувати завантажити файл, який містить різноманітну інформацію про кластер. Виглядає вона приблизно так (файл відредагований для зручності читання):

```
{
"vnode_gets":0, "vnode_puts":0,
"vnode_index_reads":0,
...
"connected_nodes":[" dev2@127.0.0.1 ",
" dev3@127.0.0.1 "
],
...
"ring_members":[" dev1@127.0.0.1 ",
" dev2@127.0.0.1 ",
" dev3@127.0.0.1 "
],
"ring_num_partitions":64, "ring_ownership":
"[{' dev3@127.0.0.1 ',21},{' dev2@127.0.0.1 ',21},{' dev1@127.0.0.1 ',22}]",
...
}
```

Переконатися, що всі сервери – рівноправні учасники кільця (ring) можна, продзвонивши порти інших серверів: 8092 (dev2) та 8093 (dev3). А поки що обмежимося статистикою, отриманою від dev1.

Погляньте на властивість ring_members – воно містить імена всіх вузлів і однаково для сервера. Далі, як connected_ nodes повинен бути список інших серверів в кільці.

Тепер зупиніть вузол.

```
$ dev/dev2/bin/riak stop
```

і знову перейдіть на адресу /stats. Як бачите, адреса dev2@127.0.0.1 більше немає у списку connected_nodes. Запустіть dev2, і він самостійно приєднається до кільця Riak (про те, що таке кільце, ми поговоримо на

другий день).

REST і Riak

Акронім REST означає REpresentational State Transfer (передачапредставимих станів). Незважаючи на жаргонне звучання, цей архітектурний стиль де-факто став основою багатьох веб-додатків, тому його назву варто запам'ятати. REST – це рекомендація щодо відображення ресурсів на URL-адреси та взаємодії з ресурсами шляхом використання дієслів, що відповідають операціям CRUD: POST (створення), GET (читання), PUT (оновлення) та DELETE (видалення).

Уставте клієнтську програму cURL для роботи з протоколом HTTP, якщо вона ще не встановлена. Ми будемо використовувати її для надсилання REST-запитів, оскільки вона дозволяє легко задавати дієслова (наприклад, GET або PUT) та HTTP-заголовки (наприклад, ContentType). За допомогою команди curl ми можемо безпосередньо звертатися до REST-інтерфейсу Riak, не користуючись інтерактивною консоллю та драйвером якоїсь мови, наприклад Ruby.

Переконатися,що curl працює, можна, продзвонивши вузол:

```
$ curl http://localhost:8091/ping OK
```

Тепер відправимо свідомо неправильний запит. Прапор -i каже cURL, що хочемо бачити тільки заголовки відповіді.

```
$ curl -I http://localhost:8091/riak/no_bucket/no_key

HTTP/1.1 404 Object Not Found
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic) Дата: Thu, 04 Aug 2011
01:25:49 GMT
Content-Type: text/plain Content-Length: 10
```

Так як робота Riak заснована на URL-адресах і діях, то іскористуються заголовки та коди помилок, визначені у протоколі HTTP. Відповідь з кодом 404 означає, що сторінку не знайдено – нема на що дивитися. А пора щось помістити в сховище Riak, скориставшись дієсловом PUT

Параметр -X PUT говорить cURL, що ми хочемо виконати HTTP дію PUT, щоб зберегти явно задане значення ключа. Прапор -H означає, що

наступний рядок повинен бути переданий у вигляді HTTP-заголовка. У цьому випадку ми встановлюємо MIME тип вмісту HTML. Рядок, наступний після прапора -d (тіло запиту), інтерпретується Riak як нове значення.

```
$ curl -v -X PUT http://localhost:8091/riak/favs/db \
-H "Content-Type: text/html" \
-d "<html><body><h1>My new favorite DB є RIAK</h1></body></html>"
```

Тепер,перейшовши у браузері за адресоюhttp://localhost:8091/riak/favs/db, Ви побачите симпатичне повідомлення від себе.

Збереження нового значення за допомогою дієслова PUT

Riak – це сховище ключів та значень, і, отже, щоб отримати значення, потрібно задати ключ. Riak розбиває множину ключів на сегменти (bucket), щоб уникнути колізій.

Ми хочемо створити систему для обліку тварин у готелі для собак. Спочатку створимо сегмент animals, в якому зберігатимуться відомості про кудлатих постояльців. URL-адреса влаштована таким чином:

```
http://SERVER:PORT/riak/BUCKET/KEY
```

Найпростіше помістити дані у сегмент Riak, якщо ключ заздалегідь відомий. Першим ми додамо пса Тузіка , зіставивши йому ключ `ace` та значення `{"nickname": "The Wonder Dog", "breed" : "German Shepherd"}`. Створювати сегмент явно необов'язково – приміщення першого значення в новий сегмент призводить до його створення.

```
$ curl -v -X PUT http://localhost:8091/riak/animals/ace \
-H "Content-Type: application/json" \
-d '{"nickname": "The Wonder Dog", "breed" : "German Shepherd"}'
```

У відповідь ми отримуємо код 204. Прапор -v (від слова verbose – докладно) призводить до виведення наступного рядка із заголовком:

```
< HTTP/1.1 204 No Content
```

Тепер можна переглянути список створених сегментів.

```
$ curl -X GET http://localhost
{"buckets":["favs","animals"]}
```

За бажання отримати результат операції установки, додавши параметр ?returnbody=true. Перевіримо це, додавши собачку Поллі:

```
$ curl -v -X PUT http://localhost:8091/riak/animals/polly?returnbody=true \
-H "Content-Type: application/json" \
-d '{"nickname": "Sweet Polly Purebred", "breed" : "Purebred"}'
```

На цей раз ми отримаємо відповідь з кодом 200.

```
< HTTP/1.1 200 OK
```

Якщо нам все одно, як називатиметься ключ, то Riak згенерує його самостійно, потрібно тільки відправити POST-запит.

```
$ curl -i -X POST http://localhost:8091/riak/animals \
-H “Content-Type: application/json” \
-d '{“nickname” : “Sergeant Stubby”, “breed” : “Terrier”}'
```

Згенерований ключ буде повернутий у заголовку Location; Зверніть також увагу, що код відповіді в цьому випадку дорівнює 201.

```
HTTP/1.1 201 Created Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)Місцезнаходження:
/riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
Date: Tue, 05 Apr 2011 07:45:33 GMT
Content-Type: application/json Content-Length: 0
```

GET-запит (за замовчуванням cURL, якщо метод запиту не заданий) на вказану в заголовку Location адресу повертає значення ключа.

```
$ curl http://localhost:8091/riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
```

Запит методом DELETE видаляє ключ.

```
$ curl -i -X DELETE http://localhost:8091/riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO

HTTP/1.1 204 No Content Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)Date: Mon, 11 Apr 2011
05:08:39 GMT
Content-Type: application/x-www-form-urlencoded Content-Length: 0
```

У відповідь на запит DELETE тіло не повертається, але у разі успішного видалення код відповіді дорівнюватиме 204, інакше – 404, цілком очікувано.

Якщо ми забули, які ключі є в сегменті, то можемо отримати їх список:

```
$ curl http://localhost:8091/riak/animals?keys=true
```

Можна також отримати їх у вигляді потоку, задавши параметр keys = stream, це зручніше, коли набір даних дуже великий, - сервер просто посилатиме порціями об'єкти, що містять масиви ключів, а в кінці пошле масив.

Посилання

Посиланнями (link) називаються метадані, що асоціюють один ключ з іншими. Базова структура має такий вигляд:

```
Link: </riak/bucket/key>; riaktag="whatever"
```

Ключ, на який вказує це посилання, укладено в кутові дужки (<…>), далі слідує крапка з комою і тег, що описує, як посилання співвідноситься з даним значенням (це може бути довільний рядок).

Переміщення за посиланнями

У нашому готелі для собак є кілька клітин (просторових, комфортабельних та гуманних). За допомогою посилань ми відстежуватимемо, в якій клітині знаходиться кожна тварина. Щоб сказати, що клітина 1 містить (contains) собачку Поллі, ми зв'яжемо клітину з її ключем (принагідно буде створено новий сегмент cages). Клітина (cage) знаходиться в номері 101, це значення ми і вкажемо у вигляді JSON даних.

```
$ curl -X PUT http://localhost:8091/riak/cages/1 \
-H “Content-Type: application/json” \
-H “Link: </riak/animals/polly>; riaktag=\"contains\"" \
-d '{“room”: 101}'
```

Зазначимо, що це посилання односпрямоване. Іншими словами, щойно створена клітина «знає», що в ній живе Поллі, але до запису про саму Поллі ми не внесли жодних змін. Переконатися в цьому можна, запитавши дані про Поллі, – вміст заголовка Link залишився тим самим.

```
$ curl -i http://localhost:8091/riak/animals/polly

HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA= Vary: Accept-Encoding


Server: MochiWeb/1.1 WebMachine/1.9.0 (participate in the frantic)Link: </riak/animals>;
rel=”up”
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT ETag:
“VD0ZAfOTsIHsgG5PM3YZW”
Date: Tue, 13 Dec 2011 17:54:51 GMT
Content-Type: application/json Content-Length: 59

{“nickname”: “Sweet Polly Purebred”, “breed”: “Purebred”}
```

Ми можемо включати будь-яку кількість посилань, розділивши їх комами. Помістимо Тузика в клітину 2 і за допомогою тега next_to вкажемо на клітину 1, щоб знати, що вона знаходиться поряд.

```
$ curl -X PUT http://localhost:8091/riak/cages/2 \
-H “Content-Type: application/json” \
-H “Link:</riak/animals/ace>;riaktag=\”contains\”,
              </riak/cages/1>;riaktag=\"next_to\"" \
-d '{“room”: 101}'
```

Посилання в Riak цікаві тим, що допускають слідування за посиланнями (і більш потужний механізм пов'язаних запитів mapreduce, про який ми поговоримо завтра). Для отримання пов'язаних даних ми додаємо URL специфікацію посилання, влаштовану наступним чином: /_,_,_. Знаки підкреслення в URL є метасимволами для кожного з параметрів посилання: сегмент, тег і ознака "залишати". Що ці терміни означають, ми пояснимо

трохи згодом. А тепер виберемо всі посилання, що ведуть із клітини 1.

```
$ curl http://localhost:8091/riak/cages/1/_,_,_

--4PYi9DW8iJK5aCvQQrrP7mh7jZs
Content-Type: multipart/mixed; boundary=Av1fawIA4WjypRlz5gHJtrRqklD

--Av1fawIA4WjypRlz5gHJtrRqklD
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA= Location:
/riak/animals/polly
Content-Type: application/json Link: </riak/animals>;
rel="up" Etag: VD0ZAfOTsIHsgG5PM3YZW
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname": "Sweet Polly Purebred", "breed": "Purebred"}
--Av1fawIA4WjypRlz5gHJtrRqklD--

--4PYi9DW8iJK5aCvQQrrP7mh7jZs--
```

Сервер повернув відповідь із вмістом типу multipart/mixed, до якого входять заголовки та тіла, що включають усі пов'язані ключі та значення.

Для тих, хто не знає, як читати дані MIME-типу multipart/ mixed, пояснимо, що в заголовку Content-Type зазначений розмежувальний рядок (boundary), який позначає початок та кінець секції заголовків та тіла.

```
--BcOdSWMLuhkisryp0GidDLqeA64 HTTP-заголовки та тіло
--BcOdSWMLuhkisryp0GidDLqeA64--
```

У нашому випадку дані в тілі – те, на що вказує клітина 1: Polly Purebred. Зверніть увагу, що у повернутих заголовках немає інформації про посилання. Це нормально, дані нікуди не поділися і зберігаються під ключем, який веде посилання.

При прямуванні за посиланнями ми можемо замінити підкреслки в специфікації посилання, щоб залишити лише значення, що нас цікавлять. З клітини 2 ведуть два посилання, тому виконання запиту, зазначеного в специфікації посилання, поверне інформацію про тузику, що проживає в цій клітці, і про клітину 1, що знаходиться поруч.залишити тільки інформацію про сегмент animals, замініть перше підкреслення ім'ям сегмента.

```
$ curl http://localhost:8091/riak/cages/2/animals,_,_
```

А щоб дізнатися про клітини поруч (next to) з цією, задайте параметр `tag`.

```
$ curl http://localhost:8091/riak/cages/2/_,next_to,_
```

Останній підкреслення – keep (залишати) – набуває значення 1 або 0. Він корисний при дотриманні за посиланнями другого порядку, тоє провідним з об'єкта, який веде перше посилання. Для цього потрібно лише

додати URL ще одну специфікацію посилання. Давайте спочатку пройдемо за посиланням next_to, що веде з клітини 2. Це дасть клітину 1. Потім перейдемо до тварин, на яких ведуть посилання з клітини 1. Оскільки ми задали keep рівним 0, Riak не повертатиме проміжний результат (дані про клітину 1), а поверне лише відомості про Поллі, яка живе у клітці по сусідству з Тузіком.

```
$ curl http://localhost:8091/riak/cages/2/_,next_to,0/animals,_,_

--6mBdsboQ8kTT6MlUHg0rgvbLhzd


Content-Type: multipart/mixed; boundary=EZYdVz9Ox4xzR4jx1I2ugUFFiZh

--EZYdVz9Ox4xzR4jx1I2ugUFFiZh
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA= Місцезнаходження:
/riak/animals/polly
Content-Type: application/json Link: </riak/animals>;
rel="up" Etag: VD0ZAfOTsIHsgG5PM3YZW
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname": "Sweet Polly Purebred", "breed": "Purebred"}
--EZYdVz9Ox4xzR4jx1I2ugUFFiZh--

--6mBdsboQ8kTT6MlUHg0rgvbLhzd--
```

Якщо нас цікавить інформація як про Поллі, так і про клітину 1, потрібно задати keep рівним 1.

```
$ curl http://localhost:8091/riak/cages/2/_,next_to,1/_,_,_

--PDVOEl7Rh1AP90jGln1mhz7x8r9
Content-Type: multipart/mixed; boundary=YliPQ9LPNEoAnDeAMiRkAjCbmed

--YliPQ9LPNEoAnDeAMiRkAjCbmed
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRKY+VIYo35gRfFgA= Місцезнаходження:
/riak/cages/1
Content-Type: application/json
Link: </riak/animals/polly>; riaktag="contains", </riak/cages>; rel="up" Etag:
6LYhRnMRrGIqsTmpE55PaU
Last-Modified: Tue, 13 Dec 2011 17:54:34 GMT

{"room": 101}
--YliPQ9LPNEoAnDeAMiRkAjCbmed--

--PDVOEl7Rh1AP90jGln1mhz7x8r9
Content-Type: multipart/mixed; boundary=GS9J6KQLsI8zzMxJluDITfwiUKA

--GS9J6KQLsI8zzMxJluDITfwiUKA
X-                                                                                R
iak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA= Місцезнаходження:
/riak/animals/polly
Content-Type: application/json Link: </riak/animals>;
rel="up" Etag: VD0ZAfOTsIHsgG5PM3YZW
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname": "Sweet Polly Purebred", "breed": "Purebred"}
--GS9J6KQLsI8zzMxJluDITfwiUKA--

--PDVOEl7Rh1AP90jGln1mhz7x8r9--
```

У відповідь сервер поверне всі об'єкти на шляху до кінцевого

результату.

Іншими словами, залишить проміжні дані.

Крім посилань, можна зберігати довільні метадані, скориставшись заголовком з префіксом X-Riak-Meta-. Якщо хочемо зберігати колір клітини, але вважаємо цю інформацію необхідної для повсякденних завдань управління клітинами, можемо просто помітити, що клітина 1 рожева (pink). Запитавши заголовки відповіді (прапор -I), ми побачимо, що сервер повернув ім'я та значення метаданих.

```
$ curl -X PUT http://localhost:8091/riak/cages/1 \
-H “Content-Type: application/json” \
-H "X-Riak-Meta-Color: Pink" \
-H “Link: </riak/animals/polly>; riaktag=\"contains\"" \
-d '{“room”: 101}'
```

Типи MIME в Riak

Riak зберігає всі дані у двійковому вигляді, як прийнято в HTTP. Тип MIME повідомляє, як інтерпретувати двійкові дані; досі ми мали справу лише з простим текстом. Типи MIME зберігаються на сервері Riak, але насправді являють собою лише вказівку клієнту - щоб, завантаживши дані, він знав, що з ними робити.

Ми хочемо, щоб у готелі зберігалися фотографії гостей. Щоб завантажити на сервер зображення і задати для нього MIME-тип image/jpeg потрібно лише вказати прапор data-binary в команді curl. І ще ми додамо посилання на ключ /animals/polly, щоб знати, на кого ми дивимося.

Для початку створіть файл polly_image.jpg із зображенням та помістіть його в той самий каталог, з якого запускається команда curl.

```
$ curl -X PUT http://localhost:8091/riak/photos/polly.jpg \
-H “Content-type: image/jpeg” \
-H “Link: </riak/animals/polly>; riaktag=”photo”
--data-binary @polly_image.jpg
```

Тепер перейдіть за цією URL-адресою в браузері і переконайтеся, що файл повертається і відображається саме так, як ми очікуємо:

```
http://localhost:8091/riak/photos/polly.jpg
```

Оскільки ми вказали, що зображення посилається на `/animals/Polly`, то можемо пройти від зображення до Поллі, але не у зворотному напрямку. На відміну від реляційних баз даних посилання явно вказує напрямок переходу.

Висновок

Ми сподіваємося, що ви оцінили потенціал Riak як гнучкий механізм зберігання даних. Але поки що ми розглянули лише стандартні способи роботи з ключами і значеннями, присмачивши страву дещицею посилань. Проєктування схеми Riak знаходиться десь між системами кешування та PostgreSQL. Дані розбиваються кілька логічних категорій (сегментів), а значення можуть посилатися друг на друга. Але нормалізувати схему так само суворо, як і реляційних базах даних, немає сенсу, оскільки Riak не вміє виконувати сполуки для відтворення даних у вихідної формі.

Завдання для індивідуальної роботи

1. Поставте закладку на документацію за проєктом Riak та знайдіть документацію по REST API.

2. Знайдіть максимально повний список типів MIME, які підтримуються браузерами.

3. Вивчіть приклад конфігураційного файлу Riak dev/dev1/etc/app.config та порівняйте його з іншими конфігураційнимифайлами в каталозidev.

4. Використовуючи дієслово PUT, змініть запис animals/polly додавши посилання, що вказує на зображення photos/polly.jpg.

5. Методом POST відправте файл з типом MIME, що ще не розглядався (наприклад, application/pdf), знайдіть згенерований ключ і перейдіть за відповідною URL-адресою в браузері.

6. Створіть новий сегмент medicines (ліки), методом PUT завантажте зображення у форматі JPEG (вказавши правильний тип MIME), задавши для нього ключ antibiotics, і зв'яжіть посиланням з Тузиком (Ace) (бідне, хворе цуценя).

Лекція 3. Riak. Mapreduce та кластери серверів

Сьогодні ми розглянемо каркас mapreduce і виконаємо складніші запити, ніж дозволяє традиційна парадигма ключів та значень. У цій лекції нам знадобляться додаткові дані. Для цього ми перейдемо на інший тип готелю – для людей, а не для тварин. Простенький скрипт мови Ruby заповнить базу даними про великий готель на 10000 номерів. Ще нам знадобиться менеджер Ruby-пакетів, який називаєтьсяRubyGems. Після Ruby і RubyGems встановіть драйвер Riak. Може також знадобитися драйвер json, тому встановлюйте відразу обидва пакети.

```
$ gem install riak-client json
```

Номери у нашому готелі мають різну місткість – від одномісних до восьмимісних – та різні типи – від односпального до апартаментів. Те й інше скрипт генерує випадковим чином.

riak/hotel.rb

```
# генерувати тисячі номерів випадкового типу та місткості

require 'rubygems'
require 'riak'
STYLES = %w{single double queen king suite}

client = Riak::Client.new(:http_port => 8091)
bucket = client.bucket('rooms')
# Створити будинок на 100 поверхів for floor in
1..100
current_rooms_block = floor * 100
puts “Створюю номери #{current_rooms_block} #{current_rooms_block+ 100}”
# На кожному поверсі буде 100 номерів (немаленький готель!) for room in 1...100
# Унікальний номер кімнати використовується як ключ
ro = Riak::RObject.new(bucket, (current_rooms_block + room))
# Вибрати випадковий тип та місткість номера style =
STYLES[rand(STYLES.length)] capacity = rand(8) + 1
# Зберегти інформацію про номер у форматі JSON ro.content_type =
“application/json”
ro.data = {'style' => style, 'capacity' => capacity} ro.store
end end
```

```
$ ruby hotel.rb
```

Отже, ми заселили готель, до якого збираємося застосувати mapreduce.

Вступ до Mapreduce

Один із найбільших та довговічних вкладів Google в інформатику – популяризація алгоритмічного каркасу mapreduce для паралельного виконання завдань на кількох вузлах. Він став цінним інструментом для виконання запитів в цілому класі сховищ даних, стійких до втрати зв'язності.

При використанні mapreduce завдання розбивається на дві частини. У першому кроці – розподілу – список даних перетворюється на новий список у вигляді функції map(). З другого краю кроці – редукції – отриманий список

перетворюється на одне чи кілька скалярних значень з допомогою функції reduce(). Наслідування цього принципу дозволяє системі розбити завдання на менші підзавдання і паралельно виконувати їх на масивному кластері серверів. Щоб підрахувати, скільки значень у базі Riak містять рядок {country : 'CA'}, ми могли б зіставити кожному відповідному документу рядок {count : 1} і на етапі редукції обчислити суму таких одиничних лічильників.

Якби в нашому наборі даних було 5012 гостей з Канади, то результатом редукції був би документ {count: 5012}.

```
map = function(v) {
var parsedData = JSON.parse(v.values[0].data); if(parsedData.country === 'CA')
return [{count: 1}]; else
return [{count: 0}];
}

reduce = function(mappedVals) { var sums = {count : 0};
for (var i in mappedVals) { sums[count] +=
mappedVals[i][count];
}
return [sums];
}
```

У системі Ruby on Rails дані можна було б отримати в такий спосіб (за допомогою вбудованого в неї ORM-інтерфейсу ActiveRecord):

```
# Конструюємо хеш для зберігання місткості номерів, взявши як
# ключ тип номера capacity_by_style = {} rooms =
Room.all
for room in rooms
total_count = capacity_by_style[room.style] capacity_by_style[room.style] =
total_count.to_i + room.capacity
end
```

Метод Room.all відправляє базі даних SQL-запит такого виду:

```
SELECT * FROM rooms;
```

База даних посилає всі результати серверу додатків, а той робить із ними якісь дії. У цьому випадку ми перебираємо всі номери готелю та підраховуємо сумарну місткість номерів кожного типу (наприклад, усі апартаменти можуть бути розраховані на 448 гостей). Для невеликих наборів даних такий підхід є прийнятним. Але зі зростанням числа номерів система починає працювати повільно, тому що база, як і раніше, передає додатку відомості про всі номери.

Mapreduce робить все навпаки. Замість того, щоб отримувати дані від

бази та обробляти їх на стороні клієнта (тобто сервера додатків), каркас mapreduce відправляє на всі вузли, де працюють сервери бази даних, алгоритм, який вони повинні виконати і повернути результат. Кожному серверному об'єкту "розподіляється" *група даних із загальним ключем, а сервер "редукує" отримані дані в одне значення.*

Функція map поставляє вихідні дані редукторам, а обчислені дані передаються редукторам наступного рівня

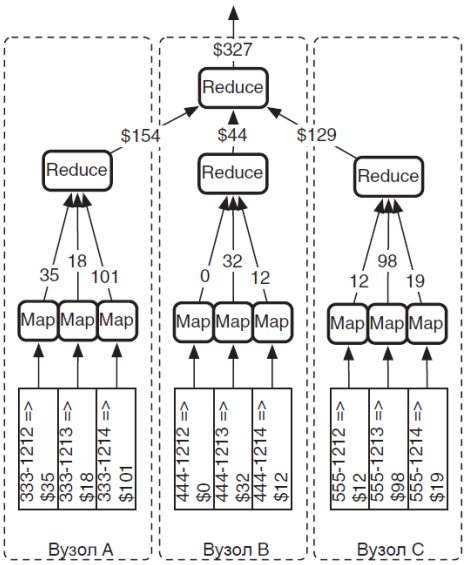


Рис. 7. Результати функції map

У разі Riak це означає, що сервери бази даних відповідають за розподіл та редукцію значень у кожному вузлі. Редуковані значення передаються на наступне коло, де якийсь інший сервер (який зазвичай надіслав запит) редукує їх далі - до тих пір, поки не буде отримано кінцевий результат, який передається клієнту, що запитав (або, наприклад, серверу додатків на платформі Rails).

Така інверсія потоку виконання – дієвий спосіб виконувати складні алгоритми локально на кожному сервері та повернути викликаючому клієнту лише невеликий за розміром результат. Найшвидше відправити алгоритм до даних, а потім дані до алгоритму. На рис. 7 показано, як можна обчислити загальну суму телефонних рахунків на трьох серверах, кожному з яких розподіляються номери із загальним префіксом.

Результати функцій map надходять на вхід функцій reduce першого рівня, а функцій reduce наступного рівня на вхід подаються результати map і функцій reduce попереднього рівня. Ми ще повернемося до цієї ідеї в наступних розділах, тому що це важливий, хоч і непростий, аспект мистецтва написання ефективних mapreduce-запитів.

Функції Mapreduce в Riak

Зараз ми напишемо функції mapreduce для набору даних Riak, які будуть працювати як розглянутий вище підрахунок сумарної місткості номерів. У реалізації mapreduce Riak є зручна особливість - функцію map() можна запускати автономно і дивитися, які виходять проміжні результати (в припущенні, що редукція взагалі потрібна). Не поспішатимемо і подивимося лише на результати для номерів 101, 102 та 103.

Для налаштування функції map необхідно задати мову програмування та вихідний код; ми напишемо функцію JavaScript (код функції – це просто рядок, тому необхідно правильно екранувати спеціальні символи).

Команда @cURL дозволяє не закривати стандартне введення,доки не буде натиснуто клавішу CTRL+D. Введені дані потраплять у тіло HTTP-запиту, який потім відправляється методом POST на адресу /mapred (зверніть увагу саме /mapred, а не /riak/ mapred).

```
$ curl -X POST -H “content-type:application/json” \ http://localhost:8091/mapred --data @-
{
"inputs": [["rooms","101"],["rooms","102"],["rooms","103"]
],
“query”:[
{“map”:{ “language”:”javascript”, “source”:
“function(v) {
/* Дістати дані з об'єкта Riaki розібрати їх як JSON */var parsed_data =
JSON.parse(v.values[0].data);
var data = {};
/* Словник місткості з ключем “тип кімнати” */ data[parsed_data.style] =
parsed_data.capacity; return [data];
}”
}}
]
}

CTRL-D
```

За адресою /mapred сервер очікує отримати коректний JSON документ,саме в ньому ми визначаємо вид команд mapreduce. Ми задали три

номери, що цікавлять нас, привласнивши елементу inputs масив, що містить пари [сегмент, ключ]. Але найголовніше – це елемент query, який повинен містити масив JSON-об'єктів, індексованих ключами map, reduce та link (про останні ми поговоримо докладніше нижче).

Наша функція map отримує дані (`v.values[0].`), розбирає рядок як JSON-об'єкт (`JSON.parse(...)`)та повертає асоціативний масив, у якому ключем є тип номера (`parsed_data.style`), а значенням - місткість (`parsed_data.capacity`). Результат буде отримано у вигляді:

```
[{"suite":6},{"single":1},{"double":1}]
```

Це лише дані, витягнуті з трьох JSON-об'єктів, що описують номери 101, 102 і 103. Але нам потрібно не просто вивести дані у форматі JSON. Значення ключа можна було б перетворити на будь-що. Ми проаналізували лише тіло запиту, але могли б отримати також метадані, інформацію про посилання, ключ або дані. Можливо все – аби значення кожного ключа ставилося у відповідність якесь інше значення.

Якщо хочете, можете розподілити всі 10000 номерів, замінивши у параметрі inputs масиви `[сегмент, ключ]`ім'ям сегмента `rooms`:

```
"inputs":"rooms"
```

Застереження: у відповідь буде повернено багато даних. Варто згадати, що починаючи з версії Riak 1.0 функції mapreduce обробляються підсистемою Riak Pipe. У попередніх версіях використовується застаріла підсистема mapred_system. На кінцевому користувачеві це не відбивається, але швидкодія та стабільність різко підвищилися.

Функції, що зберігаються

Riak надає можливість зберігати функцію map як значення сегмента. Це ще один приклад переміщення алгоритму до бази даних. По суті, це процедура, що зберігається, точніше, визначена користувачем функція - ідея, близька до тієї, що використовується в реляційних базах даних вже багато років.

```
$ curl -X PUT -H "content-type:application/json" \
http://localhost:8091/riak/my_functions/map_capacity --data @function(v) {
```

```
var parsed_data = JSON.parse(v.values[0].data); var data = {};
data[parsed_data.style] = parsed_data.capacity; return [data];
}
```

Надійно зберігши свою функцію, ми можемо виконати її. Вказавши сегмент та ключ, де вона знаходиться.

```
$ curl -X POST -H "content-type:application/json" \ http://localhost:8091/mapred --data @-
{
"inputs": [["rooms","101"],["rooms","102"],["rooms","103"]
],
"query":[
{"map":{ "language":"javascript",
"bucket":"my_functions", "key":"map_capacity"
}}
]
}
```

Результати повинні бути такими ж, як при передачі JavaScript-функції прямо у запиті.

Вбудовані функції

Якщо виконати наведений нижче код, сервер витягне з об'єктів, що описують номери, значення і поверне їх у форматі JSON. Саме це робить функцію Riak. mapValuesJson.

```
curl -X POST http://localhost:8091/mapred \
-H "content-type:application/json" --data @-
{
"inputs":[


["rooms","101"],["rooms","102"],["rooms","103"]
],
"query":[
{"map":{ "language":"javascript",
"name":"Riak.mapValuesJson"
}}
]
}
```

Riak містить інші функції, всі вони знаходяться у файлі `mapred_builtins.js`. За допомогою такого ж синтаксису можна викликати і ваші власні вбудовані функції.

Редукція

Розподіл корисний і сам собою, але дозволяє лише перетворювати одні значення на інші. Для аналізу набору даних навіть такого простого, як підрахунок кількості записів, потрібен ще один крок. Ось і вступає в гру редукція.

У розглянутому прикладі на SQL/Ruby ми бачили, як можна перебрати

всі значення і підрахувати сумарну місткість номерів кожного типу. Тепер зробимо те саме у функції reduce написана на JavaScript.

Здебільшого команди, що надсилаються на адресу /mapred, ті самі. Тільки тепер ми додамо ще функцію reduce.

```
$ curl -X POST -H "content-type:application/json" \ http://localhost:8091/mapred --data @-
{
"inputs":"rooms", "query":[
{"map":{ "language":"javascript", "bucket":"my_functions",
"key":"map_capacity"
}},
{"reduce":{ "language":"javascript", "source":
"function(v) { var totals = {};
for (var i in v) { for(var style in v[i]) {
if (totals [style]) totals [style] + = v [i] [style]; else                                totals[st
}
}
return [totals];
}"
}}
]
}
```

Виконавши цей запит для всіх номерів ми отримаємо асоціативний масив, в якому ключем є тип номера, а значенням – сумарна місткість номерів такого типу.

```
[{"single":7025,"queen":7123,"double":6855,"king":6733,"suite":7332}]
```

На вашій машині будуть інші результати, тому що дані про номери генерувалися випадково.

Фільтри ключів

Порівняно недавно Riak було додано поняття фільтра ключів. Це набір команд, які застосовуються для обробки кожного ключа перед подачею його на вход mapreduce. Фільтри дозволяють заощадити завантаження непотрібних даних. У наступному прикладі ми перетворимо ключ – номер кімнати – у ціле число і перевіряємо, що воно менше 1000 (тобто номер знаходиться на одному з нижніх десяти поверхів; всі номери, розташовані вище, ігноруються).

Функцію `reduce` простіше написати, якщо слідувати тому ж патерну, що при написанні функції`map`. Інакше кажучи, якщо одиночне значення відображається так:

```
[{name:'Eric', count:1}]
```

то результат редукції має виглядати так:

```
[{name:'Eric', count:105}, {name:'Jim', count:215}, …]
```

Зрозуміло, це не обов'язково – але доцільно. Оскільки вихід одних редукторів може подаватися на вхід інших, ви не можете знати, звідки надійшли вхідні значення для конкретної функції`reduce`: від map, від reduce або з обох джерел. Але якщо слідувати описаному вище патерну, то це й не важливо – жодної різниці немає! В іншому випадку функції `reduce` довелося б щоразу перевіряти тип отриманих даних та приймати ті чи інші рішення.

У прикладі, що повертає сумарну місткість номерів, замініть "inputs": "rooms" наступним блоком (він повинен закінчуватися комою):

```
“inputs”:{ “bucket”:”rooms”,
“key_filters”:[[“string_to_int”], [“less_than”, 1000]]
},
```

Слід звернути увагу на дві речі: по-перше, запит виконується набагато швидше (оскільки ми обробляли лише ті значення, які потрібні), а, по-друге, сумарні величини вийшли менше (оскільки враховано лише десять нижніх поверхів).

Каркас mapreduce – потужний засіб агрегування даних та виконання їх всебічного аналізу. Ми часто повертатимемося до нього в цій книзі, але базова ідея у всіх випадках одна і та ж. Riak, втім, додає до mapreduce додаткову рису – посилання.

Слідування за посиланнями за допомогою mapreduce

Подивимося, як механізм прямування за посиланнями робиться за допомогою mapreduce.

У секції query поряд із параметрами `map` і `reduce` з'являється ще один - `link`.

Повернемося до сегменту cages із вчорашнього прикладу з готелем для собак і напишемо розподільник, який повертає тільки клітину 2 (нагадаємо, що в ній мешкає Тузік).

```
$ curl -X POST -H “content-type:application/json” \ http://localhost:8091/mapred --data @-
{
“inputs”:{ “bucket”:”cages”,
“key_filters”:[[“eq”, “2”]]
},
```

```
"query":[
{"link":{
"bucket": "animals", "keep": false
}},
{"map":{ "language":"javascript", "source":
"function(v) { return [v]; }"
}}
]
}
```

Хоча mapreduce-запит відноситься до сегменту cages, повертається також інформація про собаку Тузіку, тому що на неї веде посилання з клітини 2.

```
[{
"bucket":"animals",
"key":"ace","vclock":"a85hYGBgzmDKBVIsrDJPfTKYEhnzWBn6LfiP80GFWVZay0KF5yGE2ZqTGPmCLiJLZAEA
", "values":[{
"metadata":{ "Links":[],
"X-Riak-VTag":"4JVlDcEYRIKuyUhw8OUYJS",
"content-type":"application/json",
"X-Riak-Last-Modified":"Tue, 05 Apr 2011 06:54:22 GMT", "X-Riak-Meta":[]},
"data":"{\"nickname\" : \"The Wonder Dog\",
\"breed\" : \"German Shepherd\"}"
}]
}]
```

У масиві values є дані та метадані (які зазвичай повертаються у вигляді заголовків HTTP).

Якщо об'єднати функції map і reduce, слідування за посиланнями та фільтри ключів, можна буде виконувати довільні запити до широкого спектру ключів Riak. Це набагато ефективніше, ніж переглядати всі дані за клієнта. Оскільки такі запити зазвичай виконуються одночасно на кількох серверах, довго чекати не доведеться. Але якщо нетерпиме навіть невелике очікування, то запит може вказати ще один параметр: `timeout`. Таймаут задається в мілісекундах (за умовчанням мається на увазі `"timeout": 60000`, тобто 60 секунд); якщо виконання запиту не встигає завершитися за вказаний час, запит знімається.

Про узгодженість та довговічність

Серверна архітектура Riak усуває точки загальної відмови (всі вузли рівноправні) і дозволяє за бажанням нарощувати або скорочувати кластер. Це важливо при реалізації великомасштабних проєктів, оскільки база даних залишається доступною, навіть коли кілька вузлів виходять з ладу або з якихось причин перестають відповідати.

Але розподілу даних по кількох серверах неминуче супроводжує та сама проблема. Якщо база даних має функціонувати за умов втрати зв'язності мережі (коли деякі повідомлення пропадають), необхідно йти компроміс. Або залишити базу даних доступною для запитів до сервера, або вжити заходів щодо гарантування узгодженості даних. Неможливо створити розподілену базу даних, яка була б повністю узгодженою, доступною та стійкою до втрати зв'язності. Можна забезпечити виконання лише двох умов (стійка до втрати зв'язності та узгоджена, стійка до втрати зв'язності та доступна, узгоджена, доступна, але не є розподіленою). Це твердження відоме під назвою теореми CAP (Consistency – узгодженість, Availability – доступність, Partition tolerance – стійкість до втрати зв'язності). Додаткові відомості див. у додатку 2, але вже зараз скажемо, що це проблема проєктування системи.

Однак у теоремі є одна лазівка. Насправді стверджується, що у будь-який час не можна забезпечити відразу узгодженість, доступність і стійкість до втрати зв'язності. Riak звертає цей факт на користь, дозволяючи вибирати доступність чи узгодженість лише на рівні окремих запитів. Спочатку подивимося, як у Riak організується кластер серверів, а потім навчимося налаштовувати операції читання та записи у кластері.

Кільце Riak

Riak розбиває множину серверів на секції, що позначаються 160-розрядним числом (тобто від 0 до 2^160). Розробники Riak люблять представляти це величезне ціле число у вигляді кола, яке називають кільцем. Коли обчислюється хеш ключа, що відображає його на секцію, кільце визначає, на якому сервері Riak зберігатиметься значення.

При налаштуванні кластера Riak потрібно насамперед вирішити, скільки в ньому буде секцій. Припустимо, що є 64 секції (це значення мається на увазі за умовчанням). Якщо рознести ці 64 секції трьома вузлами (тобто серверам), то Riak виділить кожному вузлу 21 чи 22 секції (64 / 3).

Кожна секція називається віртуальним вузлом або v-вузлом (vnode). На етапі запуску всі сервери Riak обходять кільце, по черзі пред'являючи заявки на секції, доки не будуть вичерпані всі v-вузли. Це показано на рис. 8.

Вузол A управляє v-вузлами 1, 4, 7, 10...63. Ці v-вузли відображаються на секції у 160-розрядному кільці. Опитавши стан трьох серверів розробки (згадайте команду`curl -H "Accept:text/ plain" http://localhost:8091/stats`, яку ми вивчали вчора), ви побачите такий результат:

```
"ring_ownership": \ "[{' dev3@127.0.0.1 ',21},{' dev2@127.0.0.1 ',21},{' dev1@127.0.0.1 ',22}]"
```

Друге число у кожному об'єкті – це кількість v-вузлов, якими володіє вузол. У сумі виходить 64 (21+21+22).

Кожен v-вузол представляє діапазон хешованих ключів. При вставці даних про номер з ключем 101 хеш-значення ключа може потрапити в діапазон v-вузла 2, і тоді об'єкт ключ-значення буде зберігатися у вузлі B. Перевагою такого рішення є те, що коли потрібно дізнатися, на якому сервері знаходиться ключ, Riak просто обчислює його хеш-значення та знаходить відповідний v-вузол. Точніше, Riak перетворює хеш-значення на список потенційних вузлів і вибирає з нього перший елемент.

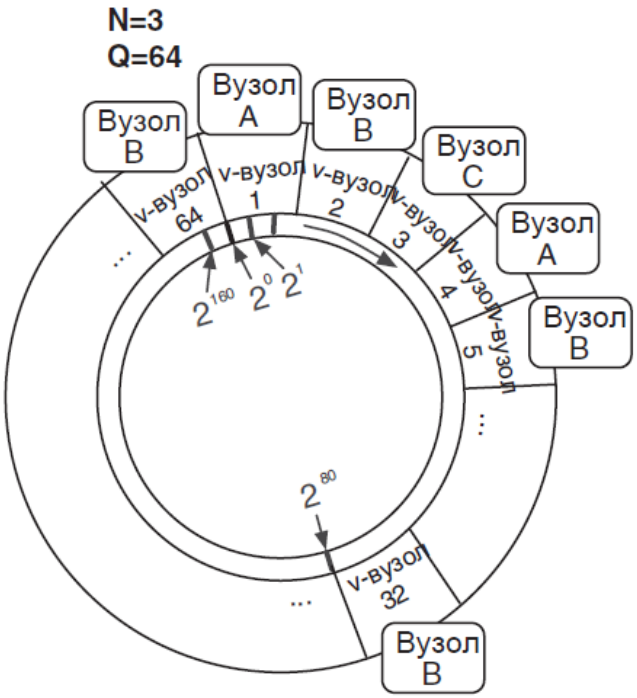


Рис. 8. Кільце Riak з 64 v-вузлами, розподіленими по трьох вузлах.

Вузли та операції читання-запису

Riak дозволяє керувати операціями читання-запису в кластер за допомогою трьох параметрів: N, W і R. N – це кількість вузлів, на які врешті-решт буде реплікована операція запису – інакше кажучи, кількість копій даних у кластері. W – кількість вузлів, на які дані мають бути фактично записані перед надсиланням відповіді про успішне завершення операції. Якщо W менше N, то запис буде вважатися успішним, хоча Riak все ще продовжує реплікувати дані. Нарешті, R – кількість вузлів, необхідне успішного читання значення. Якщо R більше кількості доступних копій, запит на читання завершиться невдало.

Розглянемо ці параметри докладніше. При записі об'єкта до бази даних Riak ми можемо зажадати реплікацію на кілька вузлів. Плюс у тому, що якщо один сервер вийде з ладу, залишиться копія даних на іншому сервері. У властивості сегмента n_val зберігається кількість вузлів, куди слід реплікувати значення (величина N); за умовчанням воно дорівнює 3. Змінити властивості вузла можна, записавши нове значення об'єкт props. Ось як встановити n_val у сегменті animals рівним 4:

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H “Content-Type: application/json” \
-d '{“props”:{“n_val”:4}}'
```

N– це кількість вузлів, які матимуть правильне значення в кінцевому рахунку. Це не означає, що ми повинні дочекатися, поки значення буде репліковано на всі вузли, а потім повернути управління. Іноді потрібно повернути результат клієнту негайно і дозволити Riak продовжити реплікацію у фоновому режимі. А іноді, навпаки, бажано дочекатися завершення реплікації на всі N вузли (заради безпеки).

Величина W визначає, на скільки вузлах запис повинен завершитися успішно, перш ніж можна буде вважати успішною операцію в цілому. В кінцевому рахунку значення буде записано на всі чотири вузли, але якщо зробити W рівним 2, то операція запису поверне керування після успішного запису всього двох копій. Реплікація на два вузли, що залишилися,

відбудеться у фоновому режимі.

```
curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"w":2}}'
```

Нарешті, R – кількість вузлів, звідки має бути прочитане значення, щоб операція читання вважалася успішною. Можна встановити величину R за умовчанням, як ми вище надійшли з n_val і w.

```
curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"r":3}}'
```

Але Riak пропонує і гнучкіше рішення. Ми можемо вказувати, з скільки вузлів слід прочитати значення, задаючи параметр r у конкретному запиті.

```
curl http://localhost:8091/riak/animals/ace?r=3
```

Можливо, вам незрозуміло, навіщо взагалі читати дані із кількох вузлів. Адже записане значення зрештою має бути репліковано на N вузлів, і можна читати з будь-якого з них. Відповідь, мабуть, буде простіше продемонструвати на зображенні.

Припустимо, що трійка NRW задана таким чином `{"n_val":3, "r":2,"w":1}`. У цьому випадку система швидше відкликатиметься на операції запису, оскільки перед поверненням управління необхідно успішно завершити операцію тільки на одному вузлі. Однак не виключено, що інший процес почне операцію читання до того, як всі вузли синхронізуються. Навіть при читанні двох вузлів є шанс отримати старе значення.

Один із способів гарантувати отримання актуального значення – встановити W=N та R=1: {“n_val”:3, “r”:1, “w”:3}. Тобто вчинити так само, як роблять реляційнийні СУБД, які забезпечують несуперечність, не повертаючи управління, поки запис не буде завершено. Це, звичайно, прискорить читання, тому що нам потрібно звернутися лише до одного вузла. Проте запис може сповільнитися – і помітно.

Можна натомість писати однією вузол, а читати з усіх, тобто задати W=1, R=N: {“n_val”:3, “r”:3, “w”:1}. Хоча не виключено, що серед них виявиться кілька застарілих значень, але гарантується, що останні записані

також будуть прочитані. Залишиться тільки з'ясувати, що це за значення (застосовуючи алгоритм векторного годинника, який ми опишемо завтра). Зрозуміло, у своїй виникає прямо протилежна проблема – повільне читання.

Можна також покласти W=2 та R=2: {“n_val”:3, “r”:2, “w”:2}. У такому випадку потрібно завершити запис більш ніж на половині вузлів і читати також більш ніж половину вузлів. При цьому ми забезпечуємо узгодженість, розподіляючи часові затримки порівну між операціями читання та запису. Це називається кворумом і дає мінімальну кількість операцій, необхідну підтримки узгодженості даних.

Ви можете вибрати для R і W будь-яке значення від 1 до N, але в загальному випадку рекомендується задавати один вузол, всі вузли або кворум. Ці налаштування настільки типові, що для R і W передбачені спеціальні рядкові значення, що їх представляють.

one - Таке значення параметру W або R означає, що операція буде вважатися успішною після завершення хоча б на одному вузлі.

all - Значення, що збігається з N. Надання такого значення параметру W або R означає, що операція буде вважатися успішною після того, як завершиться на всіх реплікованих вузлах.

quorum - дорівнює N/2+1. Означає, що операція має успішно завершитись на більшості вузлів.

за замовчуванням - Значення W чи R, задане для сегмента. За замовчуванням 3.

Перераховані вище значення можна задавати як властивості сегмента, так і в параметрах запиту:

```
curl http://localhost:8091/riak/animals/ace?r=all
```

Вимога читати з усіх вузлів таїть у собі небезпеку: якщо якийсь вузол вийде з ладу, Riak, можливо, зможе виконати запит. Проведемо експеримент – зупинимо сервер dev3.

```
$ dev/dev3/bin/riak stop
```

Тепер спроба читати з усіх вузлів цілком може завершитися невдачею

(якщо це не так, спробуйте зупинити сервер dev2 або зупинити dev1 і читати з порту 8092 або 8093; ми не можемо управляти тим, на який v-вузол пише Riak).

```
$ curl -i http://localhost:8091/riak/animals/ace?r=all HTTP/1.1 404 Object Not
Found
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)Date: Thu, 02 Jun 2011
17:18:18 GMT
Content-Type: text/plain Content-Length: 10
not found
```

Якщо запит не вдалося виконати, ви отримуєте помилку 404 (Object Not Found), що загалом має сенс у даному контексті. Об'єкт не знайдено, тому що для виконання запиту недостатньо копій. Зрозуміло, нічого хорошого в цьому немає, тому в такій ситуації Riak виконує відновлення можливості читання (read repair): вимагає N-кратну реплікацію ключа на доступні сервери. Якщо ви спробуєте знову звернутися до того ж URL, то отримаєте значення ключа, а не помилку 404. В онлайнової документації Riak є відмінні приклади мовою Erlang.

Але безпечніше просто запросити кворум (дані з більшості, але не всіх вузлів):

```
curl http://localhost:8091/riak/animals/polly?r=quorum
```

Якщо виконувати запис у режимі кворуму, який можна встановлювати на рівні окремої операції запису, операції читання будуть узгоджені. На тому ж рівні можна задавати і величину W. Якщо ви не хочете чекати, поки Riak запише дані хоча б на один вузол, то можете задати W рівним нулю, що означає "я тобі довіряю, Riak, запишеш потім, а зараз поверни управління".

```
curl -X PUT http://localhost:8091/riak/animals/jean?w=0 \
-H “Content-Type: application/json”
-d '{“nickname” : “Jean”, “breed” : “Border Collie”}' \
```

Але незважаючи на всю цю гнучкість, зазвичай використовуються значення за умовчанням, якщо немає ґрунтовних причин вчинити інакше. Задавати W=0 дуже корисно для запису журнали, а W=N, R=1 – для даних, які записуються рідко, але мають читатися дуже швидко.

Запис та довговічний запис

Для операцій запису Riak не гарантується довговічність, тобто дані не пишуться на диск негайно. Навіть якщо запис на вузол вважається успішним,

все одно не виключено, що в результаті збою дані в цьому вузлі будуть втрачені навіть у випадку, коли W=N. Перед записом на диск дані деякий час зберігаються в буфері в пам'яті, і ось ця частка мілісекунди і є небезпечна зона.

Але у Riak є спеціальний параметр DW (durable write, довговічна запис). У цьому режимі операція проводиться повільніше, але ризик знижується, оскільки Riak не повертає керування, доки об'єкт не буде фізично записаний на диск на заданій кількості вузлів. Як і у випадку звичайного запису, цю властивість можна встановити на рівні сегмента. Нижче ми надаємо dw значення one, щоб дані гарантовано були збережені хоча б на одному вузлі.

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H “Content-Type: application/json” \
-d '{“props”:{“dw”:”one”}}'
```

Якщо хочете, можете також перевизначити це значення в конкретному запиті на запис, поставивши параметр dw URL.

Проте спроба запису на недоступні вузли завершується успішно з кодом відповіді «204 No Content». Пояснюється це тим, що Riak записує дані на розташований поруч вузол, де вони зберігаються доти, доки з'явиться можливість перекинути їх у раніше недоступний вузол. Це чудова короткострокова страховка, тому що якщо один сервер виходить з ладу, то його обов'язки перехоплює якийсь інший вузол Riak. Але, звичайно, якщо всі запити, адресовані серверу A, передаються серверу B, навантаження на сервер B зростає вдвічі. Є небезпека, що в результаті відмовить сервер B, тоді навантаження ляже на сервер C, сервер D і так далі. Таке явище, зване каскадною відмовою, зустрічається рідко, але все ж таки можливе. Вважайте це попередженням – не навантажуйте сервери Riak до краю, оскільки заздалегідь невідомо,

Висновок

Riak – це розподілене сховище ключів і значень, що реплікується, з додатковими функціями, що не має точки загальної відмови.

Якщо раніше ви працювали тільки з реляційними базами даних, то Riak спочатку може здатися дивним. У ньому немає транзакцій, ні SQL, ні схеми. Є ключі, але зв'язування сегментів не схоже з'єднання таблиць. А технологія mapreduce взагалі здається темним лісом.

Однак для вирішення певного класу завдань усі ці компроміси виправдані. Здатність Riak масштабуватися шляхом збільшення кількості серверів (а не нарощуванням потужності одного сервера) та простота використання – чудовий приклад спроби вирішити специфічні проблеми масштабованості в Інтернеті. І замість того, щоб винаходити велосипед, Riak спирається на структуру HTTP, що вже є, надаючи максимальну гнучкість для побудови каркасів або систем з підтримкою веб.

Сильні сторони Riak: Якщо ви збираєтеся спроєктувати великомасштабну систему замовлень на кшталт Amazon або ставите на перше місце високу доступність, то Riak, безумовно, заслуговує на увагу. Один з безперечних плюсів Riak – увага до усунення точок загальної відмови з метою забезпечити безперебійну роботу та нарощування (або скорочення) кластера відповідно до вимог, що змінюються. Якщо структура даних не дуже складна, працювати з Riak буде просто. Але при цьому зберігається можливість витончених маніпуляцій із даними, якщо це необхідно. В даний час підтримується десяток мов (повний перелік можна знайти на сайті Riak), але якщо ви готові писати на Erlang, то можете розширювати ядро в будь-який бік. А якщо потрібна більша швидкодія, ніж здатна забезпечити протокол HTTP.

Слабкі сторони Riak: Якщо потрібні прості засоби формулювання запитів, складні структури даних або жорстка схема або якщо немає потреби в горизонтальному масштабуванні, Riak вам, мабуть, ні до чого. Один із основних недоліків Riak – відсутність простих та надійних засобів формулювання довільних запитів, хоча рух у цьому напрямі, безумовно, є. Технологія mapreduce дає фантастичну функціональність, але ми хотіли б бачити більше вбудованих дій, заснованих на URL або PUT-запитах.

Додавання індексування було великим кроком у правильному напрямку, але хотілося б, щоб ці ідеї отримали подальший розвиток. Нарешті, якщо ви не хочете писати на Erlang, можете зіткнутися з деякими обмеженнями JavaScript – зокрема, неможливості написати функції, що викликаються в точці підключення після фіксації, – і недостатньою швидкодією mapreduce. Однак команда Riak працює над усуненням цих дрібних шорсткостей.

Riak дотепно обходить обмеження, що накладаються теоремою CAP на будь-яку розподілену базу даних.

У Riak використовується висловлена у статті Amazon про систему Dynamo ідея, що умови CAP можна задавати лише на рівні сегментів чи запитів. Це великий крок у бік надійної та гнучкої СУБД з відкритим вихідним кодом. Читаючи про інші бази даних, згадуйте про Riak – вас постійно вражатиме його гнучкість.

Завдання для індивідуальної роботи

1. Прочитайте розділ про mapreduce в онлайновій документації Riak.
2. Знайдіть репозиторій додаткових функцій Riak, де є багато готових mapreduce-функцій.
3. Знайдіть в онлайн документації повний перелік фільтрів ключів, де є багато корисних функцій, наприклад: перетворення рядків у верхній регістр, порівняння числових значень з межами діапазону, логічні операції І/АБО/НЕ і навіть проста реалізація порівняння рядків за допомогою відстані Левенштейна.
4. Напишіть функції map та reduce для сегменту rooms, які обчислюють загальну місткість номерів на кожному поверсі.
5. Доповніть написані у попередній вправі функції фільтром, який обмежує обчислення лише номерами на поверхах 42 та 43.

Лекція 4. Riak. Вирішення конфліктів та розширення Riak

Сьогодні ми говоритимемо про менш очевидні аспекти Riak. Ви вже зрозуміли, що Riak – просте сховище ключів та значень, розподілене між кластером серверів. За наявності кількох вузлів можливі конфлікти даних, іноді їх доводиться вирішувати. Riak надає механізм визначення того, яка операція запису мала місце останньої, – векторний годинник та однорівневий дозвіл (sibling resolution).

Ми також побачимо, як можна контролювати правильність вхідних данихза допомогою точок підключення до та після фіксації. Ми розширимо Riak, перетворивши його на персональну пошукову систему, скориставшись вбудованими в Riak засобами пошуку (з інтерфейсом SOLR) та прискорення запитів за рахунок додаткових індексів.

Вирішення конфліктів за допомогою векторного годинника

Векторний годинник(vector clock) – це алгоритм, який у розподілених системах типу Riak використовується для збереження порядку конфліктуючих оновлень пар ключ-значення. Слідкувати за тим, в якому порядку проводяться оновлення, важливо тому, що різні клієнти можуть підключатися до різних серверів, і, отже, не виключено, що один клієнт оновлює один сервер у той час, як інший клієнт оновлює інший сервер (керувати тим, який сервер виконується запис, ви не можете).

Перше, що спадає на думку, – забезпечити значення тимчасовими мітками і віддавати перевагу тому, у якого мітка найпізніша. Однак у кластері серверів це рішення буде працювати тільки, якщо всі годинники ідеально синхронізовані. Riak такої вимоги не пред'являє, тому що точна синхронізація годинника у кращому випадку важка, а часто просто нездійсненне завдання. Використання централізованого годинника суперечить всій філософії Riak, оскільки така система стала б точкою загальної відмови.

Векторний годинник вирішує проблему, забезпечуючи кожну подію

(створення, оновлення або видалення пари ключ-значення) міткою, що містить інформацію про те, який клієнт зробив зміну і в якому порядку. У результаті самі клієнти або розробник програми можуть вирішувати, хто перемагає у разі конфлікту. Читач, знайомий із системами керування версіями типу Git або Subversion, побачить тут аналогію з тим, як вирішуються конфлікти, коли дві людини змінюють той самий файл.

Векторний годинник в теорії

Припустимо, що справи в готелі для собак йдуть так добре, що ви можете дозволити собі відбирати клієнтуру. Для прийняття оптимальних рішень ви збираєте комісію із трьох експертів із собак, які мають сказати, які кандидати перебувають у хорошому стані. Кожному собаці виставляється оцінка від 1 (у поганому стані) до 4 (ідеальний кандидат). Експерти – назвемо їх Боб, Джейн та Ракшит – повинні дійти одноголосного рішення.

На комп'ютері кожного експерта встановлено клієнта, який підключається до бази даних. Клієнт забезпечує кожен запит міткою, що містить його унікальний ідентифікатор. Ідентифікатор клієнта використовується як для побудови векторного годинника, так і для запам'ятовування в заголовку об'єкта клієнта, що здійснив останнє оновлення. Спочатку розглянемо простий псевдокод, а потім спробуємо продати його в Riak.

Боб створює об'єкт першим і проставляє гідну оцінку 3 новому цуценяті на прізвисько Амбал. У векторний годинник записується його ім'я та номер версії 1.

```
vclock: bob[1] value: {score: 3}
```

Джейн читає цей запис і дає Амбалу оцінку 2. Її оновлення зроблено після того, як це зробив Боб, тому її версія 1 додається до кінця вектора vclock.

```
vclock: bob[1], jane[1] value: {score : 2}
```

Одночасно Ракшит читає запис, створений Бобом, а не Джейн. Йому сподобався Амбал, тому він ставить йому оцінку 4. Як і в попередньому

випадку, ідентифікатор клієнта дописується в кінець вектора годин з номером версії 1.

```
vclock: bob[1], rakshith[1] value: {score: 4}
```

Того ж дня, але пізніше, Джейн (голова комісії) перевіряє оцінки. Оскільки вектор оновлення Ракшита створено не після оновлення Джейн, а одночасно з ним, то має місце конфлікт, який потрібно вирішити. Джейн отримала обидва значення і має право вирішувати, яке вибрати.

```
vclock: bob[1], jane[1]
value: {score : 2}
---
vclock: bob[1], rakshith[1]
value: {score: 4}
```

Вона вибирає середнє, тобто змінює оцінку на 3.

```
vclock: bob[1], rakshith[1], jane[2]
value: {score : 3}
```

Після вирішення конфлікту кожен, хто запитує дані, отримає останнє значення.

Векторний годинник на практиці

Тепер реалізуємо розглянутий щойно приклад у Riak.

Ми хочемо бачити всі версії, що конфліктують, щоб можна було вирішити їх вручну. Встановимо режим збереження кількох версій, надавши true властивості allow_mult у сегменті animals. Декілька значень одного ключа називаються братами (sibling).

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H “Content-Type: application/json” \
-d '{“props”:{“allow_mult”:true}}'
```

У наступному запиті Боб заносить дані про Амбал (Bruiser) в систему з оцінкою 3 та ідентифікатором клієнта bob.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
-H "X-Riak-ClientId: bob" \
-H “Content-Type: application/json” \
-d '{“score” : 3}'
```

Джейн і Ракшіт одночасно читають дані про Амбала, створені Бобом (ви побачите набагато більше заголовків, ми показуємо лише векторний годинник). Зверніть увагу, що Riak кодує векторний годинник Боба, але по суті цей рядок містить ідентифікатор клієнта та номер версії (і ще тимчасову

мітку, тому ваш рядок буде відрізнятися від наведеного нижче).

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true X-Riak-Vclock:
a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==
{“score”: 3}
```

Джейн додає свою оцінку 2 і включає вектор годинника, отриманий з версії Боба. Таким чином, Riak дізнається, що її значення оновлює значення, надане Бобом.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
-H "X-Riak-ClientId: jane" \
-H “X-Riak-Vclock: a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==” \
-H “Content-Type: application/json” \
-d '{“score” : 2}'
```

Оскільки Ракшит читав дані Боба одночасно з Джейн, то він відправляє своє оновлення (з оцінкою 4) разом з тим самим вектором годин, отриманим від Боба.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
-H "X-Riak-ClientId: rakshith" \
-H “X-Riak-Vclock: a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==” \
-H “Content-Type: application/json” \
-d '{“score”: 4}'
```

Перевіряючи оцінки, Джейн не бачить значення, як можна було б очікувати, а HTTP-код, що означає наявність декількох варіантів, і тіло, що містить значення двох братів.

```
$ curl http://localhost:8091/riak/animals/bruiser?return_body=true Siblings:
637aZSiky628lx1YrstzH5 7F85FBAIW8eiD9ubsBAeVk
```

Riak зберігає версії в багаточастинному форматі, тому щоб отримати весь об'єкт, необхідно вказати готовність прийняти цей тип MIME.

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true \
-H “Accept: multipart/mixed”

HTTP/1.1 300 Multiple Choices
X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...OYn9XY4sskQUA
Content-Type: multipart/mixed; boundary=1QwWn1ntX3gZmYQVBG6mAZRVXluContent-Length: 409

--1QwWn1ntX3gZmYQVBG6mAZRVXlu Content-Type:
application/json Etag: 637aZSiky628lx1YrstzH5

{“score”: 4}
--1QwWn1ntX3gZmYQVBG6mAZRVXlu Content-Type:
application/json Etag: 7F85FBAIW8eiD9ubsBAeVk

{“score”: 2}
--1QwWn1ntX3gZmYQVBG6mAZRVXlu--
```

Зверніть увагу, що вище показано значення тегів сутностей (Etag) HTTP (в Riak вони називаються vтегами - vtag). Принагідно зазначимо, що можна задати URL параметр vtag, запросивши тим самим одну конкретну версію. Так, запит curl http://localhost:8091/riak/animals/bruiser?vtag=7F85

FBAIW8eiD9ubsBAeVk поверне {“score”: 2}. Тепер завдання Джейн – скористатися цією інформацією, щоб зробити розумне оновлення. Вона вирішує усереднити обидві оцінки, тобто задати значення 3, використовуючи векторний годинник для вирішення конфлікту.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser?return_body=true \
-H "X-Riak-ClientId: jane" \
-H “X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...OYn9XY4sskQUA” \
-H “Content-Type: application/json” \
-d '{“score” : 3}'
```

Якщо тепер Боб і Ракшіт запросять інформацію про Амбал, то отримають безконфліктну оцінку.

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...CpQmAkonCcHFM4CAA==

{“score”: 3}
```

На всі наступні запити буде повернуто оцінку 3.

Riak вміє перетворювати дані до та після збереження об'єкта; для цього передбачені спеціальні точки підключення коду користувача. У точках підключення, що розглядаються тут, викликаються функції, написані на JavaScript або Erlang і призначені для обробки даних до і після фіксації. Функція, що викликається до фіксації, може модифікувати об'єкт (або навіть примусово завершувати операції з помилкою), а після фіксації – якось відреагувати на успішну фіксацію (наприклад, записати в журнал або відправити повідомлення електронною поштою). У кожного сервера є свій файл app.config, в який можна розмістити посилання на розташування коду користувача на JavaScript. Спочатку відкрийте файл dev/dev1/etc/app.config для сервера dev1 і знайдіть у ньому рядок js_source_dir. Пропишіть у ній шлях до будь-якого каталогу на свій розсуд (але майте на увазі, що цей рядок може бути закоментований, тобто починатися символом %, тому не забудьте видалити його). На нашому комп'ютері цей рядок виглядає так:

```
{js_source_dir, “~/riak/js_source”},
```

Цю зміну доведеться зробити тричі – для кожного сервера розробки. Давайте напишемо функцію-валідатор, яка викликається перед фіксацією, аналізує вхідні дані та перевіряє, що оцінка присутня та знаходиться в діапазоні від 1 до 4. Якщо хоча б одна умова не виконується, то збуджується

виняток і валідатор повертає JSON-об'єкт виду {“fail ”: message}, де message – повідомлення, яке хочемо передати користувачеві. Якщо дані коректні, то функція просто повертає отриманий на вході об'єкт, і Riak його зберігає.

riak/my_validators.js

```
function good_score(object) { try {
/* витягти з об'єкт Riak дані та розібрати їх як JSON-об'єкт */ var data = JSON.parse(
object.values[0].data );
/* якщо властивість score відсутня, сповістити про помилку */ if( !data.score ||
data.score === '' ) {
throw( 'Score is required');
}
/* якщо властивість score поза діапазоном, сповістити про помилку */ if( data.score < 1
|| data.score > 4 ) {
throw( 'Score must be from 1 to 4');
}
} catch (message) {
/* Riak очікує отримати у разі помилки такий JSON-об'єкт */ return { “fail” : message };
}
/* Все нормально, продовжуємо */ return object;
}
```

Збережіть цей файл у каталозі, прописаному у рядку js_source_dir. Оскільки ми змінюємо поведінку сервера, необхідно перезавантажити всі сервери розробки, вказавши в командному рядку аргумент restart.

```
$ dev/dev1/bin/riak restart
$ dev/dev2/bin/riak restart
$ dev/dev3/bin/riak restart
```

Riak знайде всі файли з розширенням .js і завантажить їх у пам'ять. Тепер, щоб JavaScript-функція викликалася з точки підключення перед фіксацією, слід задати у властивості сегмента precommit ім'я функції (не ім'я файлу!).

```
curl -X PUT http://localhost:8091/riak/animals \
-H "content-type:application/json" \
-d '{“props”:{“precommit”:[{“name” : “good_score”}]}}'
```

Протестуємо нашу функцію, поставивши оцінку більше 4. Оскільки функція перевіряє, що оцінка потрапляє в діапазон від 1 до 4, наступний запит завершується помилкою.

```
curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
-H “Content-Type: application/json” -d '{“score” : 5}' HTTP/1.1 403 Forbidden

Content-Type: text/plain Content-Length: 25
Score must be 1 to 4
```

Ми отримуємо код 403 Forbidden, а також текстове повідомлення про помилку, яке було задано у полі "fail". Відправивши GET-запит для отримання значення bruiser, можна переконатися, що оцінка залишилася колишньою – 3. Спробуйте змінити її на 2 – тут на вас чекає успіх.

Точка підключення після фіксації аналогічна. Але тут ми її не розглядатимемо, оскільки функції, викликані з цієї точки, повинні бути написані мовою Erlang. Якщо ви володієте цією мовою, можете прочитати в онлайновій документації, як встановлювати власні модулі. Насправді на Erlang можна писати і функції mapreduce. Але ми продовжимо нашу подорож Riak розглядом інших готових модулів і розширень.

Розширення Riak

У комплекті з Riak поставляється кілька розширень, які за замовчуванням відключені, але додають низку корисних поведінок.

Пошук у Riak

Підсистема пошуку Riak переглядає дані в кластері і будує по них інвертований індекс. Можливо, ви ще не забули про інвертовані індекси, що розглядалися в розділі, присвяченому PostgreSQL. Як і GIN-індекси PostgreSQL, індекси Riak призначені для швидкого пошуку рядків, але в розподіленій системі.

Для пошуку в Riak необхідно активувати відповідне розширення у файлах app.config наступним чином:

```
%% Riak Search Config
{riak_search, [
%% Щоб увімкнути пошук, встановіть цей параметр у 'true'.
{enabled, true}
]},
```

Якщо ви знайомі з такими пошуковими системами, як Lucene, то ця частина здасться вам простіше простого. Якщо ні, тоді цей матеріал легко освоїти.

Ми повинні повідомляти підсистему пошуку про всі зміни бази даних і можемо скористатися для цього точкою підключення до фіксації. Встановити функцію precommit із написаного на Erlang модуля `riak_search_kv_hook` у сегмент `animals` дозволить наступна команда:

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"precommit":
[{"mod": "riak_search_kv_hook","fun":"precommit"}]}}'
```

Звернувшись до URL `curlhttp://localhost:8091/riak/animals`, ми побачимо,

що властивість `precommit` сегменту `animals` справді змінено. Тепер при додаванні до сегменту `animals` будь-яких даних у форматі JSON або XML Riak буде індексувати імена та значення полів. Давайте завантажимо дані про кількох тварин.

```
$ curl -X PUT http://localhost:8091/riak/animals/dragon \
-H "Content-Type: application/json" \
-d '{"nickname" : "Dragon", "breed" : "Briard", "score" : 1 }'
$ curl -X PUT http://localhost:8091/riak/animals/ace \
-H "Content-Type: application/json" \
-d '{"nickname" : "The Wonder Dog", "breed" : "German Shepherd", "score" : 3 }'
$ curl -X PUT http://localhost:8091/riak/animals/rtt \
-H "Content-Type: application/json" \
-d '{"nickname" : "Rin Tin Tin", "breed" : "German Shepherd", "score" : 4 }'
```

Існує кілька способів запитати ці дані, але ми скористаємося вбудованим у Riak інтерфейсом Solr на базі HTTP (він реалізує пошуковий інтерфейс Apache Solr). Для пошуку в сегменті animals ми вказуємо в запиті шлях /solr, за яким слідує ім'я сегмента /animals і команда /select. У параметрах задаються пошукові слова. В даному випадку ми хочемо знайти всі об'єкти, в яких ключ breed (порода) має значення Shepherd (вівчарка).

```
$ curl http://localhost:8091/solr/animals/select?q=breed:Shepherd
<?xml version="1.0" encoding="UTF-8"?>
<response>
              <lst name="responseHeader">
                <int name="status">0</int>
<int name="QTime">1</int>
<lst name="params">
                 <str name="indent">on</str>
<str name="start">0</str>
<str name="q">breed:Shepherd</str>
<str name="q.op">or</str>
<str name="df">value</str>
<str name="wt">standard</str>
<str name="version">1.1</str>
<str name="rows">2</str>
</lst>
</lst>
<result name="response" numFound="2" start="0" maxScore="0.500000">
                 <doc>
<str name="id">ace</str>
<str name="breed">German Shepherd</str>
<str name="nickname">The Wonder Dog</str>
<str name="score">3</str>
                </doc>
                 <doc>
<str name="id">rtt</str>
<str name="breed">German Shepherd</str>
<str name="nickname">Rin Tin Tin</str>
<str name="score">4</str>
                </doc>
               </result>
             </response>
```

Якщо ви бажаєте отримувати відповідь у форматі JSON, додайте параметр wt=json. В одному запиті можна задати кілька пошукових критеріїв,

розділених пробілами (%20 в URL-кодованій формі), додавши ще параметр q.op=and, що означає, що умови об'єднуються зв'язкою I. Щоб знайти вівчарок, кличка яких містить слово rin, потрібно виконати наступний запит:

```
$ curl http://localhost:8091/solr/animals/select\
?wt=json&q=nickname:rin%20breed:shepherd&q.op=and
```

Підсистема пошуку в Riak допускає й інші синтаксичні конструкції, наприклад метасимволи (* відповідає кільком символам, ?-одному), правда, тільки кінці кінцевого пошуку. Запит nickname: Drag * знайде собаку Dragon, а запит nickname: * ragon - не знайде. Ще одна корисна можливість – пошук за діапазоном:

```
nickname:[dog TO drag]
```

Підтримуються також складніші запити, що включають булевські оператори, групування та пошук з урахуванням близькості. Крім того, можна визначати нестандартні кодування даних, створювати нестандартні індекси і навіть вказувати, який індекс використовувати під час пошуку.

Індексування в Riak

Починаючи з версії 1.0 Riak підтримує додаткові індекси. Вони схожі на індекси у PostgreSQL, але мають деякі особливості. Індексується не окремий стовпець або набір стовпців, а метадані, які приєднані до заголовка об'єкта.

Для включення цього розширення також доведеться змінити файлapp.config. Як підсистему зберігання вкажіть eLevelDB замість bitcask, як показано нижче, і перезапустіть сервери:

```
{riak_kv, [
%% Параметр storage_backend визначає модуль Erlang, що описує
%% механізм зберігання у цьому вузлі.
{storage_backend, riak_kv_eleveldb_backend},
```

eLevelDB – це реалізація на Erlang придуманого Google сховища ключів та значень під назвою LevelDB16. Ця підсистема допускає побудову додаткових індексів у Riak.

Підготувавши систему, ми можемо проіндексувати будь-який об'єкт із довільною кількістю спеціальних заголовків, які називаються індексними записами та визначають спосіб індексування. Імена полів починаються

префіксом x-riak-index і закачуються суфіксом _int або _bin, що означає «ціле» та «двійкове» (будь-яке, крім цілого) значення відповідно.

Додаючи бульдога Blue II, зображеного на емблемі спортивної команди Butler Bulldogs Батлерівського університету, ми хочемо проіндексувати його за назвою університету (butler) та за номером

"версії" (Blue 2 - друга емблема із зображенням бульдога).

```
$ curl -X PUT http://localhost:8098/riak/animals/blue
-H "x-riak-index-mascot_bin: butler"
-H “x-riak-index-version_int: 2”
-d '{“nickname” : “Blue II”, “breed” : “English Bulldog”}'
```

Ймовірно, ви помітили, що індекси не мають жодного відношення до значення ключа, що зберігається. Це надзвичайно корисна можливість, оскільки дозволяє будувати індекси, незалежні від даних, що зберігаються. Так можна проіндексувати навіть відео.

Отримати значення за індексом нескладно.

```
$ curl http://localhost:8098/riak/animals/index/mascot_bin/butler
```

Хоча додаткові індекси Riak – великий крок у правильному напрямку, простір для розвитку ще є. Наприклад, щоб проіндексувати дату, необхідно зберегти її у вигляді рядка у форматі, придатному для сортування – YYYYMMDD. Числа з плаваючою точкою потрібно спочатку помножити на відповідний ступінь 10, а потім зберегти як ціле – 1.45 * 100 == 145. Такі перетворення покладаються на клієнта. Тим не менш, поєднання Riak mapreduce, підсистеми пошуку, а тепер і додаткового індексування істотно послаблює обмеження, властиві класичним сховищам ключів і значень, і дає куди більш багату функціональність.

Висновок

Ми завершили знайомство з Riak розглянувши більш складні механізми: вирішення конфліктів за допомогою векторного годинника, а також перевірка та модифікація вхідних даних за допомогою точок підключення при фіксації. Ми також обговорили два корисних розширення Riak: підсистему пошуку та індексування даних для надання додаткової гнучкості запитів. У поєднанні з каркасом mapreduce і посиланнями це дозволяє створювати гнучкі рішення, що далеко перевершують можливості

класичних сховищ ключів і значень.

Завдання для індивідуальної роботи

1. Знайдіть репозиторій додаткових функцій Riak (підказка: він знаходиться на сайті GitHub).
2. Ознайомтесяз додатковою літературою про векторний годинник.
3. Прочитайте, як створювати власну конфігурацію індексів.
4. Створіть свій індекс, що визначає схему animals. Точніше, вкажіть, що поле score – ціле число та сформулюйте запит щодо діапазону його значень.
5. Підготуйте невеликий кластер із трьох серверів (наприклад, ноутбуків або екземплярів EC217) та на кожен встановіть Riak. Організуйте із серверів кластер. Встановіть набір даних Google stock з інформацією про акції (його можна знайти на сайті компанії Basho18).

Лекція 5. HBase

Система Apache HBase створена для обробки величезних обсягів даних (десятки гігабайт). На перший погляд, HBase дуже схожа на реляційну базу даних - настільки, що не знаючи, з чим маєте справу, можна сплутати одне з одним. Найскладніша частина у вивченні HBase в тому, що багато термінів, що застосовуються в HBase, оманливо знайомі. Наприклад, HBase зберігає дані у контейнерах, які називаються таблицями. Таблиці складаються з осередків, що перебувають на перетині рядків і стовпців.

Таблиці HBase поводяться зовсім не так, як відношення, рядки анітрохи не схожі на записи, а склад стовпців може бути змінним (схема його не контролює). Проєктування схеми, як і раніше, залишається важливою справою, оскільки вона впливає на продуктивність системи.

Возночас для користування такою базою даних, крім масштабованості, є кілька причин. По-перше, в HBase вбудований ряд функцій, відсутніх в інших базах даних, наприклад, версіонування, стиснення, складання сміття (для даних з терміном зберігання) і таблиці в пам'яті. Якщо ці можливості є спочатку, значить, вам доведеться писати менше коду, коли вони будуть потрібні. Крім того, HBase дає суворі гарантії несуперечності, що полегшує перехід від реляційних баз даних.

Завдяки цьому HBase може бути наріжним каменем систем оперативної аналітичної обробки. Окремі операції можуть виконуватися повільніше, ніж їхні еквіваленти в інших базах даних, але під час обробки трудомістких запитів HBase часто обганяє інші СУБД. Цим пояснюється, чому HBase так часто застосовується у великих компаніях для реалізації систем аналізу журналів та пошуку.

Введення в HBase

HBase – це стовпцева база даних, яка може похвалитися підтримкою несуперечності та горизонтальною масштабованістю. Вона влаштована за зразком BigTable, високопродуктивної закритої бази даних, яка розроблена

Google і описана у статті 2006 року Bigtable: A Distributed Storage System for Structured Data [http://research.google.com/archive/bigtable.html]. Спочатку HBase створювалася для обробки природних мов та починала своє життя як додатковий пакет до проєкту Apache Hadoop. Але з того часу перетворилася на проєкт Apache верхнього рівня.

З архітектурної точки зору, в HBase спочатку закладено стійкість до відмови. Відмова окремої машини – подія щодо рідкісна, але у великому кластері відмови вузлів є нормою. Використання попереджувального запису журналу та розподіленої конфігурації дозволяє HBase швидко відновлюватися після відмов окремих серверів.

До того ж HBase побудована на базі Hadoop – стабільної та масштабованої обчислювальної платформи, яка надає розподілену файлову систему та засоби mapreduce. Усюди, де є HBase, ви також знайдете Hadoop та інші інфраструктурні компоненти, які можете задіяти і у власних додатках.

Ця база даних активно використовується та розробляється рядом великих компаній для вирішення завдань, пов'язаних з «великими даними». Facebook вибрала HBase як основний компонент своєї інфраструктури обміну повідомленнями, анонсованої в листопаді 2010 року. На сайті Stumbleupon HBase вже кілька років застосовується як сховища даних, що працює в режимі реального часу, та обробки аналітичних даних. Деякі функції сайту беруть дані безпосередньо з HBase. У Twitter HBase також використовується для різних цілей – від генерації даних (для таких додатків, як пошук людей) до зберігання результатів моніторингу та даних про продуктивність. Серед гучних імен, які використовують HBase, можна назвати також eBay, Meetup, Ning, Yahoo! і багато інших.

За такої активності не дивно, що нові версії HBase виходять дуже часто.

CRUD та адміністрування таблиць

Наша сьогоднішня мета – вивчити складові HBase. Спочатку ми запустимо локальний екземпляр HBase в автономному режимі, а потім

покажемо, як в оболонці HBase створювати та змінювати таблиці, вставляти та модифікувати дані. Потім ми побачимо, як деякі з цих операцій програмуються за допомогою HBase Java API мовою JRuby. Принагідно ми розповімо про архітектурні концепції HBase, у тому числі зв'язки між рядками, сімейства стовпців, стовпці та значення.

Повністю працездатний кластер HBase промислової якості повинен включати щонайменше п'ять вузлів – принаймні такою є загальноприйнята думка. Але для наших цілей така конфігурація є надмірною. На щастя, HBase підтримує три режими роботи:

- автономний режим з однією-єдиною машиною;
- псевдорозподілений режим з одним вузлом, що прикидається кластером;
- повністю розподілений режим із кластером спільно працюючих вузлів.

У навчальних цілях ми здебільшого запускатимемо HBase в автономному режимі. Але навіть це не зовсім тривіально, тому ми не зможемо розглянути всі аспекти встановлення та адміністрування, але в деяких місцях відзначатимуть можливі проблеми.

Конфігурація HBase

Насамперед HBase необхідно налаштувати. Конфігураційні параметри зберігаються у файлі hbase-site.xml, що знаходиться в каталозі ${HBASE_HOME}/conf/. Змінна оточення HBASE_HOME повинна вказувати на каталог, до якого встановлено HBase.

Спочатку цей файл містить лише порожній тег `<configuration>`. Але до нього можна додавати визначення властивостей у наступному форматі:

```
<property>
<name>some.property.name</name>
<value>A property value</value>
</property>
```

Повний перелік підтримуваних властивостей разом із значеннями за умовчанням та описами знаходиться у файлі `hbase-default.xml` в каталозі `${HBASE_HOME}/src/main/resources`. За замовчуванням HBase зберігає файли даних у тимчасовому каталозі. Це означає, що всі дані будуть втрачені, коли

операційна система вирішить звільнити місце на диску.

Щоб не втратити дані, слід вказати якесь не таке вразливе місце для зберігання файлів. Запишіть як hbase. rootdir шлях до відповідного каталогу:

```
<property>
<name>hbase.rootdir</name>
<value>file:///path/to/hbase</value>
</property>
```

Щоб запустити HBase, відкрийте термінал (вікно команд) та виконайте наступну команду:

```
${HBASE_HOME}/bin/start-hbase.sh
```

Для зупинки HBase виконайте команду stop-hbase.sh у тому самому каталозі. Якщо щось піде не так, погляньте на нещодавно модифіковані файли в каталозі ${HBASE_HOME}/logs. У системах *nix наступна команда виводить на консоль дані, що записуються в журнали, у міру їх надходження.

```
cd ${HBASE_HOME}
find ./logs -name "hbase-*.log" -exec tail -f {} \;
```

Оболонка HBase

Оболонка HBase – це написана на JRuby командна програма для інтерактивної роботи з HBase. В оболонці можна додавати та видаляти таблиці, змінювати схему таблиці, додавати та видаляти дані та виконувати цілу низку інших операцій. Пізніше ми розглянемо інші способи підключення до HBase, а поки що нашим рідним будинком стане оболонка. Переконавшись, що HBase працює, відкрийте термінал та запустіть оболонку:

```
${HBASE_HOME}/bin/hbase shell
```

Щоб переконатися, що все працює нормально, запитайте номер версії.

```
hbase>version
0.90.3, r1100350, Sat May 7 13:31:12 PDT 2011
```

У будь-який момент можна ввести команду help та отримати список доступних команд або інформацію про використання конкретної команди.

Потім виконайте команду status, яка повертає коротку інформацію про стан сервера HBase.

```
hbase> status
1 servers, 0 dead, 2.0000 average load
```

Якщо в будь-якій команді відбудеться помилка або оболонка зависне,

то, можливо, є проблема з підключенням. HBase робить все можливе, щоб автоматично конфігурувати свої служби відповідно до налаштувань мережі, але іноді помиляється. Якщо ви спостерігаєте подібні симптоми, зверніться до мережевих налаштувань HBase.

Створення таблиці

Словником (map) називається набір пар ключ-значення. Це аналог хешу в Ruby або структури даних HashMap Java. Таблиця в HBase по суті є величезним словником. Або, якщо точним, словник словників.

У таблиці HBase ключі – це довільні текстові рядки, кожен із яких відображається на рядок даних. Рядок даних є словником, у якому ключі називаються стовпцями, а значення інтерпретуються як масиви байтів. Стовпці групуються у сімейства стовпців, отже повне ім'я стовпця і двох частин: ім'я сімейства і кваліфікатор стовпця. Часто вони зчіплюються разом із двокрапкою як роздільник (наприклад, 'family:qualifier').

Усі ці поняття ілюструються гіпотетичною таблицю з двома сімействами стовпців: color і shape. У таблиці є два рядки – позначені пунктирними прямокутниками, що ідентифікуються своїми ключами: first та second. У рядку first ми бачимо три стовпці із сімейства color (з кваліфікаторами red, blue та yellow) та один стовпець із сімейства shape (square). Комбінація ключа рядка та імені стовпця (що включає сімейство та кваліфікатор) дає адресу даних. У цьому прикладі кортеж first/color:red вказує на значення '#F00'.

А тепер скористаємося всім, що дізналися про структуру таблиці, і займемося чимось цікавим – ми збираємося розробити вікі!

З вікі можна асоціювати багато різної інформації, але ми обмежимося абсолютним мінімумом. Вікі-сайт складається зі сторінок, і з кожною сторінкою пов'язані унікальна назва та текст якоїсь статті.

Для створення таблиці ми скористаємося командою create:

```
hbase> create 'wiki', 'text'
0 row(s) in 1.2160 seconds
```

Тут створюється таблиця з ім'ям wiki з єдиним сімейством стовпців. На даний момент таблиця порожня, в ній немає рядків, а отже немає і стовпців. На відміну від реляційних баз даних, в HBase стовпець - невід'ємна приналежність рядка, що містить його. Коли ми почнемо додавати рядки, одночасно будемо додавати і стовпці для зберігання даних.

Ми очікуємо, що в кожному рядку буде рівно один стовпець із сімейства text, кваліфікований порожнім рядком (''). Таким чином, повне ім'я стовпця, що містить текст сторінки, дорівнюватиме 'text:'.

Зрозуміло, таблиця вікі корисна, тільки якщо в ній є якийсь вміст.

Нашому вікі-сайту потрібна початкова сторінка, з неї і почнемо. Для додавання даних таблицю HBase служить команда put:

```
hbase> put 'wiki', 'Home', 'text:', 'Welcome to the wiki!'
```

Вона вставляє до таблиці`wiki`новий рядок із ключем`'Home'`та поміщає текст `'Welcome to the wiki!'` у стовпець`'text:'`.

Запитати дані з рядка 'Home' можна командою get, яка потребує двох параметрів: ім'я таблиці та ключа рядка. Додатково можна задати список стовпців, що повертаються.

```
hbase> get 'wiki', 'Home', 'text:' COLUMN CELL
text: timestamp=1295774833226, value=Welcome to the wiki!
1 row(s) in 0.0590 seconds
```

Зверніть увагу на поле timestamp у повернутих даних. HBase зберігає разом з кожним значенням даних цілочисленну тимчасову мітку – кількість мілісекунд, що пройшли з моменту "епохи" (00:00:00 UTC 1 січня 1970). Коли в комірку записується нове значення, старе не стирається, а залишається разом зі своєю міткою. Це дуже зручна можливість. У більшості баз даних за збереження історичних даних відповідає програміст, а HBase версіонування вже вбудоване!

Приклад: індексна таблиця повідомлень у Facebook

У Facebook база даних HBase є основним компонентом інфраструктури повідомлень, де використовується як для зберігання даних повідомлень, так і для підтримки інвертованого індексу, призначеного для пошуку. Індексна таблиця влаштована в такий спосіб.

- Рядки ключів – це ідентифікатори користувачів.
- Кваліфікатори стовпців – слова, які у повідомленнях користувачів.

Часові мітки виступають у ролі ідентифікаторів повідомлень, що містять слово.

Оскільки повідомлення після введення вже не змінюються, записи про повідомлення в індексі теж статичні. Версіонування в цьому випадку ні до чого. Але Facebook використовує тимчасові мітки як ідентифікатори повідомлень, безкоштовно отримуючи таким чином поле для зберігання даних.

Команди put і get дозволяють задавати часову мітку. Якщо кількість мілісекунд із початку епохи вас чимось не влаштовує, можете вибрати інше ціле значення. Це дає додаткове поле, якщо воно вам потрібне. Якщо тимчасова мітка не задана, HBase за замовчуванням використовує поточний час в момент вставки, а при читанні повертає останню версію.

Зміна таблиць

Поки що у схемі таблиці wiki є лише назви сторінок, їх тексти та інтегрована історія версій – більше нічого. Пропонуємо нові вимоги:

- сторінка унікально ідентифікується своєю назвою;
- сторінка може мати необмежену кількість редакцій;
- кожна редакція ідентифікується своєю тимчасовою міткою;
- редакція містить текст та необов'язковий коментар, що задається при збереженні;
- редакція має автора, що ідентифікується своїм ім'ям.

Ці вимоги представлені на рис. 15. Як бачимо, кожна редакція має автор, коментар при збереженні, текст статті та тимчасову мітку. Назва сторінки не є частиною редакції, вона ідентифікує всі редакції, що стосуються однієї й тієї ж сторінки.

Після перекладення цих вимог мовою таблиць HBase виходить картина, зображена на рис. 16. У нашій таблиці wiki назва виступає в ролі ключа

рядка, а інші дані про сторінку групуються у два сімейства стовпців: text та revision. Сімейство стовпців не змінилося; ми очікуємо, що у кожному рядку є рівно один стовпець, кваліфікований порожнім рядком (''), у якому зберігатиметься вміст статті. Сімейство стовпців revision призначено для решти інформації про редакцію, наприклад, автора та коментаря при збереженні.

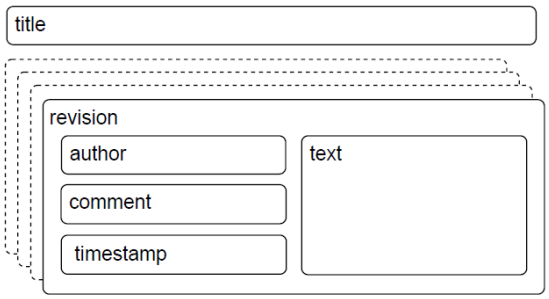


Рис. 15. Вимоги до таблиці wiki (включаючи часову мітку)

Значення за замовчуванням

Оскільки ми не ставили для таблиці wiki жодних спеціальних параметрів, скрізь використовуються прийняті в HBase значення за замовчуванням. Одне з них каже, що слід зберігати лише три версії значень у стовпцях. Збільшимо його. Щоб змінити схему, необхідно спочатку перевести таблицю в автономний режим командою disable.

```
hbase> disable 'wiki'
0 row(s) in 1.0930 seconds
```

Тепер можна модифікувати характеристики сімейства стовпців командою alter.

```
hbase>alter 'wiki', {NAME => 'text', VERSIONS =>
hbase* org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
0 row(s) in 0.0430 seconds
```

Тут ми просимо HBase змінити атрибут VERSIONS сімейства стовпців. Існують й інші атрибути, що допускають встановлення, деякі з них ми розглянемо завтра. Позначення hbase* говорить, що цей рядок є продовженням попереднього.

Операції, що змінюють характеристики сімейства стовпців, можуть коштувати дуже дорого, тому що HBase має створити нове сімейство, а потім скопіювати дані. У виробничій системі це може спричинити тривалий

простий. Тому краще правильно описувати сімейство стовпців із самого початку.

Поки таблиця wiki знаходиться в автономному режимі, додамо сімейство стовпців revision, знову скориставшись командою alter:

```
hbase> alter 'wiki', { NAME => 'revision', VERSIONS => hbase*
org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
0 row(s) in 0.0660 seconds
```

Як і у випадку `text`, ми додаємо в схему тільки сімейство стовпців`revision`, а чи не окремі стовпці. Ми, звичайно, очікуємо, що зрештою кожен рядок міститиме стовпці`revision: author` і `revision:comment`, але задовольнити ці очікування – обов'язок клієнта; у формальній схемі такі вимоги відсутні. Якщо хтось захоче додати стовпець `revision:foo`, HBase також це дозволяє.

Внісши ці зміни, ми можемо знову активувати `wiki`:

```
hbase> enable 'wiki'
0 row(s) in 0.0550 seconds
```

Тепер наша таблиця підтримує розширені вимоги, і можна розпочати заповнення стовпців із сімейства revision даними.

Додавання даних із програми

Як ми бачили, оболонка HBase чудово справляється з маніпулюванням структурою таблиць. На жаль, про підтримку вставки даних такого не скажеш. Команда put дозволяє за один раз задати значення лише одного стовпця, а нам необхідно одночасно додати кілька значень, щоб у всіх була однакова часова мітка. Тож доведеться написати скрипт.

Наведений нижче скрипт можна виконати безпосередньо в оболонці HBase, тому що вона також є інтерпретатором мови JRuby. Скрипт додає нову версію тексту для сторінки Home, одночасно заповнюючи поля автора та коментаря. JRuby виконується віртуальною машиною Java (JVM), надаючи їй доступ до написаного Java коду HBase. Цей та наступні приклади не працюватимуть з версіями Ruby, реалізованими не в JVM.

hbase/put_multiple_columns.rb

```
import 'org.apache.hadoop.hbase.client.HTable' import
'org.apache.hadoop.hbase.client.Put'
```

```
        def jbytes( *args )
        args.map { | arg | arg.to_s.to_java_bytes } end

        table = HTable.new( @hbase.configuration, “wiki” ) p = Put.new( *jbytes(
“Home” ) )

        p.add( *jbytes( “text”, “”, “Hello world” ) )
        p.add( *jbytes( "revision", "author", "jimbo" ) )
        p.add( *jbytes( "revision", "comment", "my first edit") )

        table.put(p)
```

Пропозиції import включають в оболонку посилання корисні класи HBase. Це позбавляє необхідності надалі записувати повні простори імен. Функція jbytes() приймає довільну кількість аргументів та повертає масив байтових масивів Java, очікуваний методами HBase API.

Потім ми створюємо локальну змінну table, що вказує на нашу таблицю wiki, навіщо користуємося адміністративним об'єктом @hbase, отримуючи від нього конфігураційну інформацію.

Далі ми створюємо і налаштовуємо новий екземпляр команди Put, який приймає рядок, що підлягає зміні. У даному випадку нас цікавить сторінка Home. Зрештою, в екземпляр `Put` методом `add()`додаються властивості, після чого ми просимо об'єкт `table` виконати підготовлену операцію `put`. У методу `add()` є кілька варіантів, ми скористалися варіантом із трьома аргументами: `add(column_family, column_qualifier,value)`.

Навіщо потрібні родини стовпців?

Може виникнути бажання побудувати всю інфраструктуру взагалі без родин стовпців; чому б не зберігати всі ці рядки в одному сімействі? Таке рішення реалізувати простіше. Але в нього є й недоліки – і, насамперед, неможливість тонкого налаштування продуктивності. Для кожного сімейства стовпців параметри, що визначають такі характеристики, як швидкість читання та запису та обсяг зайнятого місця на диску, задаються незалежно.

Усі операції в HBase атомарні на рівні рядків. Скільки б стовпців не було в рядку, будь-яка операція бачить узгоджене представлення того рядка, який читає чи модифікує. Таке проєктне рішення дозволяє клієнтам робити осмислені припущення даних.

Наша операція put торкається кількох стовпців і не звертається явно до поля timestamp, тому у значень у всіх стовпцях буде одна і та ж тимчасова мітка (поточний час у мілісекундах). Впевнимося в цьому, виконавши операцію get.

```
hbase> get 'wiki', 'Home' COLUMN CELL
revision: author timestamp=1296462042029, value=jimbo revision: comment
timestamp=1296462042029, value=my first edit text: timestamp=1296462042029, value=Hello world
3 row(s) in 0.0300 seconds
```

Як бачите, значення timestamp у всіх перелічених вище стовпцях дійсно однаково.

Робота з «великими даними». Імпорт даних, виконання скриптів

Одна з типових проблем, з якою стикається кожен, хто досліджує нову СУБД, як перенести в неї дані. Описане задання статичних рядків в операції Put – це, звичайно, добре, але є й кращі способи.

На щастя, копіювання тексту команди в оболонку не єдиний спосіб виконати її. При запуску оболонки HBase з командного рядка можна вказати ім'я JRuby-скрипта -HBase виконає його, якби він був введений прямо в оболонці. Синтаксис виглядає так:

```
${HBASE_HOME}/bin/hbase shell <your_script> [<optional_arguments> ...]
```

Оскільки нас цікавлять саме «великі дані», то напишемо скрипт, який імпортуватиме в нашу таблицю wiki статті з вікіпедії. Організація WikiMedia Foundation, яка займається проєктами Wikipedia, Wictionary та інші, періодично публікує вивантажені набори даних, якими ми цілком можемо скористатися. Ці набори є гігантськими файлами XML. Ось приклад запису з англомовної вікіпедії:

```
<page>
<title>Anarchism</title>
<id>12</id>
<revision>
<id>408067712</id>
<timestamp>2011-01-15T19:28:25Z</timestamp>
<contributor>
<username>RepublicanJacobite</username>
              <id>5223685</id>
</contributor>
<comment>Undid revision 408057615 by [[Special:Contributions...</comment>
<text xml:space=”preserve”>{{Redirect|Anarchist|the fictional character|
...
[[bat-smg:Anarkėzmos]]
</text>
</revision>
</page>
```

Ми завбачливо включили у свою схему всю присутню тут інформацію: назву (ключ рядка), текст, тимчасову мітку та автора. Тому скрипт імпорту редакцій не має бути надто складним.

Потокове завантаження XML

Діятимемо по порядку. Нам необхідно розібрати величезний файл XML, розглядаючи його як потік даних (за допомогою інтерфейсу SAX), з цього і почнемо. Базова структура JRuby-програми для послідовного аналізу XML-файлу виглядає приблизно так:

hbase/basic_xml_parsing.rb

```
import 'javax.xml.stream.XMLStreamConstants'

factory = javax.xml.stream.XMLInputFactory.newInstance reader =
factory.createXMLStreamReader(java.lang.System.in)

while reader.has_next type = reader.next

if type == XMLStreamConstants::START_ELEMENT tag = reader.local_name
# зробити щось із тегом
elsif type == XMLStreamConstants::CHARACTERS text = reader.text
# зробити щось із текстом
elsif type == XMLStreamConstants::END_ELEMENT
# те саме, що для START_ELEMENT end
end
```

Тут слід звернути увагу до кілька моментів. Спочатку ми створюємо екземпляр XMLStreamReader і пов'язуємо його з об'єктом java.lang.System.in, говорячи тим самим, що збираємося читати зі стандартного введення.

Потім у циклі while ми зчитуємо лексеми з потоку XML, доки дійдемо остаточно. У тілі циклу проводиться обробка лексем. Дії залежать від того, чи є поточна лексема тегом, що відкриває, закриває тегом або текстом між тегами.

Завантаження вікіпедії

Тепер можна поєднати логіку обробки XML з вивченими раніше класами HTable та Put. Остаточний скрипт наведено нижче. Здебільшого тут все вже вам знайоме, а нові речі ми обговоримо докладніше.

hbase/import_from_wikipedia.rb

```
require 'time'

import 'org.apache.hadoop.hbase.client.HTable' import
'org.apache.hadoop.hbase.client.Put' import 'javax.xml.stream.XMLStreamConstants'

def jbytes( *args )
args.map { | arg | arg.to_s.to_java_bytes } end
```

```
       factory = javax.xml.stream.XMLInputFactory.newInstance reader =
factory.createXMLStreamReader(java.lang.System.in)

           document = nil buffer = nil count = 0

       table = HTable.new( @hbase.configuration, 'wiki' )
           table.setAutoFlush(false)

       while reader.has_next type = reader.next

         if type == XMLStreamConstants::START_ELEMENT case reader.local_name
       when 'page' then document = {}
       when /title|timestamp|username|comment|text/ then buffer = [] end

         elsif type == XMLStreamConstants::CHARACTERS buffer << reader.text

unless buffer.nil?

         elsif type == XMLStreamConstants::END_ELEMENT

       case reader.local_name
       when /title|timestamp|username|comment|text/document[reader.local_name] = buffer.join
       коли 'revision'
       key = document['title'].to_java_bytes
       ts = (Time.parse document['timestamp']).to_i

       p = Put.new (key, ts)
       p.add( *jbytes( “text”, “”, document['text'] ) )
       p.add( *jbytes( “revision”, “author”, document['username'] ) ) p.add( *jbytes(
“revision”, “comment”, document['comment'] ) ) table.put( p)
       count += 1
       table.flushCommits() if count % 10 == 0 if count % 500 == 0
       puts “#{count} records inserted (#{document['title']})” end
       end end
       end

       table.flushCommits() exit
```

Перша помітна відмінність – поява кількох змінних:

- `document`: поточна стаття та дані про її редакції;
- `buffer`: символьні дані з поточного поля документа (текст, назва, автор тощо);
- `count`: лічильник уже імпортованих статей

Зверніть особливу увагу на виклик `table.setAutoFlush(false)`. HBase періодично ***автоматично скидає*** дані на диск. Для більшості програм цей режим дуже зручний. Якщо ми скасуємо автоскидання, то всі операції, що виконуютьсяputнакопичуються в буфері, поки явно не буде викликаний метод `table.flushCommits()`. Це дозволяє збирати операції запису в пакет та виконувати їх, коли нам зручно.

Далі подивимося, що відбувається на етапі аналізу. Якщо тег, що

відкриває –<page>, то ми ініціалізуємо документ, присвоюючи як значення порожній хеш. Якщо ж це якийсь інший з цікавих для нас тегів, то ми очищаємо буфер для зберігання тексту.

Обробка символьних даних зводиться до додавання до кінця буфера.

Для більшості тегів, що закривають, потрібно просто помістити вміст буфера в document. Але для тега </revision> ми створюємо новий екземпляр Put, заповнюємо його вмістом полів документа та зберігаємо у таблиці. Після цього ми на кожній десятій ітерації викликаємо метод flushCommits(), а кожної п'ятисотою друкуємо на стандартний висновок повідомлення про перебіг обробки.

Стиснення та фільтри Блума

Ми вже майже готові запустити скрипт, треба лише зробити ще одну справу. Сімейство стовпців text міститиме великі блоки тексту, і добре їх стискати. Тому включимо режим стиснення та швидкого пошуку:

```
hbase> alter 'wiki', {NAME=>'text', COMPRESSION=>'GZ', BLOOMFILTER=>'ROW'}
0 row(s) in 0.0510 seconds
```

HBase підтримує два алгоритми стиснення: Gzip (GZ) та Лемпеля-Зіва-Оберхумера (LZO). Спільнота HBase настійно рекомендує віддати перевагу алгоритму LZO, а не Gzip, але тут ми все ж таки використовуємо GZ.

З алгоритмом LZO пов'язана проблема ліцензування. Хоча йоговихідний код відкритий, але надається за ліцензією, не сумісною з принципами Apache, тому LZO не можна включати до дистрибутиву HBase. На сайті наведено докладні інструкції про те, як встановити та налаштувати LZO. Якщо вам потрібне високопродуктивне стиснення, встановіть LZO.

Фільтр Блума - дуже цікава структура даних, яка по суті дозволяє відповісти на запитання: «Чи зустрічався цей об'єкт раніше?». Спочатку Бертон Ховард Блум (Burton Howard Bloom) розробив його в 1970 для перевірки орфографії, але з тих пір фільтри Блума стали застосовуватися в додатках для зберігання даних, щоб швидко визначити, чи існує деякий

ключ. Короткий вступ на цю тему наведено на врізці «Як працюють фільтри Блума».

У HBase фільтри Блума використовуються для того, щоб дізнатися, чи існує рядок із зазначеним ключем деякий стовпець (BLOOMFILTER=>'ROWCOL'), а також для того, щоб з'ясувати, чи існує взагалі рядок з цим ключем (BLOOMFILTER=>'ROW '). Кількість стовпців у сімействі стовпців та кількість рядків практично нічим не обмежені. Фільтр Блуму дозволяє швидко визначити, чи існують дані, ще до виконання дорогої операції читання з диска.

Ось тепер все готове до запуску сценарію. Нагадаємо, що розмір файлів величезний, тому про те, щоб завантажити та розпакувати їх і мови бути не може. А як тоді бути?

На щастя, за допомогою конвеєрів, що є в будь-якій *nix-системі, ми можемо в одній команді завантажити XML-файл, розпакувати його і передати на вхід нашому скрипту:

```
curl <dump_url> | bzcat | \
${HBASE_HOME}/bin/hbase shell import_from_wikipedia.rb
```

Замість <dump_url> слід підставити URL того чи іншого вивантаженого набору даних WikiMedia Foundation3. Візьміть файл [project]-latest-pages-articles.xml.bz2, який містить або англомовну вікіпедію (~6 ГБ)4, або англомовний вікі-словник (~185 МБ)5. Ці файли містять останні редакції сторінок у просторі імен Main, тобто опущені сторінки користувачів, сторінки обговорення тощо.

Підставляйте URL-адресу – і вперед! По екрану бігтимуть рядки такого вигляду:

```
      500 records inserted (Ashmore and Cartier Islands) 1000 records inserted
(Annealing)
      1500 records inserted (Ajanta Caves)
```

Якщо не вдаватися в деталі реалізації, то фільтр Блума є статичним бітовим масивом, в якому всі елементи спочатку рівні 0. Щоразу, як фільтру подається новий блок даних, деякі біти перемикаються в 1. Які саме, залежить від згенерованого хешу даних, який потім перетворюється на

позиції бітів.

Якщо згодом ми захочемо дізнатися, чи фільтру пред'являвся конкретний блок даних, то фільтр обчислить, які біти повинні бути для нього підняті,і перевірить, чи вони рівні 1. Якщо хоча б один із цих бітів дорівнює 0, то фільтр впевнено відповідає «ні». Якщо всі рівні 1, то фільтр відповідає «так, є шанс, що цей блок раніше пред'являвся». Однак чим більше блоків було введено, тим вище ймовірність хибно-позитивної відповіді.

У цьому полягає відмінність фільтра Блума від простого хеша. Хеш ніколи не дає хибно-позитивну відповідь, зате і обсяг пам'яті для його зберігання нічим не обмежений. Розмір фільтра Блума фіксований, але іноді він дає хибно-позитивні відповіді, причому ймовірність залежить від рівня насичення і піддається точної оцінки.

Скрипт буде працювати, доки не зустріне помилку, але, швидше за все, ви захочете перервати його раніше. Для цього натисніть клавішу CTRL+C.

Механізм горизонтального масштабування HBase на диску

У HBase рядки зберігаються відсортованими за ключем. Регіоном називається множина рядків, що визначається початковим ключем (включно) та кінцевим (виключно). Регіони не перекриваються, і кожному призначається регіональний сервер у кластері. У нашій спрощеній конфігурації, де є єдиний автономний сервер, існує лише один сервер регіонів, який відповідає за всі регіони. Але у повністю розподіленому кластері регіонних серверів буде багато.

Давайте подивимося, як сервер HBase використовує місце на диску, це допоможе зрозуміти, як розміщуються дані. Для цього перейдіть до каталогу, вказаного у параметрі hbase.rootdir, та виконайте команду du. Ця стандартна команда *nix каже, скільки місця займає каталог і – рекурсивно – всі його підкаталоги. Прапор -h означає, що du повинна виводити числа у зручному для людини вигляді.

Ось що було показано на нашій машині, коли було вставлено

приблизно 12 000 сторінок і імпорт все ще продовжувався.

```
$ du -h
231M    ./.logs/localhost.localdomain,38556,1300092965081 231M      ./.logs
4.0K    ./.META./1028785192/info
12K     ./.META./1028785192/.oldlogs
28K     ./.META./1028785192
32K     ./.META.
12K     ./-ROOT-/70236052/info
12K     ./-ROOT-/70236052/.oldlogs
36K     ./-ROOT-/70236052
40K     ./-ROOT-
72M     ./wiki/517496fecabb7d16af7573fc37257905/text 1.7M
./wiki/517496fecabb7d16af7573fc37257905/revision 61M
./wiki/517496fecabb7d16af7573fc37257905/.tmp 12K
./wiki/517496fecabb7d16af7573fc37257905/.oldlogs 134M
./wiki/517496fecabb7d16af7573fc37257905
134M    ./wiki 4.0K    ./.oldlogs 365M
.
```

Звідси можна зробити корисні висновки, як HBase використовує місце на диску. Рядки, що починаються з /wiki, відносяться до таблиці`wiki`. Підкаталог з довгим ім'ям `517496fecabb7d16af7 573fc37257905` представляє один регіон (поки що єдиний). Його підкаталоги `/text` і `/Revision` відповідають сімействам стовпців `text` і `revision`. Нарешті, в останньому рядку наведено сумарний результат - всього HBase зайняла на диску 365 МБ.

І ще одне. Верхні два рядки, що починаються з /.logs, показують, скільки місця займають журнали попереджувального запису (WAL). У HBase попереджувальний запис застосовується захисту від відмов вузлів. Це досить типова техніка аварійного відновлення. Так, у файлових системах попереджувальний запис журнал називається журналированием. У HBase інформація WAL заноситься до фіксації результатів операцій редагування (put і increment) на диску.

З огляду на продуктивність результати редагування не пишуться на диск негайно. Система працює набагато швидше, коли введення/виведення буферизується і запис на диск проводиться блоками. Якщо протягом цього часу регіональний сервер, який відповідає за відповідний регіон, вийде з ладу, то HBase зможе використовувати WAL, щоб зрозуміти, які операції були успішно виконані, і вжити коригуючих дій.

Запис WAL необов'язковий, але за замовчуванням увімкнено. У класах Put та Increment є метод встановлення setWriteToWAL(), що дозволяє скасувати запис конкретної операції у WAL. Взагалікажучи, краще

використовувати режим за замовчуванням, але в деяких випадках сенс його змінити. Наприклад, коли виробляється імпорт, який у будь-який момент можна повторити (як у нашому скрипті імпорту з вікіпедії), відключення запису WAL дозволяє пожертвувати можливістю аварійного відновлення заради досягнення вищої продуктивності.

Якщо дозволити скрипту працювати досить довго, можна побачити, як HBase розбиває таблицю на кілька регіонів. Ось як у нашому випадку виглядала видача du після додавання приблизно 150 000 сторінок:

```
$ du -h
40K    ./.logs/localhost.localdomain,55922,1300094776865 44K       ./.logs
24K ./.META./1028785192/info
4.0K    ./.META./1028785192/recovered.edits 4.0K     ./.META./1028785192/.tmp
12K     ./.META./1028785192/.oldlogs
56K     ./.META./1028785192
60K     ./.META.
4.0K    ./.corrupt
12K     ./-ROOT-/70236052/info
4.0K    ./-ROOT-/70236052/recovered.edits 4.0K       ./-ROOT-/70236052/.tmp
12K     ./-ROOT-/70236052/.oldlogs
44K     ./-ROOT-/70236052
48K     ./-ROOT-
138M    ./wiki/0a25ac7e5d0be211b9e890e83e24e458/text 5.8M
./wiki/0a25ac7e5d0be211b9e890e83e24e458/revision 4.0K
./wiki/0a25ac7e5d0be211b9e890e83e24e458/.tmp 144M
./wiki/0a25ac7e5d0be211b9e890e83e24e458
149M    ./wiki/15be59b7dfd6e71af9b828fed280ce8a/text 6.5M
./wiki/15be59b7dfd6e71af9b828fed280ce8a/revision 4.0K
./wiki/15be59b7dfd6e71af9b828fed280ce8a/.tmp 155M
./wiki/15be59b7dfd6e71af9b828fed280ce8a
145M    ./wiki/0ef3903982fd9478e09d8f17b7a5f987/text 6.3M
./wiki/0ef3903982fd9478e09d8f17b7a5f987/revision 4.0K
./wiki/0ef3903982fd9478e09d8f17b7a5f987/.tmp 151M
./wiki/0ef3903982fd9478e09d8f17b7a5f987
135M    ./wiki/a79c0f6896c005711cf6a4448775a33b/text 6.0M
./wiki/a79c0f6896c005711cf6a4448775a33b/revision 4.0K
./wiki/a79c0f6896c005711cf6a4448775a33b/.tmp 141M
./wiki/a79c0f6896c005711cf6a4448775a33b
591M    ./wiki 4.0K    ./.oldlogs 591M       .
```

Головна відмінність полягає в тому, що старий регіон (517496fecabb7d16af7573fc37257905) зник, а замість нього з'явилися чотири нові. В автономному режимі всі ці регіони обслуговуються одним і тим самим сервером, але в розподіленому середовищі за них відповідали б різні регіональні сервери.

У зв'язку з цим виникає низка питань, наприклад: «Звідки регіонні сервери знають, за які регіони відповідають?». і «Як дізнатися, який регіон (і, отже, регіональний сервер) обслуговує цей рядок?».

В оболонці HBase ми можемо опитати таблицю .META., щоб отримати

додаткові відомості про регіони. . META. – це спеціальна таблиця, єдине призначення якої – відстежувати всі таблиці користувача та регіональні сервери, відповідальні за обслуговування регіонів кожної таблиці.

```
hbase> scan '.META.',{ COLUMNS => [ 'info:server', 'info:regioninfo' ] }
```

Навіть за невеликої кількості регіонів ми отримуємо масу інформації. Ось фрагмент того, що було отримано на нашому комп'ютері, – після форматування та забирання зайвого.

```
ROW
wiki,,1300099733696.a79c0f6896c005711cf6a4448775a33b.
COLUMN+CELL
column=info:server, timestamp=1300333136393, value=localhost.localdomain:3555
column=info:regioninfo, timestamp=1300099734090, value=REGION =>{
NAME => 'wiki,,1300099733696.a79c0f6896c005711cf6a4448775a33b.',STARTKEY => '',
ENDKEY => 'Demographics of Macedonia', ENCODED =>
a79c0f6896c005711cf6a4448775a33b, TABLE => {{...}}

ROW
Wiki,Demographics of Macedonia,1300099733696.0a25ac7e5d0be211b9e890e83e24e458.

COLUMN+CELL
                        column=info:server, timestamp=1300333136402,
 value=localhost.localdomain:35552column=info:regioninfo, timestamp=1300099734011, value=REGION
     => { NAME => 'wiki,Demographics of Macedonia,1300099733696.0a25...e458.',
STARTKEY => 'Demographics of Macedonia', ENDKEY => 'June 30',
ENCODED => 0a25ac7e5d0be211b9e890e83e24e458, TABLE => {{...}}
```

Як бачимо, обидва регіони обслуговуються одним і тим самим сервером `localhost.localdomain:35552`. Перший регіон починає з порожнього рядка (`''`) і закінчується рядком `'Demographics ofMacedonia'`. Другий регіон починається з рядка `'Demographics ofMacedonia'` і закінчується рядком `'June30'`.

Ключ `STARTKEY` включається до регіону, а ключ `ENDKEY` виключається. Якби ми шукали рядок `'Demographics ofMacedonia'` , то знайшли б її у другому регіоні

Оскільки рядки таблиці зберігаються у відсортованому вигляді, інформацію з таблиці .META. можна використовувати для того, щоб знайти в якому регіоні (і на якому сервері) знаходиться цей рядок. Таблиця .META. розбита на регіони та обслуговується регіонними серверами, як будь-яка інша таблиця. Щоб знайти, які сервери відповідають за різні частини таблиці `.META.`,необхідно просканувати таблицю`-ROOT-`.

```
hbase> scan '-ROOT-',{ COLUMNS => [ 'info:server', 'info:regioninfo' ] }
ROW
.META.,,1 COLUMN+CELL
                        column=info:server, timestamp=1300333135782,
 value=localhost.localdomain:35552column=info:regioninfo, timestamp=1300092965825, value=REGION
```

```
                                                => {
NAME => '.META.,,1', STARTKEY => '', ENDKEY => '',
ENCODED => 1028785192, TABLE => {{...}}
```

За регіональні сервери відповідає головний вузол, який часто називають HBaseMaster. Головний сервер може одночасно виконувати функції регіонального сервера.

Якщо регіональний сервер виходить з ладу, то головний сервер втручається і передає комусь відповідальність за обслуговування регіонів, призначених вузлу, що відмовив. Новий доглядач загляне у WAL і подивиться, чи потрібні якісь відновлювальні дії. Якщо ж виходить з ладу головний сервер, його роль приймає він одне із регіональних серверів.

Сканування однієї таблиці для побудови іншої

Перервавши скрипт імпорту, ми можемо перейти до наступного завдання: отримання інформації з імпортованого вмісту вікі. Сторінки вікі-сайту рясніють посиланнями – якісь ведуть на інші статті, а якісь – на зовнішні ресурси. Така перехресна структура таїть у собі найбагатші поклади даних про топологію. Давайте їх досліджуємо!

Наша мета – уявити зв'язки між статтями як спрямовані посилання з однієї статті на іншу. Посилання на внутрішню статтю у вікітексті має вигляд [[<target name>|<alt text>]], де <target name> – стаття, на яку вказує посилання, а <alt text> – альтернативний текст, що відображається (необов'язковий).

Наприклад, якщо текст статті про фільм Star Wars (Зоряні війни) містить рядок “[[Yoda|jedi master]]” (Йода|майстер-джедай), то ми повинні зберегти зв'язок двічі: один раз як вихідне посилання зі Star Wars, а інший – як вхідна з Yoda. У такому разі ми зможемо швидко знайти всі посилання, що виходять зі сторінки, так і всі посилання, що ведуть на сторінку.

Схема `TABLE` не показано у прикладі сканування інформації про регіони. Це зроблено для того, щоб зменшити обсяг тексту, а про налаштування продуктивності ми говоритимемо нижче. Щоб побачити визначення схеми таблиці, скористайтесь командою `describe`, наприклад:

```
hbase> describe 'wiki' hbase> describe '.META.'
```

```
hbase> describe '-ROOT-'
```

Для зберігання додаткових даних про посилання ми створимо нову таблицю. Зайдіть в оболонку та введіть таку команду:

```
hbase> create 'links', {
NAME => 'to', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'
}, {
NAME => 'from', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'
}
```

В принципі, можна було б не створювати нову таблицю, а включати читати дані про посилання в одне з вже існуючих сімейств стовпців або додати в таблицю wiki одне або кілька сімейств. Але створення нової таблиці має ту перевагу, що з нею пов'язані окремі регіони. Отже, кластер зможе за необхідності ефективніше розбивати таблиці на регіони.

В загальному випадку спільнота HBase рекомендує зводити кількість сімейств стовпців в одній таблиці до мінімуму. Зробити це можна двома способами: включити в одне сімейство більше стовпців або виносити сімейства окремі таблиці. Що віддати перевагу залежить від того, як часто клієнтам потрібна вся рядок даних (а не кілька стовпців з неї).

У нашому випадку сімейства стовпців text і revision повинні знаходитися в одній таблиці, щоб при додаванні нової редакції метаданих і тексту були однакові тимчасові мітки. Навпаки, вміст таблиці links тимчасові мітки не мають нічого спільного зі статтею, звідки вилучені посилання. Крім того, більшість клієнтів зацікавлені або в тексті статті, або інформації про посилання, але не в тому й іншому одночасно. Тому винесення сімейств шпальт to і from в окрему таблицю виправдано.

Побудова сканера

Створивши таблицю links, ми можемо перейти до написання скрипта, який сканує всі рядки таблиці wiki. З кожного рядка він витягне вікітекст і виділить у ньому посилання. І для кожного знайденого посилання створить вхідний та вихідний запис у таблиці links. Більшість скрипту вам вже знайома. Багато фрагментів використано повторно, а про нові ми розповімо нижче.

hbase/generate_wiki_links.rb

```
       import 'org.apache.hadoop.hbase.client.HTable' import
'org.apache.hadoop.hbase.client.Put' import
'org.apache.hadoop.hbase.client.Scan' import 'org.apache.hadoop.
hbase.util.Bytes'

       def jbytes( *args )
       return args.map { |arg| arg.to_s.to_java_bytes } end

       wiki_table = HTable.new( @hbase.configuration, 'wiki' ) links_table = HTable.new(
@hbase.configuration, 'links' ) links_table.setAutoFlush( false )

           scanner = wiki_table.getScanner( Scan.new )

       linkpattern = /\[\[([^\[\]\|\:\#][^\[\]\|:]*)(?:\|([^\[\]\|]+) )?]/ count = 0

       while (result = scanner.next())
           title = Bytes.toString( result.getRow() )
       text = Bytes.toString( result.getValue( *jbytes( 'text', '' ) ) ) if text
       put_to = nil
           text.scan(linkpattern) do |target, label| unless put_to
       put_to = Put.new( *jbytes( title ) ) put_to.setWriteToWAL( false )
       end

       target.strip! target.capitalize!

       label = '' unless label label.strip!

       put_to.add( *jbytes( "to", target, label ) ) put_from = Put.new( *jbytes(
target ) ) put_from.add( *jbytes( "from", title, label ) ) put_from.setWriteToWAL(
false )
         links_table.put( put_from )
       end
           links_table.put( put_to ) if put_to links_table.flushCommits()
       end


       count += 1
       puts "#{count} pages processed (#{title})" if count % 500 == 0 end

       links_table.flushCommits() exit
```

Спочатку ми створюємо об'єктScan, за допомогою якого скануватимемо таблицю `wiki`.

Для отримання даних із рядків і стовпців необхідно маніпулювати байтами, але нічого страшного тут немає.

Щоразу, як у тексті сторінки виявляється зразок `linkpattern`, ми отримуємо кінцеву статтю та текст посилання, після чого передаємо їх об'єктам `Put`.

Насамкінець ми просимо таблицю виконати відразу всі операції, що накопичилисяPut. Можливо (хоч і малоймовірно), що стаття взагалі не містить посилань, звідси й умова `ifput_to`.

Виклик методу setWriteToWAL(false) – суто суб'єктивне рішення. З одного боку, це лише навчальна вправа, а, з іншого, ми завжди можемо

перезапустити скрипт у разі помилки, тому ми вирішили вибрати швидкість і змиритися з долею, якщо вузол вийде з ладу.

Якщо ви готові, - запускайте скрипт.

```
${HBASE_HOME}/bin/hbase shell generate_wiki_links.rb
```

Скрипт повинен друкувати приблизно таке:

```
500 pages processed (10 petametres) 1000 pages processed
(1259)
1500 pages processed (1471 BC)
2000 pages processed (1683)
…
```

Як і раніше, можете дати скрипту допрацювати до кінця або будь-якої миті перервати його, натиснувши CTRL+C.

За використанням місця на диску можна стежити за допомогою du – ми вже це робили. Ви побачите нові рядки, що стосуються щойно створеної таблиці links, причому зазначений у них розмір у міру роботи скрипта буде монотонно зростати.

Дослідження результатів

Ми тільки-но написали програмний сканер для вирішення складного завдання. Тепер за допомогою команди оболонки scan просто виведемо частину вмісту таблиці на консоль. Для кожного посилання, яке скрипт знаходить у стовпці text: він створює записи to і from в таблиці links. Щоб побачити, які посилання створені, зайдіть в оболонку та проскануйте таблицю.

```
hbase> scan 'links', STARTROW => “Admiral Ackbar”, ENDROW => “It'sa Trap!”
```

Буде виведено дуже багато інформації. Але, зрозуміло, ніхто не заважає скористатися командою get, щоб вивести посилання на одну статтю:

```
hbase> get 'links', 'Star Wars'

COLUMN CELL
...
links:from:Admiral Ackbar timestamp=1300415922636, value= links:from:Adventure
timestamp=1300415927098, value= links:from:Alamogordo, New Mexico timestamp=1304 value=
links:to:20th century fox timestamp=1300419602350, value= links:to:3-d timestamp=1300419602350,
value=
links:to:Aayla secura timestamp=1300419602350, value=
...
```

У таблиці Wiki структура рядків дуже регулярна. Як ви пам'ятаєте, у будь-якому рядку є стовпці text:, revision:author та revision:comment. У

таблиці links такої регулярності вже немає. В одному рядку може бути як один, так і сотні стовпців. А назви стовпців настільки ж різноманітні, як самі ключі рядка (назви статей вікіпедії). HBase тому і називається сховищем розріджених даних.

Щоб дізнатися, скільки рядків зараз є у таблиці, можна скористатися командою count.

```
        hbase> count 'wiki', INTERVAL => 100000, CACHE => 10000
        current count: 100000, row: Alexander wilson (vauxhall) Current count: 200000,
row: Bachelor of liberal studies Current count: 300000, row:
        ...
        current count: 2000000, row: Thomas Hobbes current count: 2100000,
row: Vardousia
        Current count: 2200000, row: Wurrstadt (verbandsgemeinde) 2256081 row(s) in
173.8120 seconds
```

Через розподілену архітектуру HBase не зберігає інформацію про кількість рядків у кожній таблиці. Щоб отримати цю величину, рядки необхідно порахувати (виконавши повне сканування таблиці). На щастя, наявність регіонів дає змогу сканувати таблицю розподілено. Тож навіть у випадку, коли для вирішення завдання потрібно просканувати всю таблицю, це не так страшно, як у інших базах даних.

Висновок

Ми отримали перше представлення про запуск HBase сервера. Ми навчилися конфігурувати його та користуватися журналами для пошуку та усунення несправностей. За допомогою оболонки HBase ми здійснили найпростіші операції адміністрування та маніпуляції з даними. Моделюючи схему вікі-сайту, ми познайомилися з проєктуванням схем у HBase. Ми дізналися, як створювати таблиці та працювати із сімействами стовпців. Проєктування схеми у HBase зводиться до прийняття рішень про параметри сімейств стовпців і, що не менш важливо, про семантичну інтерпретацію тимчасових міток та ключів рядків.

Ми також зробили перший крок у дослідженні HBase Java API, виконавши в оболонці скрипт, написаний на JRuby. Завтра ми зробимо наступний крок, скориставшись оболонкою для запуску скриптів, які виконують такі тривалі завдання, як імпорт даних.

Отже, ми навчилися писати скрипт імпорту, який завантажує дані в таблицю HBase, розбираючи потік даних XML. Потім ми застосували цю техніку для закачування вмісту вікіпедії в таблицю wiki. Ми дізналися багато нового про HBase API, зокрема про деякі доступні клієнту важелі управління продуктивністю: setAutoFlush(), flushCommits(), setWriteToWAL(). Принагідно ми обговорили низку архітектурних особливостей HBase, у тому числі аварійне відновлення за рахунок попереджувального запису в журнал.

І якщо мова зайшла про архітектуру, то ми розглянули регіони таблиць і розподіл відповідальності за їх обслуговування між регіонними серверами. Просканувавши таблиці .META. і -ROOT-, ми дещо довідалися про внутрішній пристрій HBase.

І, нарешті, ми обговорили, як розріджене зберігання, закладене у проєкт HBase, впливає на продуктивність. І при цьому торкнулися деяких рекомендацій спільноти щодо використання стовпців, сімейств стовпців та таблиць.

Завдання для індивідуальної роботи

1. Знайдіть, як виконати в оболонці HBase такі операції:

- видалити з рядка значення, що зберігаються у зазначених стовпцях;
- видалити весь рядок.

2. Напишіть функцію put_many(), яка створює екземпляр Put, додає до нього довільну кількість пар стовпець значення і зберігає в таблиці. Сигнатура функції має виглядати як:

```
    def put_many( table_name, row, column_values )
# тут має бути ваш код
    end
```

3. Визначте свою функцію put_many(), скопіювавши її текст в оболонку HBase, а потім викликавши:

```
    hbase> put_many 'wiki', 'Some title', {hbase* "text:" => "Some article text", hbase*
"revision:author" => "jschmoe", hbase * "revision:comment" => "no comment" }
```

4. Для дослідження імпорту даних, створіть базу даних, що містить відомості про продукти харчування. Скачайте набір даних MyPyramid Raw Food Data із сайту http://explore.data.gov/Health-and-Nutrition/MyPyramid-

Food-Raw-Data/b978-7txq. Розпакуйте архів і знайдіть у ньому файл `Food_Display_Table.xml`. Цей набір складається з тегів `<Food_Display_Row>`. Всередині кожного такого тега є теги `<Food_Code>` (ціле число), `<Display_Name>` (Рядок) та інші відомості про продукт харчування в тегах з відповідними іменами.

5. Створіть нову таблицю foods з одним сімейством стовпців для зберігання інформації про продукти харчування. Як вибрати ключ рядка? Які параметри є сенс задати для цього сімейства стовпців?

6. Напишіть JRuby скрипт для імпорту даних про продукти харчування. Застосуйте такий самий спосіб потокового аналізу за допомогою SAX-аналізатора, як при імпорті даних з вікіпедії, адаптувавши його до інших даних.

7. За допомогою конвеєра подайте дані про продукти харчування на вхід скрипта, щоб заповнити таблицю.

8. За допомогою оболонки HBase запитайте з таблиці foods інформацію про свої улюблені продукти.

Лекція 6. Робота у хмарі з HBase

Досі наш досвід обмежувався лише одним локальним сервером. Насправді ж, вибираючи HBase, ви, напевно, захочете організувати кластер значного розміру, щоб повною мірою задіяти всі переваги розподіленої архітектури.

Тепер у фокусі нашої уваги буде експлуатація віддаленого кластера HBase. Спочатку ми розробимо клієнтський додаток на Ruby та підключимося до локального серверу за двійковим протоколом Thrift. Потім ми піднімемо кластер із кількома вузлами на платформі відомого постачальника хмарних служб –Amazon EC2 – застосувавши технологію управління кластером Apache Whirr.

Ми працювали лише з оболонкою HBase, але HBase підтримує ряд інших протоколів підключення до сервера. Нижче наведено повний список.

Назва	Спосіб підключення
Оболонка	Прямий
Java API	Прямий
Thrift	Двійковий протокол
REST	HTTP
Avro	Двійковий протокол

У таблиці вище вказано, як здійснюється звернення до написаних на Java методів API: безпосередньо, поверх протоколу HTTP або компактного двійкового протоколу. Всі способи підключення, крім Avro, пройшли тестування та готові до промислової експлуатації; Avro – порівняно новий протокол, який поки що вважається експериментальним.

З усіх варіантів Thrift (Thrift англійською означає «ощадливість»), мабуть, найпопулярніший для розробки клієнтських програм. Це зрілий двійковий протокол із мінімальними накладними витратами. Спочатку він був розроблений Facebook і поширювався у вихідних кодах, пізніше отримав статус інкубаційного проєкту Apache.

Встановлення Thrift

Як часто буває з базами даних, для роботи з протоколом Thrift необхідне попереднє налаштування. Щоб підключитися до сервера HBase за протоколом Thrift, необхідно:

1. Налаштувати HBase так, щоб запускалася служба Thrift.
2. Ускладає командну утиліту Thrift.
3. Ускладає бібліотеки для вибраної мови програмування клієнтів.
4. Згенерувати файли моделі HBase для вибраної мови.
5. Написати та запустити клієнтську програму.

Почнемо із запуску служби Thrift, оскільки це найпростіше. З командного рядка демон запускається так:

```
${HBASE_HOME}/bin/hbase-daemon.sh start thrift -b 127.0.0.1
```

Далі необхідно встановити командну утиліту thrift. Конкретні кроки залежать від середовища проживання і, взагалі кажучи, включають компіляцію вихідного коду. Щоб перевірити правильність установки, виконайте команду з прапором -version. Повинен бути надрукований приблизно такий рядок:

```
$ thrift -version Thrift version 0.6.0
```

Як клієнтська мова ми виберемо Ruby, але для інших мов дії аналогічні. Встановіть gem-пакет Thrift, виконавши таку команду:

```
$ gem install thrift
```

Щоб перевірити правильність установки gem-пакета, виконайте однорядковий скрипт:

```
$ ruby -e "require 'thrift'"
```

Якщо на консолі нічого не надруковано, то все прекрасно! Побачивши повідомлення про помилку типу no such file to load, зупиніться та розберіться у причинах.

Наступний крок – генерація залежних від мови моделей HBase. Модельні файли служать прошарком між встановленими версіями HBase і Thrift, тому вони генеруються, а не поставляються у готовому вигляді.

Для початку знайдіть файл `Hbase.thrift`в каталозі`${HBASE_ HOME}/src`. Шлях до нього виглядає приблизно так:

```
${HBASE_HOME}/src/main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift
```

Тепер згенеруйте модельні файли, підставивши в наступну команду певний шлях:

```
$ thrift --gen rb <path_to_Hbase.thrift>
```

В результаті буде створено новий каталог gen-rb, що містить такі модельні файли:

- hbase_constants.rb;
- hbase.rb;
- hbase_types.rb.

Далі ми будемо використовувати ці файли у клієнтській програмі.

Розробка клієнтської програми

Наша програма підключатиметься до HBase за протоколом Thrift і виводитиме список знайдених таблиць разом із сімействами стовпців. Це можна як перший крок на шляху створення адміністративного інтерфейсу HBase. На відміну від попередніх прикладів, цей скрипт написаний на чистому Ruby, а не на JRuby. Його цілком можна включити до веб-додатку, написаного на Ruby.

Введіть наступний текстовий файл (ми назвали його `thrift_ example.rb`):

hbase/thrift_example.rb

```
$:.push('./gen-rb') require 'thrift' require
'hbase'
socket = Thrift::Socket.new( 'localhost', 9090 ) transport =
Thrift::BufferedTransport.new( socket ) protocol = Thrift::BinaryProtocol.new(
transport )
client = Apache::Hadoop::Hbase::Thrift::Hbase::Client.new( protocol ) transport.open()

client.getTableNames().sort.each do |table| puts "#{table}"


client.getColumnDescriptors( table ).each do |col, desc| puts " #{desc.name}"
puts " maxVersions: #{desc.maxVersions}" puts " compression:
#{desc.compression}"
puts " bloomFilterType: #{desc.bloomFilterType}" end
end

transport.close()
```

Тут ми спочатку налаштовуємо середовище так, щоб Ruby міг знайти модельні файли, для чого додаємо в дорогу каталог gen-rb, а потім включаємо thrift і hbase файли. Далі ми створюємо з'єднання з сервером Thrift та зв'язуємо його з екземпляром клієнта HBase. Об'єкт client знадобиться для комунікації з HBase.

Відкривши об'єкт transport, ми перебираємо всі таблиці, повернуті

методом getTableNames(). Для кожної таблиці ми обходимо список сімейств стовпців, повернутий методом getColumnDescriptors(), і роздруковуємо деякі властивості стандартного виведення.

Тепер виконаємо цю програму з командного рядка. На вашій машині має бути схожий результат, оскільки ми підключаємося до раніше запущеного локального сервера HBase.

```
        $> ruby thrift_example.rb links
        from: maxVersions: 1
        compression: NONE bloomFilterType: ROWCOL
        to:
        maxVersions: 1 compression: NONE bloomFilterType:
ROWCOL
        wiki
        revision:
        maxVersions: 2147483647 compression: NONE
bloomFilterType: NONE
        text:
        maxVersions: 2147483647 compression: GZ
bloomFilterType: ROW
```

Легко бачити, що Thrift API для HBase має в основному ту ж функціональність, що і розглянутий вище Java API, але багато речей виражаються по-іншому. Наприклад, у Thrift замість об'єкта `Put` створюється об'єкт `Mutation` для оновлення одного стовпця або `BatchMutation`, коли в одній транзакції потрібно оновити кілька стовпців.

Файл Hbase.thrift, який ми використовували для створення модельних файлів, добре документований в коментарях, де описані всі доступні структури і методи.

Введення в Whirr

Раніше налаштування хмарного кластера, що працює, вимагало значних зусиль. На щастя, з появою Whirr все змінилося. Whirr, який зараз має статус інкубаційного проєкту Apache, надає засоби для запуску кластера віртуальних машин, підключення до нього та подальшого знищення. Він підтримує такі популярні служби, як Elastic Compute Cloud (EC2) компанії Amazon та Cloud Servers компанії RackSpace. Whirr може використовуватись для налаштування кластерів Hadoop, HBase, Cassandra, Voldemort та ZooKeeper, ведуться також розробки для підтримки інших технологій, наприклад MongoDB та ElasticSearch.

Хоча постачальники хмарних служб, зокрема Amazon, нерідко пропонують засоби для збереження даних після знищення віртуальних машин, ми користуватися ними не будемо. Для наших цілей цілком достатньо тимчасового кластера, дані якого повністю губляться після завершення роботи з ним. Але якщо згодом ви вирішите використовувати HBase у промисловому режимі, то, звичайно, треба буде подбати про постійне сховище. І тоді має сенс подумати – можливо, вам більше підійде виділене обладнання. Динамічні служби типу EC2 - відмінне рішення, коли потрібно оперативно отримати обчислювальні потужності, але, власне кажучи, від кластера виділених фізичних або віртуальних машин можна отримати більшу віддачу.

Перш ніж розпочинати налаштування кластера за допомогою Whirr, необхідно завести обліковий запис в одного з підтримуваних постачальників хмарних служб. У цій лекції ми розповімо про Amazon EC2, але ніхто не заважає вибрати іншого постачальника.

Якщо у вас ще немає облікового запису в Amazon, створіть його на порталі Amazon Web Services (AWS). Зайдіть у свій обліковий запис та активуйте службу EC2, якщо вона ще не активна. Відкрийте сторінку консолі EC2 AWS10 (див. мал. 17).

Для запуску вузлів EC2 вам знадобляться облікові дані AWS. Перейдіть на головну сторінку AWS і виберіть Account→ Security Credentials (Обліковий запис→ Облікові дані). Знайдіть розділ Access Credentials (Облікові дані для доступу) і запишіть кудись свій ідентифікатор ключа доступу (Access Key ID). Натисніть кнопку Show (Показати) під написом Secret Access Key (Секретний ключ доступу) та запишіть виведене значення. Далі під час налаштування Whirr ми називатимемо ці параметри AWS_ ACCESS_KEY_ID і AWS_SECRET_ACCESS_KEY відповідно.

Рис. 17.Консоль Amazon EC2 – екземплярів наразі немає

Отримавши облікові дані EC2, займемося Whirr. Перейдіть на сайт Apache Whirr11 та скачайте останню версію. Розпакуйте завантажений архів і в тому ж каталозі відкрийте вікно команд. Щоб перевірити працездатність Whirr, виконайте команду version.

```
$ bin/whirr version
Apache Whirr 0.6.0-incubating
```

Далі ми створимо не захищені парольною фразою ключі SSH, які Whirr використовуватиме при запуску екземплярів (віртуальних машин).

```
$ mkdir keys
$ ssh-keygen -t rsa -P ''-f keys/id_rsa
```

В результаті буде створено каталог keys, а в ньому – файли id_rsa та id_rsa.pub. Покінчивши з цим, можна розпочинати налаштування кластера.

Налаштування та запуск кластера

Для встановлення параметрів кластера Whirr потрібен файл із розширенням `.properties`. Створіть у каталозі, де знаходиться Whirr, файл `hbase.properties` з таким вмістом (замість `AWS_ACCESS_ KEY_ID` і `AWS_SECRET_ACCESS_KEY` підставте отримані від Amazon значення):

hbase/hbase.properties

```
# постачальник служб whirr.provider=aws-ec2
whirr.identity=your AWS_ACCESS_KEY_ID here whirr.credential=your
AWS_SECRET_ACCESS_KEY here

# облікові дані для ssh whirr.private-key-file=keys/id_rsa whirr.public-key-
file=keys/id_rsa.pub

# конфігурація кластера whirr.cluster-name=myhbasecluster whirr.instance-
templates=\
1 zookeeper+hadoop-namenode+hadoop-jobtracker+hbase-master,\
5 hadoop-datanode+hadoop-tasktracker+hbase-regionserver

# Завдання версій HBase та Hadoop whirr.hbase.tarball.url=\
http://apache.cu.be/hbase/hbase-0.90.3/hbase-0.90.3.tar.gzwhirr.hadoop.tarball.url=\
http://archive.cloudera.com/cdh/3/hadoop-0.20.2-cdh3u1.tar.gz
```

У перших двох розділах визначається постачальник служб і всі

необхідні облікові дані (ця частина загальна для будь-яких кластерів), а в двох останніх – параметри створюваного кластера HBase. Властивість whirr.cluster-name не має значення, якщо ви не маєте наміру запускати одночасно кілька кластерів, а в такому випадку вони повинні мати різні імена. Властивість whirr.instance-templates містить перелічені через кому ролі вузлів із зазначенням кількості вузлів у кожній ролі. В даному випадку ми хочемо мати один головний та п'ять регіональних серверів. Нарешті, властивість `whirr.hbase.tarball.url` інструктує Whirr, що слід використовувати ту версію HBase, з якою ми працюємо у цій книзі.

Зберігши параметри конфігурації у файлі hbase.properties, ми можемо запустити кластер. Перебуваючи в каталозі Whirr, виконайте команду launchcluster, вказавши ім'я створеного вище файлу властивостей.

```
$bin/whirr launch-cluster --config hbase.properties
```

Команда працює порівняно довго та виводить на екран багато інформації. Слідкувати за ходом запуску кластера можна на консолі AWS EC2, її вигляд показаний на рис. 18.

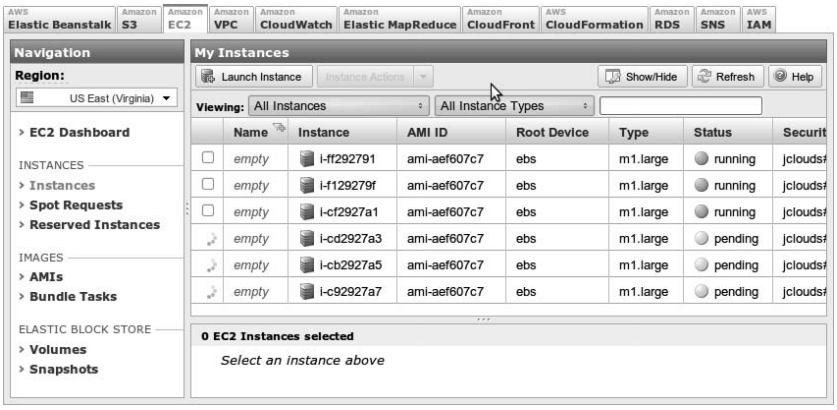


Рис. 18. Консоль Amazon EC2 – показаний хід запуску екземплярів HBase

Додаткову інформацію про стан запуску можна знайти у файлі whirr.log у каталозі Whirr.

За замовчуванням будь-який трафік із кластером має бути захищений, тому для підключення до HBase необхідно відкрити SSH-сеанс. Перш за все нам необхідно знати ім'я того сервера в кластері, до якого ми підключаємося. У вашому домашньому каталозі Whirr створив каталог .whirr/myhbasecluster. У ньому ви знайдете файл instances, в якому перераховані екземпляри

Amazon, що працюють в кластері. У третьому стовпці знаходяться доступні ззовні доменні імена серверів. Підставте у наведений нижче командний рядок ім'я першого сервера.

```
$ ssh -i keys/id_rsa ${USER}@<SERVER_NAME>
```

Після успішного підключення запустіть оболонку HBase:

```
$ /usr/local/hbase-0.90.3/bin/hbase shell
```

В оболонці можна опитати стан кластера за допомогою команди Status.

```
hbase> status
6 servers, 0 dead, 2.0000 average load
```

Починаючи з цього моменту, можна виконувати всі операції, які ми розглядали попередніми днями, зокрема створення таблиць і вставку даних. Підключення клієнтської програми на базі протоколу Thrift ми залишимо як вправу читачеві. Але перед тим, як закінчити, ми повинні розглянути ще одну дію – знищення кластера.

Закінчивши роботу з віддаленим кластером HBase у хмарі EC2, зупиніть його за допомогою команди Whirr destroycluster. Зазначимо, що при цьому будуть втрачені всі введені в кластер дані, оскільки ми не вказали, що налаштовані екземпляри повинні використовувати постійне сховище.

Перебуваючи в каталозі Whirr, виконайте таку команду:

```
$ bin/whirr destroy-cluster --config hbase.properties Destroying
myhbasecluster cluster
Cluster myhbasecluster destroyed
```

Зупинка проводиться досить швидко. Зайшовши на консоль AWS (мал. 19), переконайтеся, що екземпляри справді зупиняються.

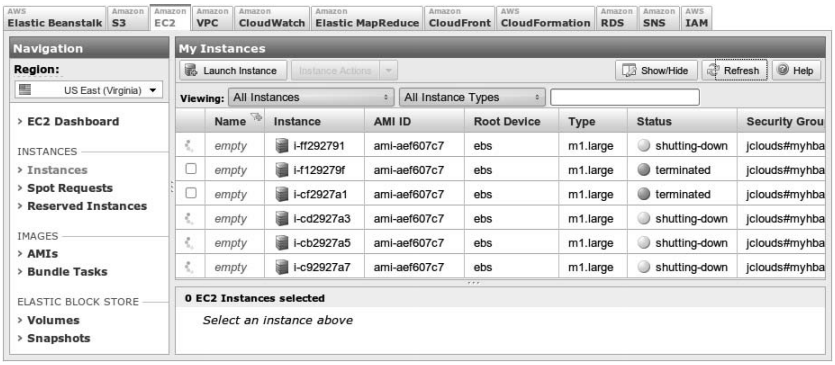


Рис. 19.Консоль Amazon EC2 – показаний хід зупинки екземплярів HBase

Якщо під час операції щось піде не так, ви завжди зможете зупинити

кластер безпосередньо на консолі AWS.

Висновок

У лекції ми вивчили відмінні від оболонки HBase способи підключення, зокрема, двійковий протокол Thrift. Ми розробили клієнтську програму на базі Thrift, а потім створили та налаштували віддалений кластер у хмарі Amazon EC2 за допомогою Apache Whirr.

HBase – це поєднання простоти та складності. Сама модель зберігання даних відносно проста, у схему вбудовуються лише деякі обмеження. Однак щодо системи дуже заважає, що термінологія запозичена зі світу реляційних баз даних (наприклад, це стосується слів таблиця і стовпець). При проєктуванні схеми HBase в основу ставиться продуктивність таблиць і стовпців.

Сильні сторони HBase. Серед особливостей HBase особливої уваги заслуговує архітектура горизонтальної масштабованості та вбудовані засоби версіонування та стиснення. Для деяких завдань вбудована версія – надзвичайно корисна функція. Наприклад, збереження історії версій сторінок вікі-сайту необхідне розробки політик та обслуговування. HBase дозволяє не вживати жодних спеціальних заходів для реалізації історії сторінок – ми отримуємо це безкоштовно.

Якщо говорити про продуктивність, то HBase спочатку задумана для масштабування по горизонталі. Якщо обсяг ваших даних вимірюється гігабайтами чи терабайтами, то HBase – саме те, що треба. HBase є механізми обліку конкретних серверних стійок; реплікація даних між вузлами, встановленими в одній або різних стійках ЦОД, забезпечує швидке відпрацювання відмов без помітного зниження функціональності.

HBase використовують багато відомих компаній, але організації, яка б підтримувала HBase на комерційній основі, немає. Іншими словами, люди з спільноти HBase роблять це виключно на власному ентузіазмі.

Слабкі сторони HBase. Хоча HBase проєктувалася з урахуванням

горизонтального масштабування, існує нижній поріг числа вузлів. Спільнота HBase згідно з тим, що для надійної роботи необхідно не менше п'яти вузлів. Оскільки система варта обробки великих масивів даних, адмініструвати її порівняно важко. Для вирішення невеликих завдань HBase навряд чи годиться, а документації для нефахівців практично не існує, що ускладнює вивчення.

Крім того, HBase майже ніколи не розгортається в ізоляції. Вона є частиною екосистеми масштабованих компонентів, до яких входить Hadoop (незалежна реалізація каркасу MapReduce, вперше запропонованого Google), розподілена файлова система Hadoop (HDFS) та Zookeeper (служба без головного сервера, що координує роботу вузлів). Ця екосистема має як плюси, так і мінуси; вона забезпечує високу стабільність, але покладає на адміністратора тягар відповідальності за обслуговування.

Слід зазначити, що HBase не надає жодних засобів сортування чи індексування, крім ключів рядків. Рядки зберігаються відсортованими за ключами, але ні за яким іншим полем, скажімо по імені та значенню стовпця, сортування не проводиться. Тому для пошуку рядка за будь-яким критерієм, крім її ключа, доведеться або сканувати всю таблицю, або будувати та супроводжувати власний індекс.

Відсутнє поняття типу даних. Значення всіх полів у HBase трактуються як масиви байтів, що не інтерпретуються. Немає різниці між цілим числом, рядком і датою. Для HBase все це просто байти, а їхня інтерпретація покладається на додаток.

HBase та теорема CAP. У термінології CAP HBase, безумовно, належить до класу CP (узгоджена та стійка до втрати зв'язності). HBase дає суворі гарантії узгодженості. Якщо один клієнт успішно записав якесь значення, то решта клієнтів побачить його при наступному запиті. Деякі бази даних, наприклад Riak, дозволяють змінювати параметри CAP лише на рівні окремої операції. Але тільки не HBase. При не надто серйозній втраті зв'язності, наприклад, при виході з ладу одного вузла, HBase залишиться

доступною – відповідальність за обслуговування запитів буде передано іншим вузлам. Але в патологічній ситуації, наприклад, коли працездатність зберіг лише один вузол, HBase не матиме іншого вибору, як відмовити в обслуговуванні запиту.

Обговорення властивостей CAP дещо ускладнюється, якщо взяти до уваги міжкластерну реплікацію, яку ми в цій лекції не розглядали. У типовій багатокластерній конфігурації кластери можуть бути територіально рознесені. У такому разі для будь-якого заданого сімейства стовпців запис проводиться тільки на один кластер, а решта просто надає доступ до реплікованих даних. Така система є узгодженою зрештою, оскільки репліковані кластери повертають останнє значення, про яке знають, а воно може і не збігатися з останнім значенням, записаним на головному кластері.

Завдання для індивідуальної роботи

У сьогоднішньому домашньому завданні від вас потрібно підключити локальну програму на базі Thrift до віддаленого кластера HBase. Для цього потрібно буде відкрити незахищений канал доступу до кластера за протоколом TCP. У виробничому середовищі було б правильніше спочатку створити захищений канал для Thrift – наприклад, організувавши віртуальну приватну мережу (VPN) з кінцевими точками в EC2 та нашій основній мережі. Опис налаштування такої мережі виходить за межі цієї книги; Досить сказати, що ми рекомендуємо захищати трафік, коли це диктується інтересами справи.

1. Запустіть кластер EC2, відкрийте SSH-сеанс із яким-небудь вузлом, запустіть оболонку hbase і створіть таблицю, що містить хоча б одне сімейство стовпців.

2. У тому ж SSH-сеансі запустіть службу Thrift:

```
$ sudo /usr/local/hbase-0.90.3/bin/hbase-daemon.sh start thrift -b 0.0.0.0
```

3. На веб-консолі Amazon EC2 відкрийте порт TCP 9090 у групі безпеки для свого кластера (Network & Security > Security Groups > Inbound >

Create a new rule).

4. Змініть розроблену нами клієнтську програму на базі протоколу Thrift, так щоб вона зверталася до вузла EC2, а не до localhost. Запустіть програму та переконайтеся, що вона відображає правильну інформацію про новостворену таблицю.

Лекція 7. Основи MongoDB

Перевагами MongoDB є гнучкість, міць, простота використання і застосовність для різних завдань, великих і малих. Першу публічну версію MongoDB було випущено в 2009 році, а тепер це висхідна зірка у світі NoSQL. Система замислювалася як масштабована база даних - назва Mongo походить від слова "humongous", що вийшло об'єднанням "huge" (гігантський) і "monstrous" (жахливий), а як основні проєктні цілі були поставлені висока продуктивність і простота доступу до даних. Це документна база даних, яка дозволяє не лише зберігати, а й опитувати вкладені дані, пред'являючи довільні запити. Схема бази даних не нав'язується (у цьому MongoDB схожа на Riak, але відрізняється від Postgres), тому один документ може містити поля чи типи, які відсутні у всіх інших документах колекції. Але не думайте, що гнучкість MongoDB перетворює її на іграшку. Цю базу даних використовують такі гігантські сайти, як Foursquare та bit.ly, а в Європейському центрі ядерних досліджень (ЦЕРН) вона застосовується для зберігання даних,

Mongo

Mongo вдало поєднує потужні засоби запитів, характерні для реляційних баз даних, і розподілену архітектуру, властиву таким сховищам, як Riak або HBase. Mongo - сховище JSON-документів (хоча, строго кажучи, дані зберігаються у двійковому варіанті JSON, який називається BSON). Документ Mongo можна уподібнити до рядка реляційної таблиці без схеми, в якій допускається довільна глибина вкладеності значень. Наступний код допоможе зрозуміти, як виглядає JSON-документ:

```
> printjson( db.towns.findOne({"_id" : ObjectId("4d0b6da3bb3 0773266f39fea")}) )
{
"_id" : ObjectId("4d0b6da3bb30773266f39fea"), "country" : {
"$ref" : "countries",
"$id" : ObjectId("4d0e6074deb8995216a8309e")
},
"famous_for" : [ "beer", "food"
],
"last_census" : "Thu Sep 20 2007 00:00:00 GMT-0700 (PDT)", "mayor" : {
"name" : "Sam Adams",
"party" : "D"
},
```

```
"name" : "Portland", "population" : 582000, "state" : "OR"
}
```

Mongo - відмінний вибір для зростаючого класу веб-проєктів, яких необхідно працювати з великими масивами даних, але бюджет занадто малий на придбання дорогого устаткування. Завдяки відсутності структурованої схеми, Mongo може зростати та змінюватися разом з моделлю даних. Якщо ви працюєте в нещодавно освіченій компанії, яка плекає грандіозні плани або вже накопичила стільки даних, що виникла потреба в горизонтальному масштабуванні, то придивіться до MongoDB.

Операції CRUD та вкладеність

Сьогодні ми розглянемо деякі операції CRUD та виконання запитів до вкладених даних. Як завжди, кроки установки ми опустимо, але з сайту Mongo (http://www.mongodb.org/downloads) ви можете завантажити готовий дистрибутив для своєї ОС та знайти інструкції зі збирання з вихідного коду. Якщо ви працюєте з OS X, рекомендуємо виконувати установку за допомогою програми Homebrew (brew install mongodb). Користувачі Debian/Ubuntu Linux можуть завантажити пакет для apt-get, зібраний самою компанією Mongodb.org.

Щоб уникнути помилок Mongo вимагає, щоб ви заздалегідь створили каталог, де сервер mongod зберігатиме дані. Зазвичай вибирають каталог /data/db. Переконайтеся, що користувач, від імені якого працює сервер, має право читання та запису до цього каталогу. Якщо служба Mongo ще не запущена, це можна зробити, виконавши команду mongod.

Щоб створити нову базу даних book, виконайте у вікні терміналу наведену нижче команду. Вона запускає інтерактивний інтерфейс, влаштований на зразок MySQL.

```
$ mongo book
```

Спочатку наберіть слово help. Наразі поточною є база даних book, але команда show dbs покаже інші бази, а команда use дозволить змінити поточну базу.

Для створення колекції (аналог сегмента в термінології Riak) Mongo

досить просто додати до неї перший запис. Оскільки в Mongo немає схем, заздалегідь нічого визначати не треба. Більше того, навіть сама база даних book фізично не існує, допоки ми не додамо до неї першого документа. Наступний код створює колекцію towns та вставляє в неї дані:

```
> db.towns.insert({ name: “New York”,
population: 22200000,
last_census: ISODate(“2009-07-31”), famous_for: [ “statue of liberty”,
“food” ], mayor : {
name : "Michael Bloomberg", party : "I"
}
})
```

Вище ми згадували, що документи зберігаються у форматі JSON (точніше, BSON), тому в такому форматі ми їх додаємо. Фігурні дужки {...} позначають об'єкт (аналог асоціативного масиву або хеш-таблиці), що містить ключі та значення, а квадратні дужки [...] – лінійний масив. Значення можуть бути вкладеними, причому глибина вкладеності не обмежена. Команда show collections дозволить переконатися, що колекція справді існує.

```
> show collections system.indexes towns
```

Колекція towns створена нами, а колекція system.indexes існує завжди. Переглянути вміст колекції дозволяє команда find(). Для зручності читання ми відформатували висновок, оскільки зазвичай виводиться суцільний рядок.

```
> db.towns.find()
{
"_id" : ObjectId("4d0ad975bb30773266f39fe3"), "name" : "New York",
“population”: 22200000,
"last_census": "Fri Jul 31 2009 00:00:00 GMT-0700 (PDT)", "famous_for" : [ "statue of
liberty", "food" ],
“mayor” : { “name” : “Michael Bloomberg”, “party” : “I” }
}
```

На відміну від реляційних СУБД Mongo не підтримує з'єднання на стороні сервера. Один виклик JavaScript-функції отримує документ і всі вкладені в нього дані.

Можливо, ви звернули увагу на додане системою поле _id типу ObjectId. Це близький аналог ключового слова SERIAL, яке у PostgreSQL служить для автоматичного інкременту числового первинного ключа. Об'єкт ObjectId завжди займає 12 байтів і складається з тимчасової мітки, ідентифікатора клієнтської машини, ідентифікатора клієнтського процесу та 3-байтового лічильника, що інкрементується.

Ця схема автоматичної нумерації має одну безперечну перевагу: будь-який процес на будь-якій машині сам відповідає за генерацію ідентифікаторів, не вступаючи в конфлікт з іншими екземплярами mongod. Це проєктне рішення - прямий натяк на розподілену природу.

JavaScript

Рідною мовою Mongo є JavaScript, який застосовується і в складних MapReduce-запитах, і для найпростішого отримання довідки:

```
> db.help()
> db.towns.help()
```

Ці команди виводять список доступних для цього об'єктафункцій. db є JavaScript-об'єктом, який містить інформацію про поточну базу даних. db.x - JavaScript-об'єкт, що представляє колекцію з ім'ям x. Самі команди – звичайні JavaScript-функції.

```
> typeof db object
> typeof db.towns object

> typeof db.towns.insert функція
```

Щоб ознайомитися з вихідним кодом функції, викличте її без параметрів і без дужок (це більше схоже на Python, ніж Ruby).

```
db.towns.insert
function (obj, _allow_dot) { if (!obj) {
throw “no object passed to insert!”;
}
if (!_allow_dot) {this._validateForStorage(obj);
}
if (typeof obj._id == “undefined”) { var tmp = obj;
obj = {_id: new ObjectId}; for (var key in tmp) {
obj[key] = tmp[key];
}
}
this._mongo.insert(this._fullName, obj); this._lastID = obj._id;
}
```

Давайте додамо ще кілька документів до колекції towns, щоб написати власну JavaScript-функцію.

mongo/insert_city.js

```
function insertCity(
name, population, last_census, famous_for, mayor_info
) {
db.towns.insert({ name:name,
population:population,
last_census: ISODate(last_census), famous_for:famous_for,
mayor : mayor_info
});
}
```

Ви можете просто скопіювати цей код в оболонку, а потім викликати

його.

```
insertCity(“Punxsutawney”, 6200, '2008-31-01', [“phil the groundhog”], {
name : “Jim Wehrle” }
)


insertCity("Portland", 582000, '2007-20-09',
[“beer”, “food”], { name : “Sam Adams”, party : “D” }
)
```

Тепер у колекції towns має бути три міста, у чому можна легко переконатися, викликавши функцію `db.towns.find()`, як і раніше.

Читання з Mongo

Раніше ми викликали функцію find() без налаштувань, щоб отримати всі документи. Для отримання конкретного документа достатньо встановити властивість _id. Оскільки _id має тип ObjectId, то формулювання запиту необхідно перетворити рядок до цього типу, обернувши її функцією ObjectId(str).

```
db.towns.find({ “_id” : ObjectId(“4d0ada1fbb30773266f39fe4”) })
{
"_id" : ObjectId("4d0ada1fbb30773266f39fe4"), "name" : "Punxsutawney",
“population” : 6200,
“last_census” : “Thu Jan 31 2008 00:00:00 GMT-0800 (PST)”, “famous_for” : [ “phil the
groundhog” ],
“mayor” : { “name” : “Jim Wehrle” }
}
```

Функція find() приймає і ще один параметр: об'єкт, що вказує, які поля витягувати з бази. Якщо вас цікавить лише назва міста (і його _id), передайте об'єкт, у якому властивість name набуває значення 1 (або true).

```
db.towns.find({_id: ObjectId(“4d0ada1fbb30773266f39fe4”) }, { name : 1 })
{
"_id" : ObjectId("4d0ada1fbb30773266f39fe4"), "name" : "Punxsutawney"
}
```

Щоб витягти всі поля, крім`name`, задайте властивість`name` рівним 0 (або `false`, або `null`).

```
db.towns.find({_id: ObjectId(“4d0ada1fbb30773266f39fe4”) }, { name : 0 })
{
"_id" : ObjectId("4d0ada1fbb30773266f39fe4"), "population" : 6200,
“last_census” : “Thu Jan 31 2008 00:00:00 GMT-0800 (PST)”, “famous_for” : [ “phil the
groundhog” ]
}
```

Як і PostgreSQL, Mongo дозволяє пред'являти довільні запити за значеннями полів, діапазонів або з декількома критеріями. Щоб знайти всі міста, назви яких починаються з літери P, а чисельність населення менше 10 000, можна скористатися сумісним із Perl синтаксисом регулярних виразів

(PCRE)2 та оператором діапазону.

```
db.towns.find(
{ name : /^P/, population : { $lt : 10000 } },
{ name : 1, population : 1 }
) {
"name": "Punxsutawney", "population": 6200}
```

Умовні оператори в Mongo записуються як field : { $op : value }, де $op – операція, наприклад $ne (не одно). Хотілося б, щоб синтаксис був коротшим, скажімо field < value. Але це код на JavaScript, а не предметно-орієнтована мова запитів, тому запити повинні підкорятися синтаксичним правилам JavaScript (нижче ми побачимо, що в деяких випадках короткий синтаксис все ж таки припустимо, але поки не будемо про це).

Але є й хороші новини: оскільки мова запитів – це JavaScript, то операції можна конструювати як об'єкти. Нижче конструюється умова чисельність населення: від 10 000 до мільйона людина.

```
var population_range = {} population_range['$lt'] =
1000000
population_range['$gt'] = 10000 db.towns.find(
{ name : /^P/, population : population_range },
{ name: 1 }
)

{ "_id" : ObjectId("4d0ada87bb30773266f39fe5"), "name" : "Portland" }
```

Можливі не лише числові діапазони, а й діапазони дат. Знайдемо всі міста, в яких дата останнього проведення перепису (last_census) менша або дорівнює 31 січня 2008 року:

```
db.towns.find(
{ last_census : { $lte : ISODate('2008-31-01') } },
{ _id : 0, name: 1 }
)

{ "name" : "Punxsutawney"}
{ "name" : "Portland" }
```

Зверніть увагу, що ми придушили включення поля _id у результат, явно надавши відповідній властивості значення 0.

Mongo любить вкладені масиви. У запиті можна ставити порівняння з конкретним значенням.

```
db.towns.find(
{ famous_for : 'food'},
{ _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "New York", "famous_for" : [ "statue of liberty", "food" ] }
{ "name" : "Portland", "famous_for" : [ "beer", "food" ] }
```

… або порівняння з підрядком…

```
db.towns.find(
{ famous_for : /statue/ },
{ _id : 0, name : 1, famous_for : 1 }
)

{ “name” : “New York”, “famous_for” : [ “statue of liberty”, “food” ] }
```

…або збіг кожного з кількох значень…

```
db.towns.find(
{ famous_for : { $all : ['food', 'beer'] } },
{ _id : 0, name:1, famous_for:1 }
)

{ “name” : “Portland”, “famous_for” : [ “beer”, “food” ] }
```

...або розбіжність з жодним із зазначених значень:

```
db.towns.find(
{ famous_for : { $nin : ['food', 'beer'] } },
{ _id : 0, name : 1, famous_for : 1 }
)

                    { “name” : “Punxsutawney”, “famous_for” : [ “phil the groundhog” ] }
```

Але справжня міць Mongo полягає у можливості зазирнути всередину документа та повернути результати пошуку глибоко вкладених піддокументів. Щоб опитати піддокумент, ім'я поля має бути рядком, у якому рівні вкладеності розділені точками. Наприклад, ось як можна знайти міста, на чолі яких стоять незалежні мери…

```
db.towns.find(
{ 'mayor.party' : 'I'},
{ _id : 0, name : 1, mayor : 1 }
)

{


"name" : "New York", "mayor" : {
"name": "Michael Bloomberg", "party": "I"
}
}
```

… чи безпартійні мери:

```
db.towns.find(
{ 'mayor.party' : { $exists : false } },
{ _id : 0, name : 1, mayor : 1 }
)

{ “name” : “Punxsutawney”, “mayor” : { “name” : “Jim Wehrle” } }
```

Наведені вище запити підходять, коли потрібно знайти документи з одним відповідним критерієм полем. Але якщо потрібно порівнювати кілька полів піддокументу?

Оператор elemMatch

Завершимо наш екскурс знайомством з оператором $elemMatch.

Створимо ще одну колекцію, в якій зберігатимемо країни. На цей раз перевизначимо поле _id, задавши рядок на свій вибір:

```
db.countries.insert({
_id : “us”,
name : “United States”, exports : {
foods : [
{ name : “bacon”, tasty : true },
{name: "burgers"}
]
             }
            })
db.countries.insert({
_id : "ca",
name : “Canada”, exports : {
              foods : [
{ name : “bacon”, tasty : false },
{ name : “syrup”, tasty : true }
]
             }
            })
db.countries.insert({
_id : "mx",
name : "Mexico", exports : {
               foods : [{
name : "salsa",


tasty: true, condiment: true
}]
              }
            })
```

Щоб перевірити, скільки країн було додано, ми можемо скористатись функцією count, яка має повернути 3.

```
> print( db.countries.count() ) 3
```

Давайте пошукаємо країну, яка експортує не просто бекон, а *смачний* (Tasty) бекон.

```
db.countries.find(
{ 'exports.foods.name' : 'bacon', 'exports.foods.tasty' : true },
{ _id : 0, name : 1 }
)

{ “name” : “United States” }
{ “name” : “Canada”}
```

Але це не те, що ми хотіли. Mongo повернула Канаду (Canada), тому що звідти експортується бекон та смачний сироп (syrup). На допомогу приходить оператор $elemMatch. Він означає, що документ (або вкладений піддокумент) вважається слушним, якщо задовольняє всім критеріям.

```
db.countries.find(
{
'exports.foods' : {
$elemMatch : { name : 'bacon', tasty : true
}
               }
             },
{ _id : 0, name : 1 }
)
```

```
{ “name” : “United States” }
```

В операторі $elemMatch допускаються і складніші умови. Так, можна знайти всі країни, які експортують смачну їжу, приправлену ще й спеціями (condiment):

```
db.countries.find(
{
               'exports.foods' : {
                $elemMatch : {


tasty : true,
condiment : { $exists : true }
}
               }
              },
{ _id : 0, name : 1 }
)
{ “name” : “Mexico” }
```

Отримали Мексику.

Булівські операції

До цих пір у всіх умовах неявно малася на увазі зв'язка `u`. Якщо запросити країну з назвою United States та ідентифікатором `_id`, що дорівнює mx, то Mongo нічого не знайде.

```
db.countries.find(
{ _id : “mx”, name : “United States” },
{ _id : 1 }
)
```

Однак пошук того чи іншого – за допомогою оператора $or – поверне два результати. Цей оператор записується у префіксній нотації: `OR A B`.

```
db.countries.find(
{
$or : [
{ _id : “mx” },
{ name : “United States” }
              ]
},
{_id:1}
)

{ “_id”: “us”}
{ “_id” : “mx”}
```

Існує ще багато інших операторів, нижче наведено неповний, але досить представницький список операторів.

Команда	Опис
`$ne`	Не дорівнює
`$lt`	Менше
`$lte`	Менше або дорівнює

$gt	Більше
$gte	Більше або дорівнює
$exists	Перевіряє існування поля
$all	Відповідність усім елементам масиву
$in	Відповідність хоча б одному елементу масиву
$nin	Невідповідність жодному елементу масиву
$elemMatch	Відповідність усіх полів вкладеного документа
$or	Або
$nor	Не чи
$size	Відповідність розміру масиву
$mod	Розподіл по модулю
$type	Відповідність, якщо поле має вказаний тип
$not	Заперечення

Всі оператори описані в онлайновій документації MongoDB.

Оновлення

Функція update(criteria,operation) приймає два обов'язкові параметри. Перший – критерій відбору – такий, як функції find(). Другий – або об'єкт, поля якого замінюють поля відібраних документів, або модифікатор. У цьому випадку модифікатор $set записує в полі state рядок OR.

```
db.towns.update(
{ _id : ObjectId("4d0ada87bb30773266f39fe5") },
{ $set : { "state" : "OR" } }
);
```

Може виникнути питання, навіщо потрібна операція $set. Mongo не працюєу термінах атрибутів; на внутрішньому рівні є лише неявне представлення про атрибути з метою оптимізації. Однак в інтерфейсі Mongo жодних атрибутів не згадується, є тільки документи. Навряд чи ви коли-небудь захочете виконати щось таке (зверніть увагу на відсутність оператора $set):

```
db.towns.update(
{ _id : ObjectId("4d0ada87bb30773266f39fe5") },
{ state : "OR" }
);
```

У цьому випадку весь відповідний документ був замінений переданим вами документом ({ state : "OR" }). Якщо ви не вказали команду, наприклад $set, Mongo вважає, що ви просто хочете повністю замінити документ. Будьте уважні.

Щоб перевірити, чи було оновлення успішним, ми можемо пошукати

документ (зверніть увагу на використання функції findOne() для пошуку лише одного відповідного документа).

```
db.towns.findOne({ _id : ObjectId("4d0ada87bb30773266f39fe5") })
{
"_id" : ObjectId("4d0ada87bb30773266f39fe5"), "famous_for" : [
"beer", "food"
],
"last_census" : "Thu Sep 20 2007 00:00:00 GMT-0700 (PDT)", "mayor" : {
"name" : "Sam Adams",
"party" : "D"
},
"name" : "Portland", "population" : 582000,
"state" : "OR"
}
```

До значення можна використовувати не тільки оператор $set. Часто буває корисний також оператор $inc (збільшити число). Давайте збільшимо чисельність населення Портленду на 1000.

```
db.towns.update(
{ _id : ObjectId("4d0ada87bb30773266f39fe5") },
{$inc:{population:1000}}
)
```

Існують і інші оператори, наприклад, позиційний оператор $ для масивів. Нові оператори додаються досить часто та описуються в онлайновій документації. Нижче наведено список найважливіших операторів.

Команда	Опис
`$set`	Записує вказане значення у вказане поле
`$unset`	Видаляєполе
`$inc`	Додає вказане число до вказаного поля
`$pop`	Видаляєостанній (або перший) елемент із масиву
`$push`	Поміщає новий елемент у масив
`$pushAll`	Поміщає всі вказані елементи в масив
`$addToSet`	Аналогічний `push`, але дублікати не додаються
`$pull`	Видаляєз масиву потрібне значення, якщо воно в ньому є
`$pullAll`	Видаляєз масиву всі відповідні значення

Посилання

Ми вже зазначали, що Mongo не призначена для виконання з'єднань. Через розподілену природу системи з'єднання виявилися б вкрай неефективною операцією. Проте іноді корисно, щоб документи могли посилатися один на одного. Для таких випадків Mongo є конструкція виду { $ref : "collection_ name", $id : "reference_id" }. Наприклад, можна додати до колекції towns посилання на документ із колекції countries.

```
db.towns.update(
{ _id : ObjectId("4d0ada87bb30773266f39fe5") },
{ $set : { country: { $ref: "countries", $id: "us" } } }
)
```

Тепер можна отримати з колекції towns дані про Портленд:

```
var portland = db.towns.findOne(
{ _id : ObjectId("4d0ada87bb30773266f39fe5") })
```

Щоб отримати відомості про країну, в якій знаходиться місто, ми можемо опитати колекцію countries, передавши збережений раніше ідентифікатор $id.

```
db.countries.findOne({ _id: portland.country.$id })
```

Можна навіть зробити ще краще: запитати у документа про місто ім'я колекції, що зберігається в полі посилання:

```
db[ portland.country.$ref ].findOne({ _id: portland.country.$id })
```

Останні два запити еквівалентні, але другий більшою мірою керується даними.

Видалення

Видалити документи з колекції нескладно. Достатньо замість функції find() скористатися функцією remove() – і всі документи, які відповідають критерію, будуть видалені. Важливо відзначити, що *видаляється документ повністю*, а не тільки елемент, що збігся, або піддокумент.

Ми рекомендуємо спочатку виконувати find(), щоб переконатися в правильності критерію, а потім remove(). Mongo не буде сім разів відміряти, перш ніж виконати операцію. Давайте видалимо всі країни, які експортують бекон несмачний.

```
var bad_bacon = {'exports.foods': {
$elemMatch : { name : 'bacon', tasty : false
}
}
}
db.countries.find( bad_bacon )

{
"_id" : ObjectId("4d0b7b84bb30773266f39fef"), "name" : "Canada",
"exports" : {
"foods" : [
{
"name": "bacon", "tasty": false
},
{
"name": "syrup", "tasty": true
}
]
```

```
}
}
```

Начебто все добре. Видаляємо.

```
db.countries.remove( bad_bacon ) db.countries.count()
2
```

Тепер виконайте команду count() і перевірте, що залишилося лише дві країни. Якщо так, то вилучення пройшло успішно.

Функціональні критерії

На останок розглянемо більш цікаву можливість представлення запитів – шляхом виконання коду. Можна вимагати, щоб MongoDB застосовувала до документів функцію, яка приймає рішення. Ми залишили цей розділ наостанок, тому що скористатися описаною тут можливістю слід тільки в крайньому випадку. Такі запити виконуються дуже повільно, вони не враховують наявні індекси, і Mongo не в змозі їх оптимізувати. Але іноді ніщо не може зрівнятися із виразною потужністю спеціалізованого коду.

Припустимо, що потрібно знайти міста, де проживає від 6000 до 600 000 осіб.

```
db.towns.find( function() {
return this.population > 6000 && this.population < 600000;
} )
```

У Mongo навіть є скорочена нотація для простих функцій ухвалення рішення:

```
db.towns.find("this.population > 6000 && this.population < 600000")
```

Цю функцію можна використовувати разом із іншими умовами, скориставшись фразою $where. У наступному прикладі відбираються також міста, славні своїми бабаками (groundhog):

```
db.towns.find( {
$where : "this.population > 6000 && this.population < 600000", famous_for : /groundhog/
} )
```

Проте Mongo просто обчислюватиме цю функцію для кожного документа, хоча немає жодної гарантії, що згадуване в ній поле взагалі існує. Наприклад, якщо ви припускаєте, що поле population існує, а хоча б в одному документі воно відсутнє, весь запит завершиться помилкою, тому що написаний на JavaScript код неможливо буде правильно виконати. Будьте уважні при написанні власних функцій прийняття рішень та постарайтеся

обійтися стандартними засобами.

Висновок

Сьогодні ми познайомилися з першою в цій книзі документною базою даних MongoDB. Ми бачили, що вона дозволяє зберігати вкладені документи у вигляді JSON-об'єктів і опитувати їх полями, розташованими на будь-якому рівні вкладеності. Ви дізналися, що документ можна розглядати як безсхемний рядок реляційної моделі зі згенерованим ключем _id. Набір документів, який Mongo називається колекцією, - аналог таблиці в PostgreSQL.

На відміну від баз даних, що раніше зустрічалися нам, в Mongo зберігаються не набори даних простих типів, а складні денормалізовані документи, представлені у вигляді колекцій довільних JSON-об'єктів. При цьому Mongo доповнює гнучку стратегію зберігання потужним механізмом запитів, не обмеженим наперед визначеною схемою.

Завдяки деноралізації документне сховище є відмінним вибором для зберігання даних із заздалегідь невідомими властивостями, тоді як в інших СУБД (реляційних або стовпцевих) типи даних потрібно знати заздалегідь, а для додавання або зміни полів необхідна міграція схеми.

Завдання для індивідуальної роботи

1. Розберіться, що видають команди `db.help()`і`db.collections.help()`.
2. Знайдіть драйвер своєї улюбленої мови програмування для Mongo (Ruby, Java, PHP тощо).
1. Надрукуйте JSON-документ, що містить`{ "hello": "world"}`.
2. Знайдіть міста, назви яких містять слово new, використовуючи регулярний вираз без урахування регістру.
3. Створіть нову базу даних blogger та в ній колекцію articles. Вставте нову статтю, яка містить ім'я та електронну поштову адресу автора,

дату створення та текст.

4. Змініть статтю, додавши масив коментарів; кожен коментар повинен містити ім'я автора та текст.

5. Виконайте запит, який зберігається у зовнішньому JavaScript-файлі.

Лекція 8. Індексування, угруповання, mapreduce у MongoDB

У лекції першим пунктом нашого плану буде підвищення продуктивності запитів у MongoDB. За ним будуть складніші запити з угрупованням. І завершимо лекцію аналізом даних за допомогою технології mapreduce за аналогією з тим, що робили при вивченні Riak.

Індексування: коли швидкодії не вистачає

Однією з корисних функцій Mongo є можливість індексування для підвищення швидкості виконання запитів – річ, яка, як ми бачили, є не у всіх NoSQL-базах. MongoDB підтримує кілька перевірених часом структур індексів, у тому числі класичне B-дерево, а також двовимірні та сферичні просторові індекси.

Для початку проведемо невеликий експеримент, щоб оцінити ефективність B-дерев у MongoDB: введемо послідовність телефонних номерів із довільним префіксом країни (можете підставити префікс своєї країни, якщо хочете). Введіть за допомогою консолі наступний код, який згенерує 100 000 номерів від 1-800-555-0000 до 1-800-565-9999 (це може тривати деякий час).

mongo/populate_phones.js

```
populatePhones = function(area,start,stop) { for(var i=start; i < stop;
i++) {
var country = 1 + ((Math.random() * 8) << 0); var num = (country * 1e10) +
(area * 1e7) + i; db.phones.insert({
_id: num, components: {
country: country, area: area,
prefix: (i * 1e-4) << 0, number: i
},
display: "+" + country + " " + area + "-" + i
});
}
}
```

Виконайте цю функцію, задавши тризначний код зони (наприклад, 800) та діапазон семизначних номерів (від 5550000 до 5650000, при введенні звертайте увагу на кількість нулів).

```
populatePhones( 800, 5550000, 5650000 ) db.phones.find().limit(2)

{ "_id": 18005550000, "components": { "country": 1, "area": 800, "prefix": 555, "number":
5550000}, "display": "+1 800-55500
```

```
{ "_id": 88005550001, "components": { "country": 8, "area": 800, "prefix": 555, "number": 5550001}, "display": "+8 800-55500
```

При створенні будь-якої колекції Mongo автоматично будує індекс _id. Ці індекси є у колекції system. indexes. Наступний запит покаже всі індекси, що є в базі даних:

```
db.system.indexes.find()

{ "name" : "_id_", "ns" : "book.phones", "key" : { "_id" : 1 } }
```

Як правило, у запитах вказують й інші поля, крім _id, тому за ними слід побудувати індекси.

Ми побудуємо індекс зі структурою B-дерева по полю display. Але оскільки наша мета – переконатися, що індекс справді прискорює роботу, ми спочатку виконаємо запит без індексу. Щоб отримати інформацію про те, як виконується операція, ми скористаємося методом explain().

```
db.phones.find({display: "+1 800-5650001"}).explain()
{
"cursor" : "BasicCursor", "nscanned" : 109999,
"nscannedObjects" : 109999,
"n": 1,
"millis": 52, "indexBounds": {
}
}
```

На вашому комп'ютері результат може відрізнятися, але розмір millis – час виконання запиту в мілісекундах – швидше за все, буде двозначним.

Для побудови індексу викликається метод колекції назви Index(fields,options). Параметр fields – це об'єкт, що містить поля, якими будується індекс, а параметр options описує тип індексу. В даному випадку нам потрібен унікальний індекс по полю display, причому виявлені при побудові дублікати повинні просто видалятися.

```
db.phones.ensureIndex(
{ display : 1 },
{ unique : true, dropDups : true }
)
```

Тепер знову виконаємо find() і подивимося, що поверне `explain()`.

```
db.phones.find({ display: "+1 800-5650001"}).explain()
{
"cursor" : "BtreeCursor display_1", "nscanned" : 1,
"nscannedObjects" : 1,
"n": 1,
"millis": 0, "indexBounds": {
"display" : [ [
"+1 800-5650001",
"+1 800-5650001"
]
]
```

```
}
}
```

Величина millis впала від 52 до 0 – прискорення на кілька порядків є. Зверніть також увагу, що поле cursor тепер не Basic, а BtreeCursor (курсор вказує на місце зберігання даних, але сам ніяких даних не містить). Mongo більше не виконує повне сканування колекції, а обходить дерево для пошуку потрібного значення. Важливо й те, що кількість переглянутих об'єктів знизилася з 109 999 до 1 – оскільки індекс унікальний.

explain() – корисна функція, але використовувати її слід лише під час перевірки конкретних запитів. Для вивчення продуктивності системи у тестовому чи промисловому режимі знадобиться *системний профільник*.

Задамо рівень профілювання 2 (при рівні 2 зберігається інформація про всі запити, при рівні 1 – тільки про «повільні» запити, що виконували більше 100 мілісекунд) і запустимо find(), як завжди.

```
db.setProfilingLevel(2)
db.phones.find({ display : “+1 800-5650001” })
```

В результаті створюється новий об'єкт у колекції system.profile,яку можна читати, як будь-яку іншу. У полі ts зберігається час початку запиту, у полі info – рядок з описом операції, а полі millis – витрачений час.

```
db.system.profile.find()
{
“ts” : ISODate(“2011-12-05T19:26:40.310Z”),
"op" : "query",
"ns" : "book.phones",
“query” : { “display” : “+1 800-5650001” }, “responseLength” : 146,
"millis": 0,
"client" : "127.0.0.1",
"user" : ""
}
```

Mongo вміє будувати індекси і за значеннями на глибших рівнях вкладеності. Щоб побудувати індекс за кодами зон, потрібно скористатися нотацією назви полів з точками: components.area. У промисловій системі індекс завжди слід будувати у фоновому режимі, задаючи параметр {background:1}:

```
db.phones.ensureIndex({ “components.area”: 1 }, { background : 1 })
```

Якщо тепер за допомогою find() пошукати індекси для колекції `phones`, то новий індекс опиниться у списку останнім. Першим йде індекс по полю _id, який завжди створюється автоматично, а другим – збудований нами

унікальний індекс.

```
db.system.indexes.find({ “ns” : “book.phones” })

{
"name" : "_id_",
"ns" : "book.phones",
"key" : { "_id" : 1 }
}
{
"_id" : ObjectId("4d2c96d1df18c2494fa3061c"), "ns" : "book.phones",
"key" : { "display" : 1 }, "name" : "display_1",
"unique" : true, "dropDups" : true
}
{
"_id" : ObjectId("4d2c982bdf18c2494fa3061d"), "ns" : "book.phones",
"key" : { "components.area" : 1}, "name" :
"components.area_1"
}
```

На закінчення відзначимо, що для створення індексу для великої кількості даних може знадобитися багато часу та системних ресурсів. Це слід враховувати та намагатися будувати індекси у періоди найменшого навантаження, у фоновому режимі та вручну, не застосовуючи автоматизованих процедур. У Мережі є і додаткові поради щодо індексування, але це основи, про які непогано б пам'ятати.

Агреговані запити

Розглянуті раніше запити корисні для того, щоб отримати дані, але їх подальша обробка покладалася на клієнта. Нехай, наприклад, потрібно підрахувати кількість телефонів, більших за 559-9999; хотілося б, звісно, щоб такі операції вміла виконувати сама СУБД. Як і в PostgreSQL, функція count() є найпростішим агрегатором. Вона приймає запит і повертає число (кількість документів, що задовольняють критерію).

```
db.phones.count({'components.number': { $gt : 5599999 } }) 50000
```

Щоб повною мірою усвідомити міць інших агрегованих запитів, додамо до колекції телефонів ще 100 000 номерів з іншим кодом зони.

```
populatePhones( 855, 5550000, 5650000 )
```

Агреговані запити повертають структуру, відмінну від окремих документів, яких ми звикли. Функція `count()` агрегує результати, повертаючи кількість документів,`distinct()` повертає масив результатів, а`group()` - Документи, які сама конструює. Навіть mapreduce у загальному

випадку намагається повертати об'єкти, що нагадують документи, які ви зберегли.

Команда distinct() повертає значення, що задовольняють критерію (не повні документи), якщо хоча б одне існує. Ось як можна отримати різні номери (без урахування зони), менші за 5 550 005:

```
db.phones.distinct('components.number',
{'components.number': { $lt : 5550005 } })

[5550000, 5550001, 5550002, 5550003, 5550004]
```

Хоча номер 5550000 зустрічається двічі (у зоні 800 і в зоні 855), до списку він увійшов лише один раз.

Функція group() агрегує результати приблизнотак само, як запити з фразою GROUP BY у SQL. Це найскладніший з агрегірванних запитіву Mongo. Ми можемо підрахувати всі номери, великі 5599999, і помістити їх в окремі групи за кодом зони. Функції передаються три параметри: key – поле, яким виробляється угруповання; cond - умова, яка визначає, які значення нас цікавлять; reduce – функція, що вирішує, як виводити результати. Пам'ятаєте про каркас mapreduce, який обговорювався на чолі про Riak? Наші дані вже відображено на наявну колекцію документів. Більше ніякого розподілу не потрібно, достатньо просто редукувати документи.

```
db.phones.group({ initial: { count:0 },
reduce: function(phone, output) {output.count++; }, cond: {
'components.number': { $gt : 5599999 } }, key: { 'components.area' : true }
})

[ { "800": 50000, "855": 50000}]
```

Наступні два приклади дещо штучні та призначені лише для демонстрації гнучкості group().

Функцію count() неважко реалізувати самостійно з допомогою наступного виклику group(). Ключ агрегування до результату не включається.

```
db.phones.group({ initial: { count:0 },
reduce: function(phone, output) {output.count++; }, cond: {
'components.number': { $gt : 5599999 } }
})
[{“count”: 100000}]
```

Тут ми спочатку налаштовуємо початковий об'єкт, записавши в полі count значення 0 – задані в цьому об'єкті поля будуть присутні в результаті. Потім ми описуємо, що робити з цим полем, – оголошуємо функцію reduce,

яка додає до count одиницю для кожного документа, що зустрівся. І нарешті, ми задаємо для групи умову, що обмежує множину документів, що редукуються. Результат буде такий самий, як при виклику функції count(), тому що задано таку ж умову. Ключ ми опустили, тому що хочемо, щоб у список поміщався кожен документ, що зустрівся. Можна реалізувати функцію distinct(). Для підвищення продуктивності ми почнемо зі створення об'єкта, в якому номери зберігаються у вигляді полів (по суті ми створюємо власний варіант структури даних *множина*).

Якщо ми хочемо повністю повторити поведінку функції distinct(), то необхідно повернути масив цілих чисел. Тому ми додамо метод finalize(out), який викликається один раз перед поверненням значення, щоб перетворити об'єкт на масив значень полів. Потім ця функція перетворює рядкові телефонні номери на цілі числа (якщо ви хочете спостерігати, як смажиться омлет, виконайте наведену нижче функцію, на задаючи функцію finalize).

```
        db.phones.group({
        initial: { prefixes : {} }, reduce: function(phone, output)
{
        output.prefixes[phone.components.prefix] = 1;
        },
        finalize: function(out) { var ary = [];
        for(var p in out.prefixes) {ary.push(parseInt(p)); } out.prefixes = ary;
        }
        })[0].prefixes

        [555, 556, 557, 558, 559, 560, 561, 562, 563, 564]
```

Функція group() не менш потужна, ніж фраза GROUP BY в SQL, але в реалізації її в Mongo є недоліки. По-перше, результат не може містити понад 10 000 документів. До того ж, якщо колекція сегментована (цю тему ми розглядатимемо завтра), то group() взагалі не працюватиме. Є ще багато способів гнучкого настроювання запиту. З цієї та низки інших причин ми присвятимо ще трохи часу питання реалізації mapreduce в Mongo. Але спочатку окреслимо кордон між командами на стороні сервера та клієнта, оскільки різниця між ними відіграватиме важливу роль у ваших додатках.

Команди на стороні сервера

Якщо ви запустите наступну функцію з командної оболонки (або

програми, написаної за допомогою мовного драйвера), то клієнт буде запитувати кожен зі 100 000 телефонів, модифікувати його і зберігати новий документ на сервері.

mongo/update_area.js

```
update_area = function() { db.phones.find().forEach(
function(phone) {phone.components.area++; phone.display =
"+"+ phone.components.country+" "+ phone.components.area+"-"+
phone.components.number;
db.phone.update({ _id : phone._id }, phone, false);
}
)
}
```

Однак об'єкт Mongo db має в своєму розпорядженні команду eval(), яка передає вказану функцію серверу. Це кардинально зменшує трафік між клієнтом та сервером, тому що код виконується віддалено.

```
> db.eval(update_area)
```

Крім обчислення JavaScript-функцій, в Mongo вбудований ще ряд готових команд, більша частина яких виконується на сервері, хоча для деяких необхідно, щоб поточною була база даних admin (зробити її поточною дозволяє команда use admin).

```
> use admin
> db.runCommand("top")
```

Команда top виводить відомості про всі колекції сервера.

```
> use book
> db.listCommands()
```

Команда listCommands() виводить список команд, багато з яких ми вже використовували. Насправді багато типових команд, наприклад підрахунок телефонних номерів, можна виконати за допомогою методу runCommand(). Проте результати трохи відрізнятимуться.

```
> db.runCommand({ "count" : "phones" })
{ "n": 100000, "ok": 1}
```

Саме число (n) обчислено правильно (100 000), але у відповіді повертається об'єкт, у якому поле ok. Пояснюється це тим, що db.phones.count() – функція обгортка, створена для нашої зручності оболонкою, тоді як метод runCommand() виконується на сервері. Нагадаємо, що ми можемо вивчити вихідний код будь-якої функції, зокрема count(), набравши її ім'я без наступних дужок.

```
> db.phones.count function (x) {
return this.find(x).count();
}
```

Метод `collection.count()`- Проста обгортка для виклику count() стосовно результатів функції find() (а та й сама є обгорткою навколо вбудованого об'єкта запиту, який повертає курсор, що вказує на результати). Якщо ж виконати такий запит…

```
> db.phones.find().count
```

то ми побачимо куди довшу функцію (надто довгу, щоб наводити тут її код). Але придивіться до коду – після різного роду ініціалізації ви знайдете такі рядки:

```
var res = this._db.runCommand(cmd); if (res && res.n !=
null) {
return res.n;
}
```

Функція `count()` виконує метод `runCommand()` та повертає значення поля `n`.

І якщо ми вже завели розмову про те, як працюють методи, розглянемо ще й функцію runCommand()

```
> db.runCommand function (obj) {
if (typeof obj == "string") { var n = {};
n[obj] = 1; obj = n;
}
return this.getCollection("$cmd").findOne(obj);
}
```

Виявляється, що `runCommand()`- теж допоміжна функція, що обгортає виклик методу колекції $cmd. Будь-яку команду можна виконати безпосередньо звернувшись до цієї колекції.

```
> db.$cmd.findOne({'count' : 'phones'})
{ "n": 100000, "ok": 1}
```

Але це вже рівень «заліза» – так спілкуються із сервером Mongo мовні драйвери.

Будь-яку JavaScript-функцію можна зберегти у спеціальній колекції system.js. Це звичайна колекція і щоб зберегти в ній функцію, потрібно просто записати її ім'я в полі _id, а об'єкт-функцію – у полі value.

```
> db.system.js.save({
_id:'getLast', value:function(collection){
return collection.find({}).sort({'_id':1}).limit(1)[0]
            }
          })
```

А далі можна виконати цю функцію прямо на сервері, передавши її ім'я

функції eval(). Сервер виконає JavaScript-код та поверне результати.

```
> db.eval('getLast(db.phones)')
```

Результатиповинен бути такими ж, як при локальному виклику`getLast(collection)`.

```
> db.system.js.findOne({'_id': 'getLast'}).value(db.phones)
```

Слід зазначити, що на час виконання eval() робота сервера mongod блокується, тому такий підхід застосовується головним чином для запуску швидких одноразових завдань або тестів, а не як загальний механізм виконання процедур у виробничій системі. Збережену функцію можна викликати також із оператора $where та з функцій mapreduce. Тепер наш арсенал поповнився останнім інструментом, необхідним, щоб розпочати дослідження каркасу mapreduce у MongoDB.

Mapreduce (і Finalize)

Принцип роботи mapreduce у Mongo аналогічний тому, що ми бачили в Riak, але є й невеликі відмінності. Функція map() не повертає перетвореного значення; натомість Mongo вимагає, щоб розподільник викликав функцію emit(), передаючи їй ключ. Ідея в тому, що функцію emit() можна викликати кілька разів під час обробки одного документа. Функція reduce() приймає один ключ та список значень, емітованих для цього ключа. Нарешті, Mongo має необов'язковий третій крок, finalize(), який виконується рівно один раз на кожне розподілене значення – після завершення роботи редукторів. Це дозволяє зробити завершальні обчислення чи очищення.

Згенеруємо звіт, у якому кожної країни підраховується кількість телефонів, що містять однаковий набір цифр. Для початку збережемо допоміжну функцію, яка повертає масив різних цифр у номері телефону (для розуміння принципу роботи mapreduce пристрій цієї функції не настільки важливий).

mongo/distinct_digits.js

```
distinctDigits = function(phone){ var
number = phone.components.number + '', seen = [],
result = [],


i = number.length; while(i--) {
seen[+number[i]] = 1;
}
for (i=0; i<10; i++) { if (seen[i]) {
```

```
result[result.length] = i;
}
}
return result;
}
db.system.js.save({_id: 'distinctDigits', value: distinctDigits})
```

Завантажте цей файл в mongo оболонку. Якщо файл знаходиться в тому ж каталозі, з якого запускається програма mongo, достатньо вказати тільки ім'я файлу, інакше – повний шлях до нього.

```
> load('distinct_digits.js')
```

Потім виконаємо простенький тест (якщо ви не цілком розумієте, що робить функція, не робіть і понавставляйте виклики print()).

```
db.eval("distinctDigits(db.phones.findOne({'components.number': 5551213 }))") [ 1, 2,
3, 5 ]
```

Теперу нас є чим зайняти розподільник. Як завжди в mapreduce, рішення про те, які поля повинен формувати розподільник, є критичним, тому що від цього залежить, які будуть повернуті агреговані значення. Оскільки наш звіт показує телефони з різними цифрами, то одним полем буде масив цифр, що розрізняються. А оскільки ми хочемо запитувати дані щодо країни, то другим полем буде країна. Скомпонуємо з обох полів складовий ключ: {digits: X, country: Y}.

Наше завдання – просто підрахувати кількість значень, тому емітуватимемо значення 1 (кожен документ додає одиницю до підсумку). Завдання редуктора – підсумувати ці одиниці.

```
mongo/map_1.js
map = function() {
var digits = distinctDigits(this);
emit({digits: digits, country : this.components.country}, {count : 1});
}

mongo/reduce_1.js
reduce = function(key, values) {var total = 0;
for(var i=0; i<values.length; i++) { total += values[i].count;
}


return {count: total};
}

results = db.runCommand({ mapReduce: 'phones',
map: map,
reduce: reduce, out: 'phones.report'
})
```

У параметрі out ми задали ім'я колекції (out:phones. report'), і її можна опитати, як будь-яку іншу. Це матеріалізоване представлення, яке показує команда show tables.

```
> db.phones.report.find({'_id.country' : 8})
{
"_id": { "digits": [0, 1, 2, 3, 4, 5, 6], "country": 8}, "value": {"count": 19}
}
{
"_id": { "digits": [0, 1, 2, 3, 5], "country": 8}, "value": {"count": 3}
}
{
"_id": { "digits": [0, 1, 2, 3, 5, 6], "country": 8}, "value": {"count": 48}
}
{
"_id": { "digits": [0, 1, 2, 3, 5, 6, 7], "country": 8}, "value": {"count": 12}
}
has more
```

Введіть it, щоб продовжити обхід результатів. Зауважте, що унікальні емітовані ключі знаходяться в полі _id, а дані, повернуті редукторами, – у полі value.

Якщо ви віддаєте перевагу тому, щоб редуктор просто виводив результати, а не записував їх до колекції, то можете присвоїти out значення `{ inline : 1 }`, але пам'ятайте про обмеження розміру результату. У версії Mongo 2.0 він складає 16 МБ.

Нагадаємо, що у розділі, присвяченому Riak, зазначалося, що на вхід редуктора може подаватися як вихід розподільника, так і вихід інших редукторів. Ось, як це могло б виглядати під час запуску на різних серверах (рис. 22).

Кожен сервер повинен виконувати власні екземпляри функцій map() і reduce(), а потім передавати результати для об'єднання тому серверу, який ініціював виклик. Це класичний алгоритм «розділяй і володарюй». Якби ми назвали вихід редуктора total, а не count, то довелося б обробляти обидва випадки, як показано нижче.

mongo/reduce_2.js

```
reduce = function(key, values) {var total = 0;
for(var i=0; i<values.length; i++) { var data = values[i];
if('total' in data) { total += data.total;
} else {
total + = data.count;
}
}
return {total: total};
}
```

Однак розробники Mongo передбачали, що можуть знадобитися якісь зміни на завершальному етапі, наприклад, перейменування поля або додаткові обчислення. Якщо нам справді так важливо, щоб поле результату

називалося total, можна написати функцію finalize(), яка працює так само, як у випадку group().

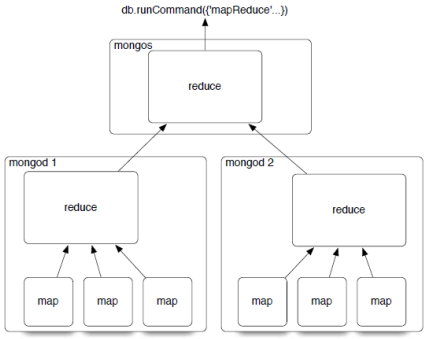


Рис. 22. Розподільники та редуктори працюють на двох серверах Mongo

Висновок

Сьогодні ми розглянули складніші запити, що включають агрегатні функції: count(), distinct() і, нарешті, group(). Для прискорення їх виконання ми користувалися вбудованими MongoDB засобами індексування. Якщо цього недостатньо, то до наших послуг каркас mapreduce.

Завдання для індивідуальної роботи

1. Користуючись JavaScript-інтерфейсом, дослідіть код трьох методів колекцій: `help()`, `findOne()` і `stats()`.

1. Реалізуйте метод finalize для виведення сумарного лічильника як поля total.

2. Установіть драйвер обраної мови програмування для Mongo та підключіться до бази даних. Заповніть якусь колекцію та побудуйте над нею індекс по одному з полів.

Лекція 9. Набори реплік, сегментування, просторові дані, GridFS у MongoDB

Mongo має розвинені засоби для зберігання та опитування даних. Але вони є й у інших СУБД. Унікальна особливість документних баз даних полягає у здатності ефективно обробляти безсхемні документи із довільним рівнем вкладеності. А саме Mongo відрізняється вмінням масштабуватись на кілька серверів шляхом реплікації (копіювання даних на інші сервери) або сегментування колекцій (розбиття колекції на частини), а також умінням паралельно виконувати запити. Те й інше підвищує доступність.

Репліки

Mongo розроблялася не як монолітний сервер, а з прицілом на горизонтальне масштабування. У проєкт закладено узгодженість даних та стійкість до часткової втрати зв'язності мережі. Однак сегментування має свої мінуси: якщо якась частина колекції втрачена, то і вся колекція втрачає цінність. Mongo вирішує цю внутрішньо властиву сегментування проблему дуже просто: за рахунок дублювання даних на кілька серверів.

Сьогодні ми почнемо з чистого аркуша та запустимо кілька нових серверів. Mongo за замовчуванням слухає порт 27017, тому додатковим серверам ми призначимо інші порти. Нагадаємо, що спочатку потрібно створити каталоги даних:

```
$ mkdir ./mongo1 ./mongo2 ./mongo3
```

Далі запустимо сервери Mongo. Цього разу поставимо прапор replSet зі значенням book і вкажемо номери портів. Заодно поставимо прапор rest, щоб можна було звертатися до веб-інтерфейсу.

```
$ mongod --replSet book --dbpath ./mongo1 --port 27011 --rest
```

Відкрийте ще одне вікно терміналу та запустіть аналогічну команду для запуску другого сервера, вказавши інший каталог та інший порт. Так само запустіть третій сервер у третьому вікні терміналу.

```
$ mongod --replSet book --dbpath ./mongo2 --port 27012 --rest
$ mongod --replSet book --dbpath ./mongo3 --port 27013 --rest
```

Зверніть увагу, що буде виводитись наступне повідомлення:

```
[startReplSets] replSet can't get local.system.replset config from self\ or any seed
(EMPTYCONFIG)
```

Це нормально – ми ще тільки маємо ініціалізувати набір реплік, і Mongo нагадує про це. Відкрийте оболонку mongo, підключіться до будь-якого сервера та виконайте функцію rs.initiate().

```
$ mongo localhost:27011
> rs.initiate({


_id: 'book', members: [
{_id: 1, host: 'localhost:27011'},
{_id: 2, host: 'localhost:27012'},
{_id: 3, host: 'localhost:27013'}
                ]
})
> rs.status()
```

Тут скористалися новим об'єктом rs (replica set – набір реплік). У нього, як будь-якого іншого об'єкта, є метод help(). Команда status() дозволяє дізнатися, чи працює цей набір реплік, продовжуйте з її допомогою перевіряти стан, доки ініціалізація не завершиться, тільки потім можна буде продовжувати. Спостерігаючи за тим, що виводять усі три сервери, ви виявите, що один надрукує рядок

```
[rs Manager] replSet PRIMARY
```

а два інших – рядок

```
[rs_sync] replSet SECONDARY
```

Той сервер, що надрукував PRIMARY, є основним. Швидше за все, це буде сервер, що прослуховує порт 27011 (оскільки він був запущений першим); якщо це не так, підключіть консоль до головного сервера. Спробуйте вставити якесь довільне слово з командного інтерфейсу і подивіться, що вийде.

```
> db.echo.insert({ say : 'HELLO!' })
```

Після вставки вийдіть з консолі та перевірте, чи зміна реплікована. Для цього зупиніть головний вузол – досить просто натиснути CTRL+C. Заглянувши в журнали двох серверів, ви виявите, що один з них підвищений в ранзі до головного (він виведе рядок replSet PRIMARY). Відкрийте консоль і підключіться до цієї машини (у нас це був сервер localhost:27012), після чого виконайте команду db.echo.find() – вона має повернути введене значення раніше.

Виконаємо перетасовування консолей ще раз. Підключіть консоль до підпорядкованого (SECONDARY) сервера. Про всяк випадок виконайте функцію isMaster(). У нашому випадку її результат виглядав так:

```
$ mongo localhost:27013 MongoDB shell version: 1.6.2
connecting to: localhost:27013/test

> db.isMaster()
{
"setName" : "book", "ismaster" : false,
"secondary" : true, "hosts" : [
“localhost:27013”, “localhost:27012”,
“localhost:27011”
],
“primary” : “localhost:27012”, “ok” : 1
}
```

У цьому екземплярі оболонки спробуємо вставити ще одне значення.

```
> db.echo.insert({ say : 'is this thing on?' }) not master
```

Повідомлення not master означає, що ми можемо писати на підлеглий сервер. І читати безпосередньо з нього також не вийде. У кожному наборі реплік є лише один головний вузол, і взаємодіяти потрібно саме з ним. Це шлюз для всього набору.

З реплікацією даних пов'язані проблеми, які у базах даних із єдиним джерелом. У випадку Mongo одна з проблем полягає в тому, щоб вирішити, який вузол підвищувати у ранзі, коли головний сервер виходить з ладу. Mongo вирішує її, проводячи вибори процесу mongod. Перемагає той, у якого найсвіжіші дані. На даний момент у вас має працювати дві служби mongod. Спробуйте зупинити головний вузол. Коли ми зробили це, маючи три вузли, один із решти став новим головним. Але тепер все буде інакше. У журналі останнього сервера з'явиться запис такого вигляду:

```
 [ReplSetHealthPollTask] replSet info localhost:27012 is now down (або...

[rs Manager] replSet can’t see a majority, will not try to elect self
([rs Manager] replSet не бачить більшості, не намагаюся вибрати себе)
```

Звідси стає зрозумілою філософія, що стоїть за конфігуруванням серверів в Mongo, і причина, через яку число серверів завжди має бути непарним (три, п'ять і т. д.).

А тепер знову запустіть інші сервери та загляньте в журнали. Коли вузол піднімається, він перетворюється на стан відновлення і намагається ресинхронізувати свої дані з новим головним вузлом.

Якщо у вихідного головного вузла були дані, які ще не встигли поширитися, то такі операції просто відкидаються. Операція запису в наборі реплік не вважається успішною, доки на більшості вузлів не буде збережено копію даних.

Проблема парної кількості вузлів

Ідея реплікації досить проста: ви пишете на один сервер MongoDB, а потім дані копіюються на решту вузлів набору реплік. Якщо головний сервер недоступний, то на його роль вибирається один із тих, що залишилися, і він продовжує обслуговувати запити. Однак сервер може виявитися недоступним не тільки внаслідок виходу з ладу. Іноді зникає мережне з'єднання між вузлами. У такому випадку Mongo заявляє, що мережа складається з більшості вузлів, які все ще здатні бачити один одного.

MongoDB очікує, що в наборі реплік непарне число вузлів. Розглянемо, наприклад, мережу із п'яти вузлів. Якщо внаслідок втрати зв'язності мережа розпалася на два фрагменти – з трьома та двома вузлами, то більший фрагмент становить явну більшість, тому може вибрати головний вузол та продовжувати обслуговування запитів. Без явної більшості зібрати кворум неможливо.

Щоб зрозуміти, чому необхідна непарна кількість вузлів, розглянемо, що може статися, коли в наборі реплік чотири вузли. Припустимо, що в результаті втрати зв'язності утворилося два сегменти із двох вузлів. В одному з них залишається вихідний головний вузол, але оскільки він не бачить явної більшості мережі, то знімає з себе обов'язки головного сервера. Але і в іншому фрагменті вибрати головний вузол неможливо, тому що немає зв'язку з більшістю вузлів. Ні той, ні інший набір не можуть обслуговувати запити і вся система зупиняється. За наявності непарного числа вузлів цей сценарій – фрагментована мережа, у жодному фрагменті якої немає явної більшості, – менш імовірний.

Деякі СУБД (наприклад, CouchDB) допускають наявність кількох головних вузлів, але Mongo до них не належить і невміє вирішувати

конфлікти, що виникають під час оновлення даних. MongoDB підходить до конфліктів між кількома головними вузлами просто – не допускає їх зовсім.

На відміну від, скажімо, Riak, Mongo завжди знає останнє записане значення; клієнту нічого не треба вирішувати. Мета Mongo – забезпечити строгу узгодженість під час запису, і заборона кількох головних вузлів – не найгірший метод її досягнення.

Не завжди можна забезпечити наявність непарного числа серверів для реплікації даних. У такому разі можна або призначити спеціального арбітра (рекомендується), або дати більше голосів деяким серверам (не рекомендується). У Mongo арбітром називається член набору реплік, який бере участь у голосуванні, але не займається реплікацією. Запускається як звичайний сервер, але в конфігураційному файлі задається спеціальний прапор, наприклад:`{_id: 3, host: 'localhost:27013',arbiterOnly: true}`. Арбітри корисні для виходу з глухого кута. За замовчуванням у кожного примірника `mongod` є рівно один голос.

Сегментування

Одна з основних причин існування Mongo – безпечна та швидка обробка великих наборів даних. Очевидний спосіб досягнення цієї мети - горизонтальне сегментування за діапазонами значень - або для стислості просто сегментування (sharding). Замість зберігання всієї колекції на одному сервері, вона розбивається на частини (іншими словами, сегментується), які розміщуються на декількох серверах. Наприклад, ми могли б помістити телефонні номери, менші за 1-500-000-0000, на сервер A, а великі або рівні 1-500-000-0001 – на сервер B. Mongo спрощує вирішення цього завдання за допомогою механізму автосегментування, який самостійно здійснює розбиття.

Давайте запустимо два (нереплікуючі) сервери mongod. Як і у випадку набору реплік, існує спеціальний параметр, що означає, що сервер може

брати участь у сегментуванні.

```
$ mkdir ./mongo4 ./mongo5
$ mongod --shardsvr --dbpath ./mongo4 --port 27014
$ mongod --shardsvr --dbpath ./mongo5 --port 27015
```

Тепер потрібно зробити так, щоб сервер справді вів облік ключів. Припустимо, що ми створили таблицю для збереження назв міст у алфавітному порядку. Необхідно якось повідомити, що назви, що починаються з букв від A до N (наприклад) повинні зберігатися на сервері mongo4, а з букв від O до Z - на сервері mongo5. Mongo для цієї мети створює конфігураційний сервер (звичайний екземпляр mongod, запущений зі спеціальним прапором), який враховує, який сервер (mongo4 або mongo5) які значення зберігає.

```
$ mkdir ./mongoconfig
$ mongod --configsvr --dbpath ./mongoconfig --port 27016
```

Нарешті нам необхідний ще четвертий сервер, mongos, що є єдиною точкою входу для клієнтів. Сервер mongos підключається до сервера конфігурації (назвемо його mongoconfig) для отримання інформації про сегментування. Запустимо його на порту 27020, задаючи параметр chunkSize (розмір порції) рівним 1. (Порція розміром 1 МБ є мінімально допустимою. Таке значення дозволить спостерігати за тим, як відбувається сегментування нашого невеликого набору даних. У виробничій системі слід залишити значення за замовчуванням або вибрати набагато більший розмір.) Про місцезнаходження конфігураційного сервера (доменне ім'я: номер порту) mongos дізнається завдяки прапору

```
--configdb.
$ mongos --configdb localhost:27016 --chunkSize 1 --port 27020
```

Може виникнути питання, навіщо Mongo розділяє ролі *конфігурація* і *точка входу* (`mongos`). Пов'язано це про те, що у виробничих системах ці процеси зазвичай запускаються на фізично різних серверах. Конфігураційний сервер (який сам реплікується) управляє інформацією про сегментування на користь інших сегментованих серверів, тоді як `mongos` зазвичай запускається на тій же машині, де працює сервер додатків, щоб клієнти

могли легко до нього підключатися (не думаючи про те, де знаходяться конкретні сегменти).

Цікаво, що mongos є полегшеною версією повнофункціонального mongod сервера. Майже всі команди розпізнаються mongod, розпізнаються і mongos, що робить його ідеальним посередником для клієнтів, яким необхідно підключатися до кількох сегментованих серверів. На рис. 23 зображено конфігурацію наших серверів.

Тепер підключіть консоль до бази даних admin на сервері `mongos`. Зараз ми займемося конфігуруванням сегментів.

```
$ mongo localhost:27020/admin
> db.runCommand( { addshard : "localhost:27014" } )
{ "shardAdded": "shard0000", "ok": 1}
> db.runCommand( { addshard : "localhost:27015" } )
{ "shardAdded": "shard0001", "ok": 1}
```

Далі необхідно вказати базу даних, що підлягає сегментуванню колекцію та поле, за яким сегментувати (у нашому випадку назва міста).

```
> db.runCommand( { enablesharding : "test" } )
{ "ok": 1}
> db.runCommand( { shardcollection : "test.cities", key : {name : 1} } )
{ "collectionsharded" : "test.cities", "ok" : 1 }
```

Отже, налаштування завершено і можна завантажувати дані. У вихідному коді, який додається до книги, є файл mongo_cities1000. json розміром 12 МБ, в якому містяться відомості про всі міста світу з населенням більше 1000 осіб. Завантажте цей файл та запустіть скрипт, який імпортує дані на наш сервер mongos:

```
$ mongoimport -h localhost:27020 -db test --collection cities \
--type json mongo_cities1000.json
```

На підключеній до `mongos` консолі введіть команду `usetest`, щоб повернутися до бази даних `test`.

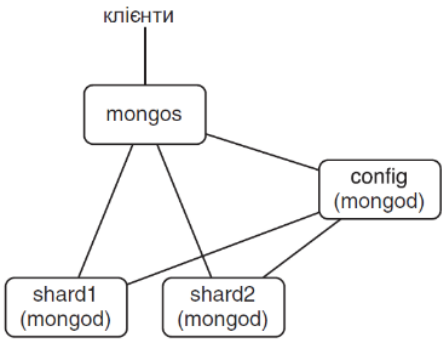


Рис. 23. Маленький сегментований кластер

Просторові запити

У Mongo є ще одна чудова річ, яка називається просторові запити. Спочатку підключіться до сервера mongos.

```
$ mongo localhost:27020
```

Секрет просторових запитів полягає у спеціальному способі індексування географічних даних, який називається геохешуванням. Такий індекс дозволяє шукати не лише за значенням або діапазоном значень координат, але й швидко знаходити прилеглі точки. До речі, у попередньому розділі ми завантажили великий масив географічних даних. Щоб звертатися до них із запитами, ми повинні насамперед проіндексувати дані по полю location. Для побудови двовимірного індексу (типу 2d) необхідно вказати будь-які два поля. У нашому випадку вони представлені у вигляді хешу (наприклад, {longitude:1.48453, latitude:42.57205}), але з тим самим успіхом могли б бути елементами масиву (наприклад, [1.48453, 42.57205]).

```
> db.cities.ensureIndex({ location : “2d” })
```

Якби колекція була сегментована, можна було б запитувати, яке місто (чи міста) перебуває у цій точці чи поруч із нею. Але в поточній версії Mongo наведений нижче запит працює лише для несегментованих колекцій.

```
> db.cities.find({ location : { $near : [45.52, -122.67] } }).limit(5)
```

У майбутніх версіях планується поширити цю можливість і сегментовані колекції. А поки що для пошуку прилеглих міст у сегментованій колекції cities користуйтесь командою geoNear(). Ось як виглядає повертається їй результат:

```
> db.runCommand({geoNear : 'cities', near : [45.52, -122.67], num : 5, maxDistance :
1})
{
"ns" : "test.cities",
"near" : "1000110001000000011100101011100011001001110001111110",
"results" : [
{
"dis" : 0.007105400003747849,
“obj” : {
"_id" : ObjectId("4d81c216a5d037634ca98df6"), "name" : "Portland",
...
                }
},
...
],
"stats" : {
"time" : 0,
"btreelocs" : 53,
```

```
"nscanned" : 49,
"objectsLoaded" : 6,
"avgDistance" : 0.02166813996454613,
"maxDistance" : 0.07991909980773926
},
"ok": 1
}
```

Команда geoNear() заодно допомагає налагоджувати просторові команди. Вона повертає джерело корисної інформації, наприклад: відстань до вказаної в запиті точки, середня та максимальна відстань для поверненого набору, а також відомості про сам індекс.

GridFS

Один із недоліків розподіленої системи – відсутність єдиної файлової системи. Допустимо, ви працюєте з сайтом, на який користувачі можуть завантажувати фотографії. Якщо веб-сайт обслуговується кількома веб-серверами, розміщеними в різних вузлах, то необхідно або вручну копіювати завантажені зображення на кожен сервер, або організовувати якусь централізовану файлову систему. Mongo для такого випадку має власну розподілену файлову систему GridFS.

У дистрибутив Mongo включено командну утиліту для взаємодії з GridFS. Що чудово, для роботи з нею нічого не потрібно налаштовувати. Якщо зараз за допомогою команди `mongofiles` запитати файли на керованих mongos сегментах, ми отримаємо порожній список.

```
$ mongofiles -h localhost:27020 List connected to:
localhost:27020
```

Але спробуйте завантажити якийсь файл.

```
$ mongofiles -h localhost:27020 put my_file.txt connected to:
localhost:27020
added file: { _id: ObjectId('4d81cc96939936015f974859'), \ filename:
"my_file.txt", chunkSize: 262144, \ uploadDate: new Date(1300352150507)
md5: "844ab0d45e3bded0d48c2e77ed4f3b0e", length: 3067 } done!
```

І якщо тепер знову виконати mongofiles, то ми виявимо завантажений файл.

```
$ mongofiles -h localhost:27020 List connected to:
localhost:27020 my_file.txt 3067
```

Повернувшись до консолі mongo, можна подивитися, в яких колекціях Mongo зберігає дані.

```
> show collections cities
```

```
fs.chunks fs.files system.indexes
```

Так як це звичайні колекції, їх можна реплікувати і опитувати звичними способами.

На цьому завершується дослідження MongoDB. Сьогодні ми познайомилися з тим, як Mongo забезпечує довговічність даних за допомогою наборів реплік, та з підтримкою горизонтальної масштабованості за рахунок сегментування. Ми показали, як слід конфігурувати кластер серверів, і прояснили роль сервера mongos як посередника, який управляє автосегментуванням між кількома вузлами. Нарешті, ми поекспериментували з додатковими вбудованими Mongo засобами, а саме з просторовими запитами і розподіленою файловою системою GridFS.

Висновок

Сподіваємося, що знайомство з MongoDB зацікавило вас і довело, що недаремно вона заслужила назву «монструозної» бази даних.

Сильні сторони Mongo. Головна риса Mongo – здатність обробляти гігантські масиви даних (і величезний потік запитів) за рахунок реплікації та горизонтального масштабування. Але вона також пропонує дуже гнучку модель даних, тому що не потрібно заздалегідь визначати схему, а дані можуть бути вкладеними (для досягнення аналогічного ефекту РСУБД довелося б використовувати з'єднання).

Нарешті, під час проєктування MongoDB закладалася простота використання. Можливо, ви звернули увагу на схожість між командами Mongo та деякими концепціями баз даних на основі SQL (за вирахуванням з'єднань на стороні сервера). Це не випадково, і саме з цієї причини Mongo приваблює так багато прихильників з-поміж колишніх користувачів об'єктно-реляційних моделей (ORM). Ця система має достатньо відмінностей, щоб підбадьорити багатьох розробників, але не настільки багато, щоб відлякати своєю чужістю.

Слабкі сторони Mongo. Те, що Mongo заохочує денормалізацію схеми

(хоча самі схеми відсутні), деяким може здатися неприйнятним. Є розробники, які жорсткі обмеження, що накладаються реляційної СУБД з її холодною логікою, вселяють впевненість. Можливість вставлення в будь-яку колекцію значення довільного типу викликає побоювання. Єдина помилка може стати причиною багатогодинних мук, якщо ви не здогадаєтеся шукати причину в іменах полів та колекцій. Гнучкість Mongo - не така вже й велика перевага, якщо модель даних вже досить зріла і давно зафіксована.

Оскільки Mongo орієнтована великі набори даних, то використовувати її має сенс у великих кластерах, для проєктування та обслуговування яких потрібні деякі зусилля. На відміну від Riak, де додавання нових вузлів здійснюється прозоро і майже непомітно в процесі експлуатації, налаштування кластера Mongo передбачає попереднє планування.

Mongo – відмінний вибір для тих, хто вже звично використовує для зберігання даних реляційну базу даних та звертається до неї за допомогою якоїсь системи об'єктно-реляційного відображення. Ми часто рекомендуємо її працюючим на платформах Rails, Django і взагалі всім, хто користується патерном модель-представлення-контролер (MVC), тому що вони все одно реалізують перевірку даних та керування полями за допомогою моделей на рівні програми і оскільки з міграцією схеми можна буде розпрощатися (здебільшого). Для додавання нових полів у документ досить просто додати поле до моделі даних, і Mongo спокійно сприйме нові поняття. Ми вважаємо, що порівняно з реляційними базами даних Mongo дає набагато природніше рішення багатьох типових завдань, у яких набір даних визначається додатком.

Завдання для індивідуальної роботи

1. Прочитайте в онлайновій документації про всі параметри конфігурації набору реплік.
2. З'ясуйте, як створюється тривимірний просторовий індекс (за сферичними координатами).
1. У Mongo є підтримка обмежуючих фігур (точніше, квадратів та

кіл). Знайдіть усі міста, розташовані в межах 50 миль від центру Лондона (http://www.mongodb.org/display/DOCS/Geospatial+Indexing).

2. Запустіть шість серверів: по три сервери у двох наборах реплік, кожен із яких є сегментом. Запустіть конфігураційний сервер та mongos. Налаштуйте у такій конфігурації GridFS (це вирішальний іспит).

Лекція 10. Основи CouchDB

База даних Apache CouchDB може масштабуватися вгору і вниз, тому легко адаптується до завдань будь-якого розміру і складності. CouchDB – типова документоорієнтована база даних на основі JSON та REST. Вже перша версія, випущена в 2005 році, була спроєктована в розрахунку на мережу Інтернет і всі незліченні помилки, відмови та глюки, які їй супроводжують. Тому за стабільністю з CouchDB не може зрівнятися майже жодна з баз даних. Якщо інші системи здатні пережити випадкові зникнення мережі, то CouchDB чудово почувається навіть тоді, коли поява мережі – рідкісна подія.

Як і MongoDB, CouchDB зберігає документи – JSON-об'єкти, що складаються з пар ключ-значення, причому значеннями можуть бути дані різних типів, у тому числі інші об'єкти з необмеженою вкладеністю. Проте довільні запити не підтримуються; Основний спосіб пошуку документів – це індексовані представлення, що породжуються інкрементною процедурою mapreduce.

CouchDB (couch - кушетка) розрахована не лише на гігантські кластери з дорогих серверів, а й на інші сценарії розгортання: від центру обробки даних до смартфона. Запустити CouchDB можна на телефоні Android, на персональному MacBook або в ЦОДі. Будучи написана на Erlang, CouchDB на подив живуча: зупинити її можна лише одним способом - вбивши процес! А модель зберігання, яка допускає лише дописування, гарантує практичну непошкоджуваність даних, а також простоту реплікації, резервного копіювання та відновлення.

CouchDB – документоорієнтована база даних, в якій форматом зберігання та взаємодії із зовнішнім світом є JSON. Як і Riak, всі звернення до CouchDB проводяться за допомогою REST-інтерфейсу. Реплікація може бути як одно-, так і двосторонньою, на запит або безперервною. CouchDB забезпечує високу гнучкість при прийнятті рішень про структуру, захист та

розподіл даних.

Одне з найцікавіших питань– у чому різниця між CouchDB та MongoDB? На перший погляд, CouchDB та MongoDB виглядають дуже схожим. Обидві – документоорієнтовані сховища даних, тісно пов'язані з мовою JavaScript. В обох форматах транспортування даних є JSON. Тим не менш, існує багато відмінностей, починаючи з ідеології проєкту і закінчуючи реалізацією та характеристиками масштабованості.

Інтерфейс Futon

До складу CouchDB входить корисний веб-інтерфейс, який називається Futon (*футон, традиційна японська постільна приналежність у вигляді товстого бавовняного матраца*). Встановивши та запустивши CouchDB, відкрийте браузер та перейдіть за адресою http://localhost:5984/_utils/. Відкриється сторінка Overview (Загальні відомості), що показана на рис. 24.

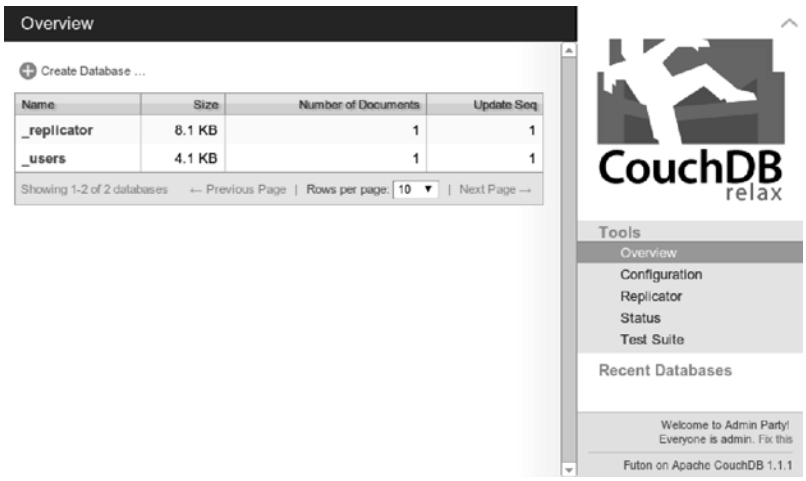


Рис. 24. CouchDB Futon: сторінка Overview

Перш ніж розпочинати роботу з документами, ми повинні створити базу даних для їх зберігання. У нашій базі ми зберігатимемо відомості про музикантів, альбоми та композиції. Клацніть на Create Database... (Створити базу даних). У спливаючому вікні введіть ім'я бази music та натисніть Create. В результаті ви будете перенаправлені на сторінку бази даних. Тут можна створювати нові документи та відкривати існуючі.

На сторінці бази даних клацніть посилання New Document (Новий документ). Ви перейдете на сторінку на рис. 25.

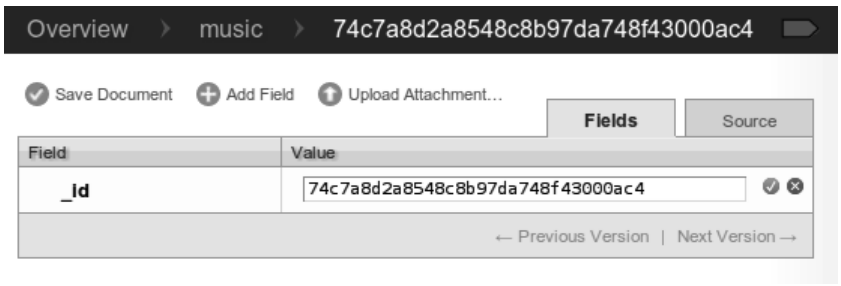


Рис. 25. CouchDB Futon: створення документа

Як і в MongoDB, документ являє собою JSON-об'єкт, що складається з пар ключ-значення, які називаються полями. Будь-який документ CouchDB має поле _id, яке має бути унікальним і ніколи не змінюється. Можна встановити значення _id явно, але якщо ви цього не зробите, CouchDB згенерує його автоматично. Нас це влаштовує, тому клацніть на посилання Save Document (Зберегти документ).

Відразу після збереження CouchDB запише до документа додаткове поле _rev. Йому надається нове значення щоразу, як у документ вносяться зміни. Структурно це рядок, де спочатку йде ціле число, потім дефіс, а потім псевдовипадковий унікальний рядок. Початкове ціле число – це номер редакції, наразі 1.

Імена полів, що починаються зі знака підкреслення, є CouchDB спеціальними; їх _id і _rev особливо важливі. Щоб оновити чи видалити існуючий документ, необхідно вказати як _id, так і _rev. Якщо значення не відповідають один одному, CouchDB відмовиться виконувати операцію. Саме так запобігають конфліктам – гарантується, що будь-яка модифікація застосовується до останньої версії даних.

У CouchDB немає транзакцій, чи блокувань. Щоб модифікувати існуючий запис, ви спочатку читаєте його та запам'ятовуєте значення _id і _rev. Потім ви запитуєте оновлення, надаючи документ повністю, включаючи поля _id і _rev. Усі операції виконуються строго по порядку – першим прийшов, першим обслужений. Вимагаючи узгодженості _rev, CouchDB гарантує, що документ, який ви модифікуєте, не був за вашою спиною кимось змінений.

На сторінці документа клацніть посилання Add Field (Додати поле). У стовпці Field введіть ім'я поля name, а стовпці Value – значення The Beatles. Клацніть по зеленій галочці зліва, щоб підтвердити свій намір, а потім – за посиланням Save Document (Зберегти документ). Зверніть увагу, що поле _rev тепер починається з цифри 2.

CouchDB може зберігати не тільки рядки, а й довільні JSON-структури з необмеженою вкладеністю. Знову клацніть на посилання Add Field. Цього разу назвіть поле albums, а в стовпці Value введіть наступні назви альбомів групи Бітлз (список неповний):

```
[
"Help!",
"Sgt. Pepper's Lonely Hearts Club Band", "Abbey Road"
]
```

Після клацання за посиланням Save Document сторінка має виглядати так, як показано на рис. 26.

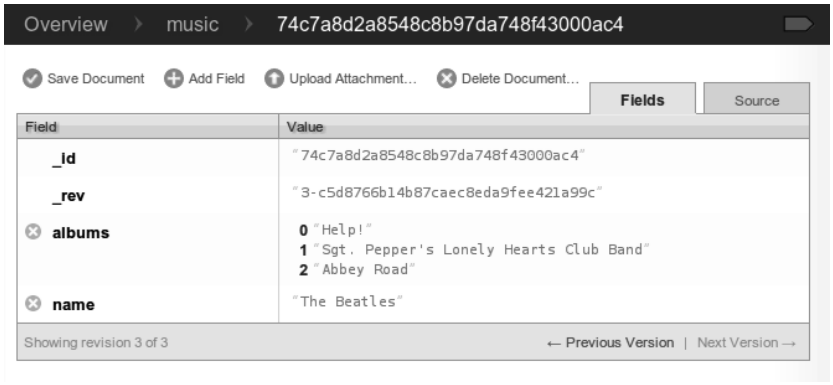


Рис. 26. CouchDB Futon: документ зі значенням-масивом

Альбом характеризується не тільки назвою, тому давайте введемо і додаткову інформацію. Замініть значення у полі albums таким:

```
[{
"title": "Help!", "year": 1965
}, {
"title": "Sgt. Pepper's Lonely Hearts Club Band", "year": 1967
}, {
"title": "Abbey Road", "year": 1969
}]
```

Тепер після збереження документа ви зможете розкрити значне значення albums і показати вкладені документи (рис. 27).

Рис. 27. CouchDB Futon: документ із глибоким рівнем вкладеності

Клік на посиланні Delete Document (Видалити документ) робить те, що й очікується, – видаляє документ з бази даних music. Але поки давайте перейдемо до командного рядка і подивимося, як взаємодіяти з CouchDB через REST-інтерфейс.

Виконання операцій CRUD за допомогою REST-інтерфейсу та cURL

Будь-яка взаємодія з CouchDB заснована на REST, і це означає, що команди надсилаються лише за протоколом HTTP. CouchDB – не перша з обговорюваних нами баз даних, яка має цю якість. Riak (див. розділ 3) також використовує спілкування з клієнтами REST-интерфейс. І, як і у випадку Riak, для відправлення команд CouchDB можна застосувати утиліту cURL.

Перш ніж переходити до вистав, спробуємо кілька простих операцій CRUD. Для початку введіть у вікні терміналу таку команду:

```
$ curl http://localhost:5984/
{"couchdb":"Welcome","version":"1.1.1"}
```

У відповідь на GET-запити (cURL надсилає запити таким методом, якщо явно не вказано інше) CouchDB отримує інформацію про об'єкт, вказаний в URL. При зверненні до кореня сайту, як у даному випадку, CouchDB просто інформує про те, що працює, та повідомляє номер встановленої версії. Тепер отримаємо відомості про створену раніше базу даних music (для зручності читання результат переформатовано):

```
$ curl http://localhost:5984/music/
{
"db_name":"music", "doc_count":1,
```

```
"doc_del_count":0, "update_seq":4, "purge_seq":0,
"compact_running":false, "disk_size":16473,
       "instance_start_time":"1326845777510067", "disk_format_version":5,
"committed_update_seq":4
       }
```

У відповідь повертається інформація про те, скільки в базі даних документів, як довго працює сервер та скільки операцій він уже виконав.

Читання документа за допомогою GET

Для пошуку конкретного документа вкажіть URL його _id після імені бази даних:

```
       $ curl http://localhost:5984/music/ 74c7a8d2a8548c8b97da748f43000ac4
       {
       "_id":"74c7a8d2a8548c8b97da748f43000ac4", "_rev":"4-93a101178ba65f61ed39e60d70c9fd97",


       "name":"The Beatles", "albums": [
       {
       "title":"Help!", "year":1965
       }, {
       "title":"Sgt. Pepper's Lonely Hearts Club Band", "year":1967
       }, {
       "title":"Abbey Road", "year":1969
       }
       ]
       }
```

У CouchDB виконання GET-запитів завжди безпечне, оскільки жодних змін до документа не вноситься. Якщо потрібно змінити щось, необхідно використовувати інші дієслова HTTP, а саме: PUT, POST і DELETE.

Створення документа за допомогою POST

Для створення нового документа використовується дієслово POST. Не забудьте увімкнути заголовок Content-Type, вказавши в ньому значення application/json, інакше CouchDB відмовиться виконувати запит.

```
       $ curl -i -X POST "http://localhost:5984/music/" \
       -H "Content-Type: application/json" \
       -d '{ "name": "Wings" }' HTTP/1.1 201 Created
       Server: CouchDB/1.1.1 (Erlang OTP/R14B03) Місцезнаходження:
http://localhost:5984/music/ 74c7a8d2a8548c8b97da748f43000f1b
       Date: Wed, 18 Jan 2012 00:37:51 GMT
       Content-Type: text/plain;charset=utf-8 Content-Length: 95
       Cache-Control: must-revalidate

       {
       "ok":true, "id":"74c7a8d2a8548c8b97da748f43000f1b", "rev":"1-
2fe1dd1911153eb9df8460747dfe75a0"
       }
```

Код HTTP-відповіді `201Created` означає,що створення завершилося успішно. У тілі відповіді знаходиться JSON об'єкт із корисною інформацією, зокрема, значення _id та _rev.

Оновлення документа за допомогою PUT

Ключове слово `PUT` застосовується для оновлення існуючого документа або створення нового із явно заданим значенням _id. Як і у випадку GET, URL-адреса для PUT повинна містити ім'я бази даних, за яким слідує _id документа.

```
$ curl -i -X PUT \ "http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b" \
-H "Content-Type: application/json" \
-d' {
"_id": "74c7a8d2a8548c8b97da748f43000f1b", "_rev": "1-
2fe1dd1911153eb9df8460747dfe75a0", "name": "Wings",
"albums": ["Wild Life", "Band on the Run", "London Town"]
}'
HTTP/1.1 201 Created
Server: CouchDB/1.1.1 (Erlang OTP/R14B03) Location:
http://localhost:5984/music/ 74c7a8d2a8548c8b97da748f43000f1b
Etag: "2-17e4ce41cd33d6a38f04a8452d5a860b" Date: Wed, 18 Jan 2012
00:43:39 GMT
Content-Type: text/plain;charset=utf-8 Content-Length: 95
Cache-Control: must-revalidate
{
"ok":true, "id":"74c7a8d2a8548c8b97da748f43000f1b", "rev":"2-
17e4ce41cd33d6a38f04a8452d5a860b"
}
```

На відміну від MongoDB, де модифікація документа виконується *на місці*, CouchDB за будь-якої зміни перезаписує документ цілком. Веб-інтерфейс Futon створюють ілюзію, ніби можна змінити єдине поле, але за лаштунками при натисканні Save все одно листується весь документ.

Як ми вже зазначали, при оновленні документа передані поля _id і _rev повинні точно відповідати збереженим, інакше операція завершиться помилкою. Щоб переконатися в цьому, спробуйте виконати ту саму операцію PUT ще раз.

```
HTTP/1.1 409 Conflict
Server: CouchDB/1.1.1 (Erlang OTP/R14B03) Date: Wed, 18 Jan 2012
00:44:12 GMT
Content-Type: text/plain;charset=utf-8 Content-Length: 58
Cache-Control: must-revalidate

{"error":"conflict","reason":"Document update conflict."}
```

Отримуємо відповідь із кодом 409 Conflict, у тілі якого знаходиться JSON-об'єкт із описом причини помилки. Таким чином, CouchDB забезпечує узгодженість даних.

Вилучення документа за допомогою DELETE

Нарешті, видалення документа з бази служить дієслово DELETE.

```
$ curl -i -X DELETE \ "http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b" \
```

```
        -H "If-Match: 2-17e4ce41cd33d6a38f04a8452d5a860b" HTTP/1.1 200 OK
        Server: CouchDB/1.1.1 (Erlang OTP/R14B03) Etag: "3-
42aafb7411c092614ce7c9f4ab79dc8b" Date: Wed, 18 Jan 2012 00:45:36 GMT
        Content-Type: text/plain;charset=utf-8 Content-Length: 95
        Cache-Control: must-revalidate

        {
        "ok":true, "id":"74c7a8d2a8548c8b97da748f43000f1b", "rev":"3-
42aafb7411c092614ce7c9f4ab79dc8b"
        }
```

Операція DELETE повертає новий номер редакції, незважаючи на те, що документа вже немає. Варто зазначити, що насправді документ не видаляється з диска, а додається новий документ, в якому стоїть позначка видалення старого. Як і у разі оновлення, CouchDB не змінює документи на місці. Але для всіх практичних цілей документ можна вважати видаленим.

Тепер, навчившись виконувати прості операції CRUD в інтерфейс Futon і за допомогою cURL, ми готові перейти до більш складних тем. Зокрема створенням індексованих представлень, які відкривають інші шляхи отримання документів – не лише за значенням _id.

Реалізація представлень

У CouchDB *представлення (View)* можна розглядати як вікно в множину документів, що зберігаються в базі. Представлення – це основний спосіб доступу до документів, якщо не брати до уваги тривіальних випадків, наприклад, окремих операцій CRUD. Ми навчимося виконувати довільні запити до представлень, застосовуючи cURL. Нарешті, ми імпортуємо дані про музикантів, щоб було з чим працювати, і продемонструємо використання популярної написаної на Ruby бібліотеки couchrest для доступу до CouchDB.

Представлення складається з функцій розподілу та редукції, які використовуються для генерації впорядкованого списку пар ключ-значення. Як ключі, так і значення можуть бути довільними об'єктами JSON. Найпростіше представлення називається _all_docs. Воно автоматично надається для будь-якої бази даних і містить по одному запису для кожного документа, що зберігається, причому ключем є рядковий ідентифікатор документа _id.

Щоб отримати всі документи з бази, виконайте GET-запит до представлення _all_docs.

```
$ curl http://localhost:5984/music/_all_docs
{
"total_rows":1, "offset":0,
“rows”:[{ “id”:”74c7a8d2a8548c8b97da748f43000ac4”,
“key”:”74c7a8d2a8548c8b97da748f43000ac4”, “value”:{
“rev”:”4-93a101178ba65f61ed39e60d70c9fd97”
}
}]
}
```

Як бачите, у відповідь повернено той єдиний документ, який ми поки що створили. Результат представлений як JSON-об'єкта, що містить масив рядків rows. Кожен рядок – це об'єкт із трьома полями:

- id – ідентифікатор документа _id;
- key – ключ (JSON-об'єкт), породжений mapreduce-функціями;
- value - асоційоване з ключем значення (JSON-об'єкт), що також породжується mapreduce-функціями.

У випадку _all_docs поля id і key збігаються, але для власних представлень так майже ніколи не буває.

За замовчуванням представлення не включає у повернене значення значення весь вміст документа. Щоб вибрати всі поля, додайте URL параметр include_docs=true.

```
$ curl http://localhost:5984/music/_all_docs?include_docs=true
{
"total_rows":1, "offset":0,
“rows”:[{ “id”:”74c7a8d2a8548c8b97da748f43000ac4”,
“key”:”74c7a8d2a8548c8b97da748f43000ac4”, “value”:{
“rev”:”4-93a101178ba65f61ed39e60d70c9fd97” “name”:”The Beatles”,
"albums":[{
“title”:”Help!”, “year”:1965
}, {
“title”:”Sgt. Pepper's Lonely Hearts Club Band”, “year”:1967
}, {
“title”:”Abbey Road”, “year”:1969
}]
}
}]
}
```

Як бачите, тепер до об'єкта value включені додаткові властивості name та albums. Пам'ятаючи про цю базову структуру, приступимо до створення власних представлень .

Створення першого представлення

Розуміючи загалом, як працюють представлення, спробуємо створити своє власне. Для початку відтворимо поведінку представлення _all_docs, а потім ускладнимо завдання і будемо витягувати з документів дані,

розташовані на більш глибоких рівнях, для індексування.

Щоб виконати тимчасову представлення, відкрийте у браузері інтерфейс Futon, як ми робили раніше. Потім відкрийте базу даних music, клацнувши відповідне посилання. З розкривного списку View у верхньому правому кутку сторінки виберіть пункт «Temporary view...» (мал. 28).

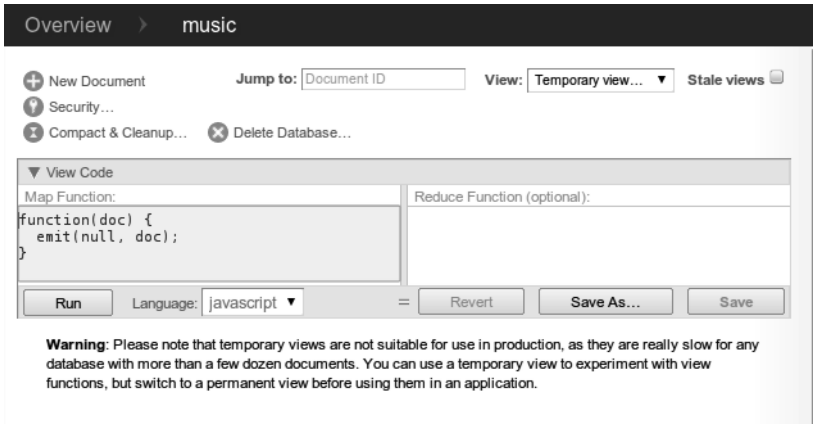


Рис. 28. CouchDB Futon: часове представлення

У лівій області Map Function буде присутній такий код:

```
function(doc) {emet(null, doc);
}
```

Якщо зараз натиснути кнопку під цією областю Run, то CouchDB виконає цю функцію по одному разу для кожного документа у базі, передаючи поточний документ у параметрі doc. В результаті буде згенеровано таблицю, що містить по одному рядку для кожного документа:

Ключ Значення

```
null {_id:"74c7a8d2a8548c8b97da748f43000ac4", _rev:"4-
93a101178ba65f61ed39e60d70c9fd97",name: "The Beatles", albums: [{title: "Help!",
year: 1965},
{title:"Sgt. Pepper's Lonely Hearts Club Band", year: 1967}, {title: "Abbey
Road", year: 1969}]}
```

Секрет полягає у функції emit() (яка працює аналогічно однойменній функції MongoDB). Ця функція приймає два аргументи: ключ та значення. Функція розподілу може викликати emit для одного документа нуль один або багато разів. У разі розподільник емітує пару null/doc. Як видно з таблиці, ключ дійсно дорівнює null, а значення – той самий об'єкт, який ми бачили вчора, коли запитували документ за допомогою cURL.

Але щоб досягти того ж результату, що _all_docs, розподільник має емітувати щось інше. Нагадаємо, що _all_docs емітує поле _id документа як

ключ і простий об'єкт, що містить лише поле _rev – як значення. Тому змініть код у області Map Function, як показано нижче, та натисніть кнопку Run.

```
function(doc) {
emit(doc._id, { rev: doc._rev });
}
```

Тепер буде повернуто таблицю з тією ж парою ключ-значення, яку ми бачили раніше, коли перебирали записи за допомогою `_all_docs`:

Ключ **Значення**

```
“74c7a8d2a8548c8b97da748f43 000ac4” {rev:“4-93a101178ba65f61ed 39e60d70c9fd97”}
```

Зазначимо, що для виконання часових представлень Futon не є обов'язковим. Можна натомість відправити POST-запит обробникові _temp_view, а в тілі запиту передати функцію-розподільник у вигляді об'єкта JSON.

```
$ curl -X POST \ http://localhost:5984/music/_temp_view \

-H “Content-Type: application/json” \
-d '{“map”:”function(doc){emit(doc._id,{rev:doc._rev});}”}'
{
"total_rows":1, "offset":0,
“rows”:[{ “id”:”74c7a8d2a8548c8b97da748f43000ac4”,
“key”:”74c7a8d2a8548c8b97da748f43000ac4”, “value”:{
“rev”:”4-93a101178ba65f61ed39e60d70c9fd97”
}
            }]
}
```

Відповідь така ж, як при використанні _all_docs. Тепер додамо параметр `include_docs=true`

```
$ curl -X POST \ http://localhost:5984/music/_temp_view?include_docs=true \
-H “Content-Type: application/json” \
-d '{“map”:”function(doc){emit(doc._id,{rev:doc._rev});}”}'
{
"total_rows":1, "offset":0,
“rows”:[{ “id”:”74c7a8d2a8548c8b97da748f43000ac4”,
“key”:”74c7a8d2a8548c8b97da748f43000ac4”, “value”:{
“rev”:”4-93a101178ba65f61ed39e60d70c9fd97”
},
“doc”:{ “_id”:”74c7a8d2a8548c8b97da748f43000ac4”, “_rev”:”4-
93a101178ba65f61ed39e60d70c9fd97”, “name”:”The Beatles”,
“albums”:[...]
}
            }]
}
```

На цей раз додаткові поля не включаються до об'єкта value, а у рядок додається властивість doc, в якому міститься весь документ. Представлення може емітувати будь-яке значення, у тому числі null. Створення окремої властивості doc запобігає проблемам, які можуть виникнути в результаті об'єднання value з документом. Далі ми покажемо, як зберегти

представлення, щоб CouchDB могла проіндексувати результати.

Збереження представлення у вигляді проєктного документа

Виконуючи часове представлення, CouchDB має застосувати переданий розподільник до кожного документа в базі даних. Ця процедура потребує величезної кількості ресурсів, споживає багато процесорного часу та взагалі працює повільно. Тимчасові представлення краще використовувати лише на етапі розробки. У виробничій системі представлення слід зберігати як проєктних документів (design documents).

Щоб зберегти часове представлення у вигляді проєктного документа у Futon, натисніть кнопку Save As... (Зберегти як), а потім заповніть поля Design Document та View Name (Ім'я представлення).

Ідентифікатори проєктних документів завжди починаються рядком _design/. Один проєктний документ може містити одне або кілька подань. Подання, що зберігаються в одному проєктному документі, розрізняються за іменами. Яке представлення в який проєктний документ поміщати залежить від додатка, і рішення віддається на відкуп розробнику. Взагалі, представлення рекомендується групувати відповідно до того, як вони співвідносяться з даними. Приклади будуть наведені нижче, коли ми почнемо розглядати цікавіші представлення.

Пошук виконавців за ім'ям

Отже, з принципами створення представлень ми познайомилися, тепер розробимо представлення для конкретного додатка. Нагадаємо, що в базі даних music зберігається інформація про музикантів, у тому числі назву гурту в полі name. Звернувшись до представлення _all_docs із звичайним GET-запитом, ми зможемо отримати документ за значенням _id, але нас більше цікавить пошук груп за назвою.

Іншими словами, зараз ми вміємо знаходити документз _id, рівним 74c7a8d2a8548c8b97da748f43000ac4, але як знайти документ, у якого в полі name зберігається рядок The Beatles? Для цього потрібне представлення. У

Futon поверніться на сторінку Temporary View, введіть у Map Function наведений нижче код і натисніть кнопку Run.

```
couchdb/artists_by_name_mapper.js function(doc) {
if ( 'name' in doc) {emit(doc.name, doc._id);
}
}
```

Ця функція перевіряє, чи є у поточному документі поле `name` і, якщо так, емітує його значення та ідентифікатор _id у вигляді пари ключ-значення. В результаті вийде таблиця виду:

Ключ	Значення
"TheBeatles"	"74c7a8d2a8548c8b97da748f43000ac4"

Натисніть кнопку Save As... і введіть у поле Design Document значення artists, а поле View Name – значення by_name. Натисніть Save, щоб зберегти зміни.

Пошук альбомів за назвою

Вміння шукати виконавців за назвою – річ корисна, але навіщо на цьому зупинятись? Давайте ще напишемо представлення для пошуку альбомів. Це перший приклад, коли функція-розподільник емітує кілька результатів одного документа.

Знову перейдіть на сторінку Temporary View та введіть такий код розподільника:

couchdb/albums_by_name_mapper.js

```
function(doc) {
if ('name' in doc && 'albums' in doc) { doc.albums.forEach(function(album){
var
key = album.title || album.name,
value = { by: doc.name, album: album }; emit(key, value);
});
}
}
```

Ця функція перевіряє, чи є у поточному документі поля name та albums. Якщо так, то вона емітує пару ключ-значення для кожного альбому, причому ключем є назва альбому, а значенням – складовий об'єкт, що містить ім'я виконавця (або назву групи) та об'єкт, що представляє сам альбом. Виходить така таблиця:

Ключ **Значення**

"AbbeyRoad" {by:"The Beatles", album: {title:"AbbeyRoad", year: 1969}}

"Help!" {by:"The Beatles", album: {title: "Help!", year: 1965}}

"Sgt.Pepper's Lonely Hearts Club Band" {by:The Beatles, album: {title: "Sgt. Pepper's Lonely Hearts Club Band",year: 1967}}

Як і раніше, натисніть кнопку Save As..., тільки тепер у полі Design Document введіть albums, а поле View Name – by_name. Натисніть Save, щоб зберегти зміни. Тепер побачимо, як опитувати такі документи.

Опитування представлень виконавців та альбомів

Тепер, коли ми маємо два проєктні документи, повернемося у вікно терміналу і спробуємо запросити їх за допомогою curl. Почнемо з вистави виконавців за іменами. Введіть таку команду:

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name
{
"total_rows":1, "offset":0,
"rows":[{ "id":"74c7a8d2a8548c8b97da748f43000ac4", "key":"The Beatles", "value":"74c7a8d2a8548c8b97da748f43000ac4"
}]
}
```

URL для опитування представлення має вигляд `path/<database_name>/_design/<design_doc>/_view/<view_name>`, де замість рядків у кутових дужках підставляються імена проєктного документа та подання. В даному випадку ми опитуємо представлення by_name у проєктному документі artists, який зберігається в базі даних music. Не дивно, що в результаті повертається єдиний документ із ключем, який містить назву групи.

А тепер спробуємо опитати представлення альбомів за назвою:

```
$ curl http://localhost:5984/music/_design/albums/_view/by_name
{
"total_rows":3,
"offset":0,
"rows":[{ "id":"74c7a8d2a8548c8b97da748f43000ac4", "key":"Abbey Road",
            "value": {
"by":"The Beatles", "album":{
"title":"Abbey Road", "year":1969
}
}
}, {
"id":"74c7a8d2a8548c8b97da748f43000ac4", "key":"Help!",
"value": {
"by":"The Beatles", "album":{
"title":"Help!", "year":1965
}
}
}, {
"id":"74c7a8d2a8548c8b97da748f43000ac4", "key":"Sgt. Pepper's Lonely Hearts
```

```
Club Band", "value":{
        "by":"The Beatles", "album":{
        "title":"Sgt. Pepper's Lonely Hearts Club Band", "year":1967
        }
                    }
                   }]
        }
```

CouchDB гарантує, що записи відсортовані в алфавітно-цифровому порядку за емітованими ключами. Фактично, це результат індексування, виконаного CouchDB. При проєктуванні представлень важливо вибирати ключі, що емітуються, так щоб порядок сортування був осмисленим. Показаний вище запит повертає весь вміст представлення, але що, якщо нам потрібне лише його підмножина? Домогтися цього можна, наприклад, додавши URL параметр key. У цьому випадку повертаються лише рядки із заданим ключем.

```
$ curl 'http://localhost:5984/music/_design/albums/_view/by_name? \ key="Help!"'
{
"total_rows":3, "offset":1,
"rows":[{ "id":"74c7a8d2a8548c8b97da748f43000ac4", "key":"Help!",
"value": {
"by":"The Beatles", "album":{"title":"Help!","year":1965}
}
                    }]
}
```

Зверніть увагу на поля `total_rows` і `offset`. В полі `total_rows` знаходиться загальна кількість записів у поданні (не тільки тих, що повернуто у відповідь на запит). Поле offset показує усунення першого поверненого запису з початку повного набору. Знаючи ці два числа та розмір масиву rows, ми можемо обчислити, скільки записів залишилося зліва та праворуч від обраної порції. Вибірку з вистав можна здійснювати не тільки за параметром key, але для демонстрації нам знадобиться більше даних.

Імпорт даних у CouchDB за допомогою програми на Ruby

Імпорт даних – завдання, яке незмінно постає незалежно від використовуваної СУБД. І CouchDB – не виняток. У цій лекції ми напишемо на Ruby програму імпорту структурованих даних до бази music. Це, по-перше, дасть можливість познайомитися з масовим імпортом у CouchDB, а, по-друге, ми отримаємо досить великий набір даних вивчення більш складних представлень .

Ми скористаємося даними про музичні твори із сайту Jamendo.com, на якому розміщується музика з безкоштовною ліцензією http://www.jamendo.com. Jamendo представляє дані про виконавців, альбомів та композицій у форматі XML, ідеальному для імпорту в документоорієнтовану базу типу CouchDB.

Перейдіть на сторінку NewDatabaseDumps (http://developer.jamendo.com/en/wiki/NewDatabaseDumps) сайту Jamendo та завантажте файл `dbdump_artistalbumtrack.xml.gz` (http://img.jamendo.com/data/dbdump_artistalbumtrack.xml.gz). Цей архів «важить» всього близько 15МБ. Для розбору XML-файлами скористаємосяgem-пакетом libxml-ruby.

Замість того, щоб писати власний драйвер Ruby-CouchDB або безпосередньо формувати HTTP-запити, ми вдамося до популярного gem-пакету couchrest, який обгортає виклики зручним API на Ruby. Нам знадобиться лише кілька методів цього API, але до послуг бажаючих використовувати драйвер у проєктах є хороша документация (наприклад http://rdoc.info/github/couchrest/couchrest/master/).

Уставте необхідні gem-пакети, виконавши таку команду:

```
$ gem install libxml-ruby couchrest
```

Як і при розборі даних з вікіпедії в розділі 4, ми будемо використовувати SAX-аналізатор, який послідовно читає та вставляє документи в базу, начебто вони надходили зі стандартного введення. Код програми наведено нижче.

couchdb/import_from_jamendo.rb

```
   require 'rubygems' require 'libxml' require
'couchrest'

            include LibXML

   class JamendoCallbacks
            include XML::SaxParser::Callbacks
  def initialize()
@db = CouchRest.database!("http://localhost:5984/music")
@count = 0
@max = 100 # максимальна кількість документів, що вставляються
@stack = []
@artist = nil
@album = nil
@track = nil
```

```
        @tag = nil
        @buffer = nil end
          def on_start_element(element, attributes) case element
        коли 'artist'
        @artist = { :albums => [] }
        @stack.push @artist when 'album'
        @album = { :tracks => [] }
        @artist[:albums].push @album
        @stack.push @album when 'track'
        @track = { :tags => [] }
        @album[:tracks].push @track
        @stack.push @track when 'tag'
        @tag = {}
        @track[:tags].push @tag
        @stack.push @tag
        коли 'Artists', 'Albums', 'Tracks', 'Tags'
        # ігнорувати else
        @buffer = [] end
        end
          def on_characters(chars)
        @buffer << chars unless @buffer.nil? end
        def on_end_element(element) case element
        коли 'artist'
        @stack.pop
        @artist['_id'] = @artist['id'] # використовуємо ідентифікатор,
        # привласнений Jamendo
        # Виконавцю, як
        #_id документа
        @artist[:random] = rand
        @db.save_doc(@artist, false, true)
        @count += 1
        if !@max.nil ? && @count >= @max on_end_document
        end
        if @count % 500 == 0
        puts " #{@count} records inserted" end
        when 'album', 'track', 'tag' top = @stack.pop
top[:random] = rand
        коли 'Artists', 'Albums', 'Tracks', 'Tags'
        # ігнорувати else
        if @stack[-1] && @buffer
        @stack[-1][element] = @buffer.join.force_encoding('utf-8')
        @buffer = nil end
        end end

        def on_end_document()
        puts "TOTAL: #{@count} records inserted" exit(1)

        end end
           parser = XML::SaxParser.io(ARGF) parser.callbacks =
JamendoCallbacks.new parser.parse
```

Насамперед ми імпортуємо модуль rubygems і необхідні нам gem-пакети.

Стандартний спосіб роботи з пакетом LibXML передбачає створення класу зворотного дзвінка. Тут ми визначаємо клас JamendoCallbacks, який інкапсулює обробники різних подій SAX.

На етапі ініціалізації наш клас спочатку підключається до локального сервера CouchDB, використовуючи CouchRest API, а потім створює базу данихmusic(якщо її ще немає). Потім ініціалізуються різні

змінні екземпляри для зберігання інформації про стан аналізу. Зазначимо, що якщо присвоїти параметру`@max`значення`nil`, то будуть імпортовані всі документи, а не лише перші 100.

Під час розбирання методу`on_start_element()`буде обробляти всі теги, що відкривають. Нас цікавлять лише теги`<artist>`,`<album>`,

`<track>`і`<tag>`. Деякі контейнерні елементи`<Artists>`,

`<Albums>`,`<Tracks>`і`<Tags>`– ми явно пропускаємо, проте інші трактуємо як властивості, встановлювані для найближчого контейнера.

Розпізнані аналізатором символьні дані ми буферизуємо і згодом додаємо як властивість поточного контейнерного елемента (наприкінці масиву`@stack`).

Найцікавіше відбувається у методі `on_end_element()`. Тут ми закриваємо поточний контейнерний елемент, виштовхуючи його зі стека. Якщо виявлено закриваючий тег для елемента`<artist>`, то ми зберігаємо документ у CouchDB, викликаючи метод`@db.save_doc()`. Крім того, для будь-якого контейнерного елемента додається властивість`random`, Що містить згенероване випадкове число Згодом ми скористаємося ним для вибору випадкової композиції, альбому чи виконавця.

У потоці ARGF Ruby поєднує стандартне введення та всі файли, вказані в командному рядку. Ми передаємо цей потік пакету LibXML і вказуємо екземпляр класу JamendoCallbacks, який оброблятиме розпізнані лексеми: теги, що відкривають і закривають, а також символьні дані.

При запуску скрипту імпорту передаємо йому конвеєр результат розпакування архіву з XML-файлом:

```
$zcat dbdump_artistalbumtrack.xml.gz | ruby import_from_jamendo.rb TOTAL: 100 records inserted
```

Після імпорту подивимося, як тепер виглядають наші представлення. Спочатку відберемо кілька виконавців. Параметр `limit` в URL каже, що хочемо отримати трохи більше зазначеної у ньому кількості документів.

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?limit=5
{"total_rows":100,"offset":0,"rows":[
```

```
{"id":"370255","key":"\"\"ATTIC\"\"","value":"370255"},
{"id":"353262","key":"10centSunday","value":"353262"},
{"id":"367150","key":"abdielyromero","value":"367150"},
{"id":"276","key":"AdHoc","value":"276"},
{"id":"364713","key":"Adversus","value":"364713"}
]}
```

Цей запит починає відбір із початку списку виконавців. Щоб почати з середини, потрібно встановити параметр startkey:

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?\
limit=5\&startkey=%22C%22
{"total_rows":100,"offset":16,"rows":[
{"id":"340296","key":"CalexB","value":"340296"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"},
{"id":"364714","key":"Daringer","value":"364714"}
]}
```

Вище ми розпочали з виконавців, імена яких починаються з літери C. Параметр endkey дає ще один спосіб обмежити вибірку. Нижче ми запитуємо виконавців з іменами, що починаються з літер C та D:

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?\
startkey=%22C%22\&endkey=%22D%22
{"total_rows":100,"offset":16,"rows":[
{"id":"340296","key":"CalexB","value":"340296"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"}
]}
```

Щоб відсортувати рядки у зворотному порядку, скористаємося параметром descending. Не забудьте лише поміняти місцями startkey та endkey.

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?\
startkey=%22D%22\&endkey=%22C%22\&descending=true
{"total_rows":100,"offset":16,"rows":[
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"340296","key":"CalexB","value":"340296"}
]}
```

Є ще ряд параметрів, що задаються в URL, які модифікують запити до представлень , але описані найбільш уживані. Деякі параметри стосуються угруповання, що є частиною фази редукції CouchDB.

Ми навчилися створювати у CouchDB прості представлення та зберігати їх у вигляді проєктних документів. Ми ознайомилися з різними способами опитування представлень для отримання вибірки індексованого вмісту. Скориставшись мовою Ruby та популярним gem-пакетом couchrest, ми імпортували структуровані дані та на їх основі побудували

представлення.

Висновок

Прагнення CouchDB забезпечити надійність за умов невизначеності робить її вдалою системою для протистояння жорстокої реальності Інтернету. Маючи стандартні веб-технології, зокрема HTTP/REST і JSON, CouchDB відмінно вбудовується в будь-яке рішення, де такі технології активно застосовуються, тобто – з урахуванням сучасних тенденцій – скрізь.

У CouchDB є чимало інших особливостей, що роблять її унікальною і заслуговує на увагу як база даних. На жаль, ми не можемо розглянути їх усі, але хоча б перерахуємо деякі: простота резервного копіювання, двійкові вкладення в документи та CouchApps – система для розробки та розгортання веб-програми прямо через CouchDB без додаткового ПЗ проміжного шару.

Завдання для індивідуальної роботи

1. За допомогою cURL надішліть PUT-запит, який помістить убазу даних music новий документ з конкретним _id на ваш вибір.
2. За допомогою cURL створіть нову базу даних, а потім – також за допомогою cURL – видаліть її.
3. Знову ж таки за допомогою cURL створіть новий документ, який міститиме текстовий документ, заданий у вигляді вкладення. І, нарешті, відправте (звісно, за допомогою cURL) запит, який поверне цей документ-вкладення.
4. Ми бачили, що метод emit() може виводити рядкові ключі. А які типи він підтримує? Що станеться, якщо емітувати масив як ключ?
5. Знайдіть опис переліку параметрів, що підтримуються в URL (типу limit і startkey) і з'ясуйте, для чого вони призначені.
6. Скрипт import_from_jamendo.rb зіставив кожному виконавцю випадкове число, записавши його як random. Напишіть функцію-розподільник, яка емітує пари ключ-значення, де ключем є це випадкове число, а значенням – назва групи. Збережіть її у новому проєктному документі _design/random під ім'ям artist.

7. Сформулюйте cURL запит, який повертатиме випадкового виконавця. Потрібно буде використовувати параметрstartkey і згенерувати у командному рядку випадкове число за допомогою команди `ruby -e 'puts rand'`.

8. Скрипт імпорту сформував також властивість random для кожного альбому, композиції та тега. Створіть у проєктному документі _design/random ще три представлення з іменами album, track і tag за зразком раніше створеного представлення artist.

Лекція 11. Редукція, Changes API й реплікація даних за допомогою CouchDB

Ми навчилися виконувати прості операції CRUD та за допомогою представлень шукати дані. Сьогодні ми продовжимо цю тему та розберемося з фазою редукції у технології mapreduce. Потім ми розробимо кілька програм на JavaScript для сервера Node.js, щоб вивчити особливості API оновлення, що надається CouchDB. І насамкінець обговоримо реплікацію та механізм вирішення конфліктів.

Створення складніших представлень за допомогою редукторів

Засновані на технології mapreduce представлення дозволяють скористатися вбудованими в CouchDB засобами індексування та агрегування. Усі створені вчора представлення містили лише розподільники. А сьогодні ми підключимо редуктори та застосуємо нові знання до даних, імпортованих із сайту Jamendo.

Дані з сайту Jamendo мають чудову особливість – глибоку вкладеність. У виконавців є альбоми, альбоми – композиції, а композиції – атрибути, у тому числі теги. Зараз ми займемося тегами і подивимося, як написати представлення для їхньої вибірки та підрахунку.

Поверніться на сторінку Temporary View та введіть наступну функцію розподільника.

couchdb/tags_by_name_mapper.js

```
function(doc) {
(doc.albums || []).forEach(function(album){ (album.tracks ||
[]).forEach(function(track){
(track.tags || []).forEach(function(tag){ emit(tag.idstr, 1);
});
});
});
}
```

Ця функція заходить всередину документа, що описує виконавця, потім усередину кожного альбому, кожної композиції та, нарешті, кожного тега. Для кожного тега вона емітує пару ключ-значення, що складається з властивості idstr тега (яке містить його рядкове представлення, наприклад

rock) і числа 1.

Після цього введіть наступний код до області Reduce Function:

couchdb/simple_count_reducer.js

```
function(key, values, rereduce) { return sum(values);
}
```

Ця функція просто підсумовує всі числа у списку values, про який ми поговоримо трохи нижче – після запуску цього представлення. Натисніть Run. В результаті буде виведено таку таблицю:

Ключ	Значення
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"acid"	1
"acousticguitar"	1
"acousticguitar"	1
"action"	1
"action"	1

Нічого дивного. Значення всюди одно 1, оскільки його сформував розподільник, а той самий ключ повторюється стільки разів, скільки зустрічається в композиціях. Однак зверніть увагу на прапорець Reduce у верхньому правому кутку виведеної таблиці. Позначте його та знову подивіться на таблицю. Тепер вона виглядає так:

Ключ	Значення
"17sonsrecords"	5
"acid"	1
"acousticguitar"	2
"action"	2
"adventure"	3
"aksband"	1
"alternativ"	1
"alternativ"	3
"ambient"	28
"Autodidacta"	17

Що сталося? Якщо коротко, то редуктор редукував результат, поєднавши однакові рядки так, як диктує функція редукції. Механізм mapreduce у CouchDB концептуально влаштований так само, як в інших розглянутих нами базах даних (Riak та MongoDB). Точніше CouchDB

виконує наступні кроки при побудові представлення.

1. Надіслати документи функції-розподільнику.
2. Зібрати всі емітовані значення.
3. Відсортувати емітовані рядки за ключами.
4. Надіслати групи рядків з однаковими ключами функції редактору.
5. Якщо даних занадто багато для редукування за один виклик, викликати редуктор ще раз, але вже з раніше редукованими значеннями.
6. Повторювати рекурсивні виклики редуктора стільки разів, скільки необхідно, доки не залишиться ключів, що повторюються.

Функції-редуктори CouchDB приймають три аргументи: key, values і rereduce. У аргументі key передається масив кортежів; кожен кортеж – це масив із двох елементів: ключа, емітованого розподільником, і поля _id того документа, яким цей ключ породжений. В аргументі значення передається масив значень, що відповідають ключам.

Третій аргумент, `rereduce` - булевське значення, що дорівнює true, якщо даний виклик є *повторною редукцією*,тобто ключі і значення надійшли не від розподільника, а в результаті попередніх викликів редукторів. У такому разі параметр key дорівнюватиме `null`.

Відстеження змін у CouchDB

Інкрементний підхід CouchDB до mapreduce – безумовно, новаторська ідея, одна з багатьох особливостей, що виділяють CouchDB серед інших СУБД. Далі ми розглянемо ще одну: Changes API – інтерфейс, що дозволяє відстежувати зміни у базі даних та миттєво отримувати нові значення.

Changes API робить CouchDB ідеальним кандидатом на роль системи, куди записуються оригінальні дані. Уявіть собі систему з кількох СУБД, в яку дані стікаються з різних джерел, причому всі бази повинні підтримуватися в актуальному стані (щось подібне ми побудуємо в розділі 8.4 «Комбінування з іншими базами даних»). Як приклад можна назвати

пошукову систему на основі Lucene або ElasticSeach або шар кешування, реалізований за допомогою memcached або Redis. Крім того, у відповідь на зміни можуть запускатись різні адміністративні скрипти – наприклад, для стиснення бази або віддаленого резервного копіювання. Коротше кажучи, цей простий API відкриває багато можливостей. Сьогодні ми навчимося використовувати його собі на благо. Для демонстрації API ми розробимо кілька простих клієнтських програм, використовуючи Node. js - платформу для виконання JavaScript-коду на стороні сервера, побудовану на основу движка JavaScript V8 - того самого, який вбудований у браузер Google Chrome. Оскільки в Node.js застосовується подієво-керована модель, а код пишеться на JavaScript, він цілком природно інтегрується з CouchDB. Якщо на вашій машині Node.js ще не встановлено, зайдіть на сайт продукту та скачайте останню стабільну версію (ми працюємо з версією 0.6).

Існує три різновиди Changes API: опитування, тривале опитування та безперервне відстеження. Розглянемо їх по черзі. Як зазвичай, почнемо з cURL, щоб бути «ближчим до заліза», а потім поговоримо про доступ із програми.

Найпростіший спосіб звернутись до Changes API – скористатися інтерфейсом опитування. Виконайте наступну команду (ми зменшили висновок, на вашій машині результати можуть бути іншими).

```
$ curl http://localhost:5984/music/_changes
{
“results”:[{
“seq”:1, “id”:”370255”,
“changes”:[{“rev”:”1-a7b7cc38d4130f0a5f3eae5d2c963d85”}]
}, {
“seq”:2, “id”:”370254”,
“changes”:[{“rev”:”1-2c7e0deec3ffca959ba0169b0e8bfcef”}]
}, {
... ще 97 записів ...
}, {
“seq”:100, “id”:”357995”,
“changes”:[{“rev”:”1-aa649aa53f2858cb609684320c235aee”}]
}],
"last_seq":100
}
```

У відповідь на GET-запит до URL _changes без параметрів CouchDB повертає все, що має. Але, як і при доступі до представлень, можна встановити параметр limit, щоб отримати підмножина даних, а параметр

include_docs=true змушує повернути повні документи.

Як правило, нас цікавлять лише ті зміни, які відбулися з моменту останньої перевірки. Для цього призначено параметр since.

```
$ curl http://localhost:5984/music/_changes?since=99
{
“results”:[{
“seq”:100, “id”:”357995”,
“changes”:[{“rev”:”1-aa649aa53f2858cb609684320c235aee”}]
}],
"last_seq":100
}
```

Якщо значення since більше за останній порядковий номер, то буде повернена порожня відповідь:

```
$ curl http://localhost:5984/music/_changes?since=9000
{
“results”:[
],
"last_seq":9000
}
```

Таким чином клієнтський додаток може періодично вимагати нові зміни та вживати відповідних дій.

Опитування– річ хороша, якщо програма може змиритися із запізнювальною реакцією на оновлення, наприклад, у випадку, коли оновлення відбуваються порівняно рідко. Так, опитування нових записів у блозі раз на п'ять хвилин може виявитися цілком прийнятним рішенням.

Якщо ж отримувати оновлення потрібно швидше, не витрачаючи час на повторне відкриття з'єднання, краще підійде довге опитування (long polling). Якщо додати до URL параметр feed=longpoll, CouchDB залишить з'єднання відкритим на деякий час, очікуючи, що можуть відбутися якісь зміни, про які слід повідомити у відповіді. Спробуйте такий запит:

```
$ curl 'http://localhost:5984/music/_changes?feed=longpoll&since=9000'
{“results”:[
```

Як бачите, сервер повернув початок JSON-відповіді та більше нічого. Якщо залишити термінал відкритим, то зрештою CouchDB закриє з'єднання та закінчить відповідь:

```
],
"last_seq":9000}
```

З точки зору розробки, написання драйвера, яке спостерігає за змінами в базі даних CouchDB за допомогою опитування, еквівалентне написанню

драйвера, що реалізує тривале опитування. Різниця лише в тому, скільки часу CouchDB триматиме з'єднання відкритим. А тепер перейдемо до написання програми для Node.js, яка стежитиме за «стрічкою» змін.

Опитування змін у програмі для Node.js

Node.js – система, керована подіями, тому і спостерігач для CouchDB дотримуватиметься того самого принципу. Драйвер буде стежити за змінами та генерувати події, отримавши від CouchDB змінені документи. Спочатку покажемо загальну структуру драйвера, обговоримо її основні компоненти, а потім заповнимо деталі, що відносяться до конкретної стрічки - потоку змін.

Ось як улаштована наша програма-спостерігач.

couchdb/watch_changes_skeleton.js

```
var
http = require('http'), events = require('events');

/**
*       створити спостерігач за CouchDB, використовуючи параметри з'єднання;
*       слід патерну EventEmitter в node.js, що генерує події 'change'.
*/
   exports.createWatcher = function(options) {

       var watcher = новий events.EventEmitter();

watcher.host = options.host | 'localhost'; watcher.port=options.port || 5984;
watcher.last_seq = options.last_seq || 0; watcher.db = options.db | '_users';

       watcher.start = function() {
// ... деталі, що стосуються стрічки оновлень
};
return watcher;
};

// почати спостереження за змінами в CouchDB, якщо запущено як
// головний скрипт
   if (!module.parent) { exports.createWatcher({
db: process.argv[2], last_seq: process.argv[3]
})
.on('change', console.log)
.on('error', console.error)
.start();
}
```

Додавання методу `createWatcher()` в об'єкт `exports` робить його доступним іншим скриптам Node.js, тобто він стає частиною бібліотеки. Аргумент`options`дозволяє програмі, що викликає, вказати, до якої бази даних підключатися і задати інші параметри з'єднання.

Метод `createWatcher()` породжує об'єкт `EventEmitter`, за

допомогою якого програма, що викликає, може прослуховувати події зміни. Метод `on()` об'єкта `EventEmitter`дозволяє підписатися на зовнішню подію та генерувати внутрішню подію шляхом виклику методу `emit()`.

Метод `watcher.start()` відповідає за надсилання HTTP-запитів для спостереження за змінами в CouchDB. Коли зміниться будь-який документ, спостерігач сповіщає про це програму шляхом генерації події зміни. Усі специфічні деталі реалізації мають бути поміщені сюди.

Цей блок визначає, що повинен робити скрипт, якщо він викликаний безпосередньо з командного рядка. У цьому випадку скрипт викличе метод `createWatcher()`, після чого налаштує повернутий об'єкт, щоб результати друкувалися на стандартний висновок. До якої бази даних підключатися і з якого порядку починати, задається в аргументах командного рядка.

Поки що в цьому коді немає нічого, що стосується CouchDB. Це просто стандартний спосіб програмування Node.js.

Тепер, додамо код для довгого опитування CouchDB та генерації подій. Наведений нижче код слід помістити в метод `watcher.start()` (туди, де знаходиться коментар «деталі, що стосуються стрічки оновлень». Результуючий файл назвіть `watch_changes_longpolling.js`.

couchdb/watch_changes_longpolling_impl.js

```
var
   http_options = { host: watcher.host, port:
watcher.port, path:
'/' + watcher.db + '/_changes' + '?feed=longpoll&include_docs=true&since=' +
watcher.last_seq
};
   http.get(http_options, function(res) { var buffer = '';
res.on('data', function (chunk) { buffer += chunk;
});
res.on('end', function() {
  var output = JSON.parse(buffer); if (output.results) {
watcher.last_seq = output.last_seq; output.results.forEach(function(change){
watcher.emit('change', change);
});

watcher.start();
} else {
watcher.emit('error', output);
}
            })
})
.on('error', function(err) { watcher.emit('error', err);
});
```

Цей скрипт налаштовує конфігураційний об'єкт `http_options`, готуючи його до надсилання запиту. Параметр `path` вказує на вже відому нам URL_ `changes`, в якому `feed` задає режим тривалого опитування, а `include_docs=true`.

Потім скрипт викликає бібліотечний метод Node.js `http.get()`, який відправляє `GET`-запит із зазначеними параметрами. Другий параметр `http.get` – функція зворотного дзвінка, яка отримає об'єкт `HTTPResponse`. Отримуючи вміст, об'єкт відповіді генерує події `data`, у відповідь на які ми додаємо отриманий вміст у буфер.

Нарешті, у відповідь на подію, що згенерувала об'єктом відповіді `end` ми розуміємо вміст буфера (це має бути JSON-об'єкт) і дізнаємося про нове значення `last_seq`. Потім ми генеруємо події зміни та повторно викликаємо `watcher.start()` в очікуванні нової порції змін.

З командного рядка цей скрипт запускається так (видача скорочена):

```
$node watch_changes_longpolling.js music
{ seq: 1,
id: '370255',
changes: [ { rev: '1-a7b7cc38d4130f0a5f3eae5d2c963d85' } ], doc:
{ _id: '370255',
_rev: '1-a7b7cc38d4130f0a5f3eae5d2c963d85', albums: [ [Object] ],
id: '370255', name: ''ATTIC'',
url: 'http://www.jamendo.com/artist/ATTIC_(3)',mbgid: '',
random: 0.4121620435325435 } }
{ seq: 2,
id: '370254',
changes: [ { rev: '1-2c7e0deec3ffca959ba0169b0e8bfcef' } ], doc:
{ _id: '370254',
_rev: '1-2c7e0deec3ffca959ba0169b0e8bfcef',
... ще 98 записів ...
```

Після виведення записів для всіхдокументів, процес не завершується, а продовжує опитувати CouchDB про нові зміни.

Спробуйте модифікувати якийсь документ у Futon безпосередньо або збільште значення @max у скрипті `import_from_jamendo.rb` і запустіть його ще раз. Ви побачите, що зміни надрукуються у вікні терміналу.

Реплікація даних у CouchDB

CouchDB замислювалася насамперед заради асинхронної обробки

даних та забезпечення їх довговічності. Згідно з ідеологією CouchDB, найбезпечніше місце зберігання даних – усюди, і вона надає необхідні інструменти. У інших СУБД для гарантії узгодженості виділяється один головний вузол. В інших для цієї мети використовується кворум приголосних між собою вузлів. У CouchDB нічого цього немає; натомість вона підтримує так звану реплікацію головний-головний, чи реплікацію з кількома головними вузлами.

Будь-який сервер CouchDB здатний нарівні з іншими приймати оновлення, відповідати на запити та видаляти дані, навіть якщо він не може зв'язатися з жодним іншим сервером. У такій моделі зміни вибірково реплікуються в одному напрямку, і всі дані можуть служити предметом реплікації. Інакше кажучи, сегментування немає. Кожен сервер, який бере участь у реплікації, зберігає всі дані.

Реплікація – остання велика особливість CouchDB, яку ми обговоримо. Спочатку подивимося, як налаштувати одноразову та безперервну реплікацію баз даних. А потім розглянемо, до чого можуть призводити конфлікти в даних і як писати програми, здатні вирішувати ці конфлікти.

Для початку клацніть посилання Replicator в меню Tools, розташованому на сторінці праворуч. В результаті відкриється сторінка, показана на рис. 29. У формі «Replicate changes from» (Реплікувати зміни з) виберіть у лівому списку, що розкривається, базу даних music, а в поле праворуч від нього введіть music-repl. Прапорець Continuous (Неперервно) не відзначайте та натисніть кнопку Replicate. Коли система запитає, чи створити базу даних music-repl, натисніть OK. В результаті журналу подій під формою з'явиться повідомлення про нову подію.

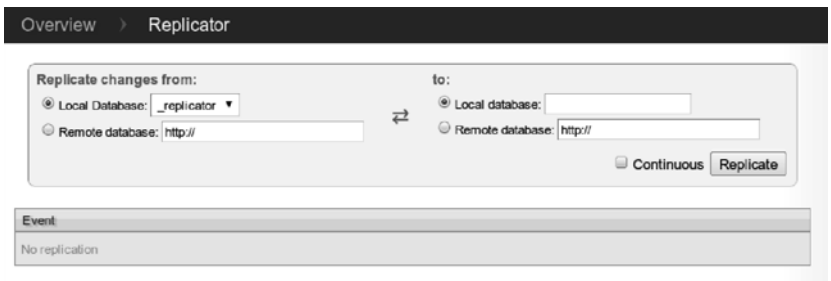


Рис. 29. CouchDB Futon: вікно реплікатора

Щоб переконатися, що запит реплікації спрацював, поверніться на сторінку Overview у Futon Overview. На ній має з'явитися нова база даних music-repl з таким самим числом документів, як у базі даних music. Якщо виявилося, що документів менше, то зачекайте трохи й оновіть сторінку – можливо, CouchDB ще не встигла скопіювати всі дані. Не дивуйтеся, якщо значення Update Seq (порядковий номер оновлення) не співпадають. Це пов'язано з тим, що у вихідній базі даних music ми оновлювали та видаляли документи, а при реплікації до бази music-repl виконуються лише вставки – задля прискорення.

Створення конфліктів

Далі ми навмисно створимо конфлікт та подивимося, як його обробляти. Тримайте сторінку Replicator відкритою, оскільки ми часто запускатимемо одноразову реплікацію між базами music і music-repl.

Введіть у вікні терміналу наступну команду для створення документа у базі даних music:

```
$ curl -X PUT "http://localhost:5984/music/theconflicts" \
-H “Content-Type: application/json” \
-d '{ “name”: “The Conflicts” }'
{
“ok”:true, “id”:”theconflicts”,
“rev”:”1-e007498c59e95d23912be35545049174”
}
```

На сторінці Replicator натисніть кнопку Replicate, щоб знову запустити синхронізацію. Переконатись, що документ реплікований успішно, можна, запитавши його з бази даних music-repl.

```
$ curl “http://localhost:5984/music-repl/theconflicts”
{
“_id”:”theconflicts”,
“_rev”:”1-e007498c59e95d23912be35545049174”,
“name”:”The Conflicts”
}
```

Потім оновіть базу даних `music-repl` додавши альбом *Conflicts of Interest*.

```
$ curl -X PUT "http://localhost:5984/music-repl/theconflicts" \
-H “Content-Type: application/json” \
-d' {
"_id": "theconflicts",
"_rev": "1-e007498c59e95d23912be35545049174",
"name": "The Conflicts",
“albums”: [“Conflicts of Interest”]
```

```
}'
{
"ok":true, "id":"theconflicts",
"rev":"2-0c969fbfa76eb7fcdf6412ef219fcac5"
}
```

А у базі music створіть конфліктуюче оновлення, додавши інший альбом: Conflicting Opinions.

```
$ curl -X PUT "http://localhost:5984/music/theconflicts" \
-H "Content-Type: application/json" \
-d' {
"_id": "theconflicts",
"_rev": "1-e007498c59e95d23912be35545049174",
"name": "The Conflicts", "albums": ["Conflicting Opinions"]
}'
{
"ok":true, "id":"theconflicts",
"rev":"2-cab47bf4444a20d6a2d2204330fdce2a"
}
```

Зараз у базах `music` і `music-repl` є документ з одним і тим самим значенням _id: theconflicts. В обох документів версія дорівнює 2, і обидва засновані на одній і тій же вихідній редакції (1-e007498c59e95d23912be35545049174). Виникає питання: що станеться при спробі реплікації з однієї бази до іншої?

Тепер, коли бази перебувають у конфліктному стані, перейдіть на сторінку Replicator і знову запустіть реплікацію. Якщо ви очікували на помилку, то будете вражені, побачивши, що операція завершилася успішно. Як з'ясовується, CouchDB просто вибирає одне оновлення та оголошує його переможцем. Завдяки використанню детермінованого алгоритму CouchDB при виявленні конфлікту завжди призначає переможцем одне й те саме оновлення. Однак, на цьому історія не закінчується. CouchDB зберігає і проігноровані документи, щоб клієнтська програма згодом могла оцінити ситуацію і вирішити конфлікт самостійно.

Щоб дізнатися, яка версія документа перемогла під час останньої реплікації, ми можемо запросити її за допомогою звичайного GET-запиту. Якщо додати URL параметр conflicts=true, то CouchDB включить також відомості про конфліктуючі редакції.

```
$ curl http://localhost:5984/music-repl/theconflicts?conflicts=true
{
"_id":"theconflicts",
"_rev":"2-cab47bf4444a20d6a2d2204330fdce2a", "name":"The Conflicts",
"albums":["Conflicting Opinions"], "_conflicts":[
"2-0c969fbfa76eb7fcdf6412ef219fcac5"
]
```

```
}
```

Як бачимо, перемогло друге оновлення. Зверніть увагу на поле _conflicts у відповіді. Воно містить список редакцій, що конфліктують із обраною. Додавши до GET-запиту параметр rev, ми зможемо отримати конфліктуючі редакції та вирішити, що з ними робити.

```
$ curl http://localhost:5984/music-repl/theconflicts?rev=2-0c969f...
{
"_id":"theconflicts",
"_rev":"2-0c969fbfa76eb7fcdf6412ef219fcac5", "name":"The Conflicts",
"albums":["Conflicts of Interest"]
}
```

CouchDB не намагається інтелектуально поєднувати конфліктуючі зміни. Як це робити, залежить від програми, і жодного спільного рішення не існує.

Наприклад, розглянемо базу даних подій у календарі. Одна її копія зберігається у вашому смартфоні, інша – у вашому ноутбуку. Ви отримуєте від планувальника прийомів текстове повідомлення із зазначенням місця проведення прийому, який ви організуєте, та вносите дані до смартфонної бази даних. Пізніше, вже в офісі, ви отримуєте від планувальника повідомлення електронною поштою, де вказано інше місце проведення. Тепер ви оновлюєте базу даних у ноутбуці та запускаєте реплікацію. CouchDB не знає, яке місце проведення правильне. Максимум, що вона може зробити, - підтримати узгодженість, зберігши десь старе значення, щоб ви могли вирішити, яке з двох конфліктуючих значень залишити. Програма повинна буде організувати інтерфейс для представлення такої ситуації, залишивши остаточне рішення на розсуд користувача.

Висновок

На цьому закінчується наше коротке ознайомлення з CouchDB. Ми почали сьогодні з того, що додали функції-редуктори до представлень , що генерується каркасом mapreduce. Потім ми поринули у вивчення інтерфейсу Changes API і здійснили захоплюючу подорож у світ серверних подійно-керованих програм JavaScript на прикладі Node.js. І нарешті, ми трохи поговорили про те, як у CouchDB реалізована стратегія реплікації головний-головний і як клієнтська програма може виявити та вирішити конфлікти.

У цій лекції ми розглянули досить широкий спектр завдань, які вирішуються за допомогою CouchDB, - від найпростіших операцій CRUD до побудови представлень за допомогою mapreduce-функцій. Ми бачили, як відстежувати зміни, і познайомилися з розробкою клієнтських програм, які не блокують подійно-керованих клієнтів. Зрештою, ми навчилися виконувати одноразову реплікацію баз даних, а також виявляти та вирішувати конфлікти. І хоча ще залишилося багато нерозглянутих тем, настав час підбити підсумки і переходити до наступної бази даних.

Сильні сторони CouchDB. CouchDB – надійний та стабільний представник сімейства баз даних категорії NoSQL. Припускаючи, що мережі принципово ненадійні, а апаратні збої неминучі, CouchDB пропонує децентралізований підхід до організації сховищ даних. Досить мала, щоб уміститися в смартфоні, і досить велика для підтримки корпоративних рішень, CouchDB може бути розгорнута на різних платформах.

CouchDB– однаково база даних та API. У цій лекції ми приділили основну увагу канонічному проєкту Apache CouchDB, що існують альтернативні реалізації та постачальники служб CouchDB, побудованих на базі гібридних серверних рішень, причому їх кількість постійно зростає. Оскільки CouchDB з'явилася «з веба для веб», то вона досить просто вбудовується як окремий шар у вебтехнології – подібно до балансувальників навантаження та розподілених кешів, – але при цьому зберігає унікальність своїх API.

Сильні сторони CouchDB. Зрозуміло, CouchDB годиться не для будь-якого завдання. Засновані на технології mapreduce представлення, при всій своїй новизні, не можуть забезпечити настільки ж гнучкі засоби вибірки даних, як реляційні СУБД. Насправді, у виробничій системі взагалі не рекомендується вдаватися до довільних запитів. Крім того, прийнята CouchDB стратегія реплікації не завжди є правильним вибором. У CouchDB в основу реплікації покладено принцип «все або нічого», тобто на всіх серверах, що реплікуються, зберігаються одні і ті ж дані. Не існує механізму

сегментування, що дозволяє розподілити дані щодо різних серверів у ЦОД. Основна причина додавання нових вузлів у CouchDB – не так розподілити дані, як збільшити продуктивність операцій читання та запису.

Завдання для індивідуальної роботи

1. Які редуктори спочатку вбудовані в CouchDB? У чому переваги використання вбудованих редукторів у порівнянні з користувачами, написаними на JavaScript?

2. Як відфільтрувати зміни, що надходять від служби _changes на стороні сервера?

3. Як і все інше в CouchDB, ініціалізація та скасування реплікації керуються HTTP-запитами. Які REST-команди використовуються для налаштування та видалення відносин реплікації між серверами?

4. Як можна скористатися базою даних _replicator для збереження відносин реплікації?

5. Створіть новий модуль watch_changes_continuous.js для Node.js на базі заготовки, наведений у розділі «Опитування змін у програмі для Node.js».

6. Документи з конфліктуючими редакціями мають властивість _conflicts. Створіть представлення, яке емітуватиме конфліктуючі редакції та відображатиме їх на ідентифікатор документа _id.

Лекція 12. Основи Neo4J

Neo4j – новий тип сховищ даних NoSQL, який отримав назву графової бази даних. Як випливає із назви, дані в ній зберігаються у вигляді графа (у тому сенсі, в якому він розглядається в математиці). Відмінною особливістю таких баз даних є можливість малювати їх структуру на дошці у вигляді прямокутних блоків і ліній, що їх з'єднують. Все, що можна зобразити у такому вигляді, можна зберегти в Neo4j. У Neo4j акцент робиться скоріше на зв'язку між значеннями, ніж на загальні характеристики значень (як у колекціях документів або в рядках таблиці). Таким чином, абсолютно різні дані можна зберігати просто і природно. Розмір Neo4j невеликий - настільки, що її можна впровадити практично в будь-яку програму. З іншого боку, в Neo4j можна зберігати десятки мільярдів вузлів і стільки ж ребер.

СУБД здатна впоратися із завданням будь-якого розміру. Припустимо, нам необхідно створити систему рекомендації вин, в якій для вина можуть бути визначені сорт, регіон виробництва, виноградник, вінтаж. Можливо, потрібно зберігати статті авторів, що описують вина. Або дати користувачам можливість помічати свої улюблені вина.

У реляційній моделі ви, напевно, створили б таблицю категорій і зв'язок багато-багатьом між вином з одного виноградника і деякою комбінацією категорій та інших даних. Але людина подумки моделює дані інакше. У Neo4j ця проблема вирішується неявно – за рахунок того, що значення та структурні зв'язки визначаються лише там, де потрібно. Якщо для купажного вина немає вінтажу, можна додати рік розливу по пляшках і послатися з вінтажів на вузол купажування. Жодної жорсткої схеми не існує.

Ми будемо використовувати версію Neo4j 1.7 Enterprise.

Графи, Groovy та операції CRUD

Все в Neo4j обертається навколо зв'язків. Запустимо веб-інтерфейс і подивимося, як Neo4j представляє дані у вигляді графа та як цей граф *обходити*. Завантаживши та розпакувавши пакет Neo4j, перейдіть в інсталяційний каталог і запустіть сервер:

```
$ bin/neo4j start
```

Щоб перевірити, чи сервер запущений і працює, надішліть за допомогою curl запит на таку URL-адресу:

```
$ curl http://localhost:7474/db/data/
```

Як і CouchDB, стандартний пакет Neo4j включає вражаючий адміністративний веб-інтерфейс та засоби перегляду даних. Запустіть браузер і перейдіть на сторінку адміністрування:

```
http://localhost:7474/webadmin/
```

Ви побачите кольоровий, але поки що порожній граф, зображений на рис. 32. Натисніть посилання Data Browser у верхній частині сторінки. У щойно встановленому екземплярі Neo4j є один зумовлений вузол зв'язку: вузол 0.

Рис. 32. Індикаторна панель на сторінці адміністративного веб-інтерфейсу

Вузол в графової базі даних певною мірою нагадує вузли тому, у якому ми вживали це у попередніх главах. Раніше, говорячи про сайт, ми мали на увазі фізичний сервер у мережі. Якщо розглядати всю мережу як гігантський взаємопов'язаний граф, серверні вузли будуть вершинами графа, а зв'язки між ними – ребрами.

У Neo4j під вузлом розуміється те саме: вершина, з якої виходять ребра, де можуть зберігатися дані у вигляді пар ключ-значення. Натисніть кнопку + Property і задайте ключ name та значення Prancing Wolf Ice Wine 2007, що представляють конкретне вино та вінтаж. Потім натисніть наведену нижче кнопку + Node, яка додає вузол.

Для новоствореного вузла додайте властивість `name` зі значенням Wine Expert Monthly (ми коротко записуватимемо це у вигляді `[name: "Wine Expert`

`Monthly"]`). Номер вузла автоматично збільшиться.

Тепер ми маємо два вузли, але їх ніщо не пов'язує. Оскільки журнал Wine Expert оцінював вино Prancing Wolf, ми маємо зв'язати ці вузли, створивши ребро. Натисніть кнопку + Relationship і встановіть зв'язок, що йде з вузла 1 у вузол 0, надавши їй reported_on.

Ви перейдете на сторінку, що відповідає цьому зв'язку:

`http://localhost:7474/db/data/relationship/0`

де показано, що `Node 1 reported_on Node 0.`

Як і в вузлів, зв'язки можуть мати властивості. Натисніть кнопку + Add Property і введіть властивість [rating : 92] – це оцінка виставлена вину.

Це конкретне вино з винограду сорту riesling, додамо і цю інформацію. Можна було б додати цю властивість прямо у вузол вина, але сорт - це загальна категорія, яка може бути асоційована і з іншими винами, тому створимо новий вузол і задамо йому властивість [name : “riesling”]. Потім додайте ще один зв'язок, що йде з вузла 0 у вузол 2, надавши їй тип grape_type і задавши властивість [style : "ice wine"].

Тепер натиснувши кнопку «переключити режим перегляду» ви побачите граф, зображений на рис. 33. Кнопка Style відкриває меню, де можна вибрати, який профіль використовувати для візуалізації графа. Натисніть Style і виберіть пункт New Profile, щоб показати на діаграмі більш корисну інформацію. Ви потрапите на сторінку Create new visualization profile (Створити новий профіль візуалізації). Введіть у полі нагорі ім'я wines, після чого змініть мітку з {id} на {id}: {prop.name}. Після натискання кнопки Save ви опинитеся знову на сторінці візуалізації. Тепер із меню Style можна вибрати пункт wines – вийде картина, зображена на рис. 34.

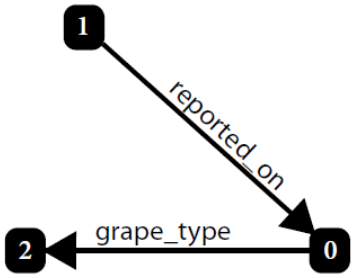


Рис. 33. Граф вузлів, пов'язаних з поточним

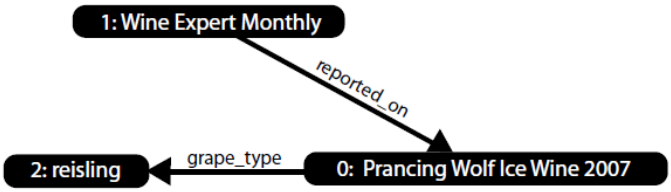


Рис. 34. Граф вузлів, візуалізований за спеціальним профілем

Хоча описаного веб-інтерфейсу достатньо для внесення простих змін, створення професійних графів потребує більш досконалих інструментів.

Neo4j та Gremlin

До Neo4j можна звертатися з кількох мов: Java, REST-інтерфейс, Cypher, Ruby та інших. Але ми скористаємось мовою Gremlin, яка написана мовою програмування Groovy і спеціально призначена для обходу графів. Втім, щоб використовувати Gremlin, Groovy знати необов'язково; Ви можете вважати, що це просто декларативна предметно-орієнтована мова типу SQL. Як і інші консольні оболонки, що зустрічалися нам раніше, Gremlin дає доступ до інфраструктури мови, на якій заснований. Це означає, що у програмі на Gremlin можна використовувати конструкції Groovy та бібліотеки, написані на Java. Більше того, консоль Gremlin доступна і в адміністративному веб-інтерфейсі; просто клацніть посилання Console у верхній частині сторінки і виберіть Gremlin.

За ухваленим порядком, змінна g представляє граф як об'єкт. Дії над графом – це функції, що викликаються від імені цього об'єкта.

Оскільки Gremlin – універсальна мова обходу графів, то ньому використовуються терміни з математичної теорії графів. Те, що у Neo4j називається вузлом (node), в Gremlin називається вершиною (vertex), а замість зв'язку (relationship) використовується термін ребро (edge).

Для доступу до багатьох вершин графа є властивість з ім'ям V (від слова vertices).

```
gremlin > gV
==>v[0]
==>v[1]
==>v[2]
```

а доступу до ребрів графа – властивість E.

```
gremlin > g.E
==> e[0][1-reported_on->0]
==> e[1][0-grape_type->2]
```

Отримати конкретну вершину можна, передавши номер вузла методу v (у нижньому регістрі).

```
gremlin> gv(0)
==> v[0]
```

Переконатися, що отримана необхідна вершина, можна, перерахувавши її властивості шляхом map(). Зазначимо, що у Groovy/Gremlin методи можна зчеплювати:

```
gremlin> gv(0).map()
==> name=Prancing Wolf Ice Wine 2007
```

Виклик v(0) повертає конкретний вузол, але можна було б профільтрувати множину вузлів за вказаним значенням. Наприклад, щоб отримати вузол riesling на ім'я, можна скористатися синтаксисом фільтра {…}, який у Groovy називається замиканням (closure). Весь код усередині фігурних дужок визначає функцію; якщо вона повертає true, програма відвідує відповідну вершину. Змінна it всередині замикання представляє поточний об'єкт, значення їй надається автоматично.

```
gremlin> gVfilter{it.name=='riesling'}
==> v[2]
```

Маючи вершину, ми можемо отримати ребра, що виходять з неї, викликавши метод outE(). Для отримання вхідних ребер служить метод inE(), а отримання одночасно вхідних і вихідних – метод bothE().

```
gremlin> gVfilter{it.name=='Wine Expert Monthly'}.outE()
==> e[0][1-reported_on->0]
```

Зазначимо, що в Groovy, як і в Ruby, дужки при викликі методів можна опускати, тому запис outE дає той самий результат.

```
gremlin> gVfilter{it.name=='Wine Expert Monthly'}.outE
==> e[0][1-reported_on->0]
```

Знаючи вихідні ребра, ми можемо пройти в сусідні вершини, викликавши метод inV, тобто отримати вершини, до яких ведуть ребра.

Ребро `reported_on`, що виходить із вузла Wine Expert, веде у вершину Prancing Wolf Ice Wine 2007, тому виклик `outE.inV` її й поверне. Потім можна прочитати властивість `name` цієї вершини:

```
gremlin> gVfilter{it.name=='Wine Expert Monthly'}.outE.inV.name
==> Prancing Wolf Ice Wine 2007
```

Вираз outE.inV повертає всі вершини, до яких ведуть вихідні ребра. Зворотна операція (отримати всі вершини, з яких ведуть ребра в задану множину вершин) реалізується виразом inE.outV. Оскільки ці операції використовуються дуже часто, у Gremlin є для них скорочена нотація. Вираз out – це скорочений запис для outE.inV, а вираз in – для inE.outV.

```
gremlin> gVfilter{it.name=='Wine Expert Monthly'}.out.name
==> Prancing Wolf Ice Wine 2007
```

В одному виноробному господарстві можуть вироблятися різні вина, і, якщо ми плануємо зберігати кілька вин, то маємо додати виноградник як сполучну вершину, з якої веде ребро у вершину Prancing Wolf.

```
gremlin> pwolf = g.addVertex([name : 'Prancing Wolf Winery'])
==> v[3]
gremlin> g.addEdge(pwolf, gv(0), 'produced')
==> e[2][3-produced->0]
```

Тепер можна додати ще два сорти: Kabinett і Spatlese.

```
gremlin> kabinett = g.addVertex([name : 'Prancing Wolf Kabinett 2002'])
==> v[4]
gremlin> g.addEdge(pwolf, kabinett, 'produced')
==> e[3][3-produced->4]
gremlin> spatlese = g.addVertex([name : 'Prancing Wolf Spatlese 2007'])
==> v[5]
gremlin> g.addEdge(pwolf, spatlese, 'produced')
==> e[4][3-produced->5]
```

Поповнимо цей невеликий граф, додавши ребра, що ведуть з вершини riesling до нових вершин. Встановимо змінну riesling, відфільтрувавши вершину riesling; для отримання першої вершини з конвеєра необхідний метод next(), про який ми поговоримо трохи нижче.

```
gremlin> riesling = gVfilter{it.name=='riesling'}.next()
==> v[2]
gremlin> g.addEdge([style:'kabinett'], kabinett, riesling, 'grape_type')
==> e[5][4-grape_type->2]
```

Вершину Spatlese можна з'єднати ребром з riesling аналогічно, тільки у властивості style слід записати spatlese. Після цих дій можна візуалізувати граф (рис. 35).

Рис. 35. Граф вузлів після додавання даних за допомогою Gremlin

Конвеєри

Операції в Gremlin можна як послідовність каналів (pipe). Кожен канал на вході приймає колекцію, а на виході повертає іншу колекцію. Колекція може бути нуль, один або багато елементів. Елементами можуть бути вершини, ребра чи значення властивостей.

Наприклад, канал outE приймає колекцію вершин та виводить колекцію ребер. Послідовність каналів називається конвеєром (pipeline) і служить для декларативного опису завдання. Порівняйте це з типовим імперативним програмним підходом, в якому ви повинні описати кроки, які виконуються для вирішення задачі. Конвеєри – один із найкоротших способів сформулювати запит до графової бази даних.

За своєю суттю Gremlin – мова для побудови каналів. Точніше він створений на базі проєкту Pipes, написаного на Java. Для вивчення ідеї каналів повернемося до нашого графа вин. Припустимо, що потрібно знайти вина, схожі на це, тобто виготовлені з того ж сорту винограду. Ми можемо пройти з вершини, що відповідає крижаному вину, по ребру grape_type і потім знайти вершини, з яких виходять ребра, що ведуть в ту саму вершину (пропустивши вузол, з якого вийшли).

```
      ice_wine = gv(0)
      ice_wine.out('grape_type').in('grape_type').filter{
!it.equals(ice_wine)}
```

Якщо вам доводилося писати мовою Smalltalk або працювати з контекстами на платформі Rails, такий спосіб зчеплення методів вам знайомий. А тепер порівняйте це з кодом, написаним за допомогою стандартного Neo4j Java API, де для доступу до вузлів вин з тими ж сортовими ознаками доводиться обходити зв'язки з вузла:

```
      enum WineRelationshipType implements RelationshipType { grape_type
      }
      import static WineRelationshipType.grape_type; public static List<Node>
same_variety( Node wine ) {
      List<Node> wine_list = новий ArrayList<Node>();
      // пройти по всіх ребрах, що виходять з цієї вершини
      for( Relationship outE : wine.getRelationships( grape_type ) ) {
      // пройти по всіх ребрах, що входять у протилежну вершину
      // даного ребра
```

```
      for( Edge inE : outE.getEndNode().getRelationships( grape_type ) ) {
      // додавати тільки вершини, відмінні від вихідної
if(!inE.getStartNode().equals(wine)) {
      wine_list.add( inE.getStartNode() );
      }
      }
      }
      return wine_list;
      }
```

Замість вкладених циклів та обходу ребер проєкт Pipes пропонує спосіб оголошувати вхідні та вихідні вершини. Спочатку створюється послідовність вхідних і вихідних каналів, фільтрів і значень, що запитуються у конвеєра. Потім циклі викликаємо метод конвеєра next(), який повертає наступний відповідний вузол. Іншими словами, конвеєр обходить дерево. Але доки у конвеєра нічого не запрошено, ми просто оголошуємо порядок обходу.

Наприклад наведемо ще одну реалізацію методу same_variety(), в якій замість явних циклів використовується Pipes:

```
      gVfilter{it.id=='navigation'}.out.filter{it.tag=='li'}.
out.filter{it.name=='section1'}.text

      public static void same_variety( Vertex wine ) { List<Vertex> wine_list
= new ArrayList<Vertex>(); Pipe inE = new InPipe( "grape_type");
      Pipe outE = new OutPipe( "grape_type");
      Pipe not_wine = новий ObjectFilterPipe (wine, true); Pipe<Vertex,Vertex>
pipeline =
      new Pipeline<Vertex,Vertex>(outE, inE, not_wine); pipeline.setStarts(
Arrays.asList( wine ) );
      while( pipeline.hasNext() ) { wine_list.add( pipeline.next() );
      }
      return wine_list;
      }
```

Завдання обходу графа все одно вирішується на сервері Neo4j, але Gremlin спрощує формулювання запитів, які розуміє Neo4j.

Користувачам популярної бібліотеки jQuery для мови JavaScript метод обходу колекцій, прийнятий у Gremlin, ймовірно, здасться знайомим. Розглянемо наступний фрагмент HTML:

```
      <ul id="navigation">
      <li>
      <a name=”section1”>section 1</a>
      </li>
      <li>
      <a name=”section2”>section 2</a>
      </li>
      </ul>
```

Припустимо, що потрібно знайти текст усередині всіх тегів з ім'ям

section1, які є нащадками елементів <li>, які знаходяться всередині елемента з ідентифікатором navigation (id=navigation). На jQuery для цього можна було б написати такий вираз:

```
$('[id=navigation]').children('li').children('[name=section1]').text()gV
filter{it.id=='navigation'}.out.filter{it.tag=='li'}.
out.filter{it.name=='section1'}.text
```

А тепер розглянемо, як міг би виглядати запит на Gremlin до аналогічного набору даних, що з кожного батьківського вузла ведуть ребра в усі його дочірні вузли.

Конвеєр та вершина

Щоб отримати колекцію, яка містить лише одну вказану вершину, ми можемо її відфільтрувати від списку всіх вузлів. Саме це ми й робимо у викликі gVfilter{it.name=='reisling'}. Властивість V представляє список всіх вузлів, з якого вибираємо підсписок. Але щоб отримати саму вершину, потрібно викликати метод next(), який повертає перший елемент конвеєрі. Це схоже на різницю між масивом з одного елемента і самим елементом.

Cypher – ще одна підтримувана Neo4j мова запитів до графів. Він заснований на порівнянні з зразками і має SQL-подібний синтаксис. Фрази виглядають знайомо, що допомагає зрозуміти зміст запиту. Особливо інтуїтивно зрозуміла фраза MATCH, що робить вирази схожими на стародавні малюнки ASCII-символами

Ось, як на Cypher записується еквівалент запиту «схожі вина»:

```
START ice_wine=node(0)
MATCH (ice_wine) -[:grape_type]-> () <-[:grape_type](similar) RETURN
similar
```

Ми починаємо з прив'язки ідентифікатора ice_wine до вузла 0. У фразі MATCH використовуються ідентифікатори в круглих дужках для позначення вузлів та типізовані «стрілки» виду -[:grape_type]->для позначення двосторонніх зв'язків.

Однак можна піти і далі.

```
START ice_wine=node:wines(name="Prancing Wolf Ice Wine 2007") MATCH
ice_wine -[:grape_type]-> wine_type <-[:grape_type]similar WHERE wine_type =~
/(?i)riesl.*)/
RETURN wine_type.name, collect(similar) as wines, count(*) as wine_count
```

```
ORDER BY wine_count desc
      LIMIT 10
```

Хоча у цій лекції ми зупинилися на Gremlin, але обидві мови мирно співіснують одна з іншою. У повсякденній роботі можна використовувати той чи інший – в залежності від поставленого завдання.

Поглянувши на клас, сконструйований у результаті виклику властивості фільтра class, ви виявите, що фільтр повертає об'єкт GremlinPipeline.

```
gremlin> gVfilter{it.name=='Prancing Wolf Winery'}.class
==>class com.tinkerpop.gremlin.pipes.GremlinPipeline
```

Але порівняйте це із класом наступного вузла, отриманого з конвеєра. Це буде щось інше – Neo4jVertex.

```
gremlin> gVfilter{it.name=='Prancing Wolf Winery'}.next().class
==>class com.tinkerpop.blueprints.pgm.impls.neo4j.Neo4jVertex
```

Хоча на консолі зручно виводити списки вузлів, вилучених із конвеєра, конвеєр залишається таким, поки з нього не буде витягнуто щось.

Безсхемна соціальна мережа

Для привнесення до графа соціального аспекту достатньо додати ще кілька вузлів. Припустимо, потрібно додати трьох людей, знайомих між собою, кожен з яких має улюблені вина.

Аліса – обожнює крижане вино.

```
      alice = g.addVertex([name:'Alice'])
      ice_wine= gVfilter{it.name=='Prancing Wolf Ice Wine 2007'}.next()
g.addEdge(alice, ice_wine, 'likes')
```

Том любить Кабінет (Kabinett) і крижане вино і довіряє всьому, що пише Wine Expert Monthly.

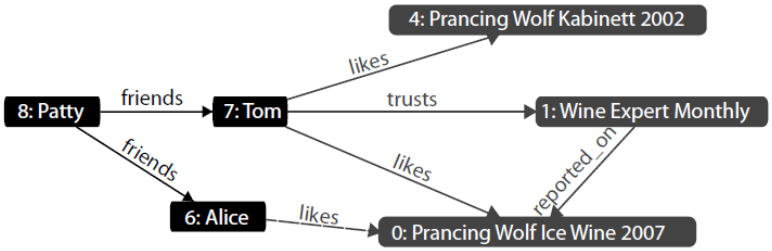
```
      tom = g.addVertex([name:'Tom'])
      kabinett = gVfilter{it.name=='Prancing Wolf Kabinett 2002'}.next()
g.addEdge(tom, kabinett, 'likes')
      g.addEdge(tom, ice_wine, 'likes')
      g.addEdge(tom, gVfilter{it.name=='Wine Expert Monthly'}.next(),
'trusts')
```

Петті товаришує з Томом і Алісою, але у світі вина новачок, їй ще доведеться визначитися зі своїми уподобаннями.

```
      patty = g.addVertex([name:'Patty']) g.addEdge(patty, tom, 'friends')
g.addEdge(patty, alice, 'friends')
```

Не змінюючи базову структуру існуючого графа, ми змогли накласти на нього поведінку, яка наперед не закладалася. Нові вузли з'єднані, як

показано нижче.



Подальша взаємодія з графом

Ми розглянули кілька основних кроків, або блоків обробки каналів у Гремлін. Але це далеко ще не все. Познайомимося з іншими елементами, які дозволяють не тільки обходити граф, але також перетворювати об'єкти, фільтрувати кроки та робити побічні ефекти, наприклад, підраховувати вузли, що згруповані за деяким критерієм.

Ми вже зустрічалися з методами inE, outE, inV та outV, які являють собою кроки трансформації, що дозволяють витягувати вхідні та вихідні ребра та вершини. Надаються також методи bothE і bothV, які йдуть вздовж ребра незалежно від того, є воно вхідним або вихідним.

Наступний запит знаходить Алісу та всіх її друзів. Наприкінці виразу ми додали name, щоб отримати властивість name кожної вершини. Оскільки нам байдуже, куди направлено ребро friend, ми скористалися кроками bothE та bothV.

```
alice.bothE('friends').bothV.name
==> Alice
==> Patty
```

Якщо Аліса нас не цікавить, можна включити в конвеєр фільтр except(), передавши йому список непотрібних вузлів.

```
alice.bothE('friends').bothV.except([alice]).name
==> Patty
```

Протилежністю except() є фільтр retain(), який, як легко здогадатися, відвідує ті вузли, які йому передані.

Можна зробити і по-іншому – на останньому кроці відфільтровувати вершину за допомогою коду, що перевіряє, що ця вершина не збігається з alice.

```
alice.bothE('friends').bothV.filter{!it.equals(alice)}.name
```

А як дізнатися, з ким дружать друзі Аліси? Просто повторити кроки:

```
alice.bothE('friends').bothV.except([alice]).
bothE('friends').bothV.except([alice])
```

Точно так само можна було б знайти друзів друзів друзів Аліси, додавши в ланцюжок ще одну послідовність викликів `bothE/bothV/except`. Але довелося б надто довго друкувати, і до того ж такий підхід не узагальнюється змінною кількістю кроків. Для цього призначений метод loop(). Він повторює попередні кроки, доки задане замикання залишається рівним true.

Наведений нижче код у циклі повторює три попередні кроки, відраховуючи назад точки перед викликом методу loop: крок except – перший, крок bothV – другий та крок bothE – третій.

```
alice.bothE('friends').bothV.except([alice]).loop(3){ it.loops <=
2}.name
```

Після кожної ітерації loop() викликає передане йому замикання, тобто код фігурних дужках. В даному випадку у властивості `it.loops` зберігається кількість вже виконаних ітерацій циклу. Ми порівнюємо це число з 2 і повертаємо результат перевірки, отже цикл зупиниться після двох ітерацій. Фактично, замикання дуже схоже умова циклу while у типовій мові програмування.

```
==> Tom
==>Patty
==>Patty
```

Цикл правильно знайшов Тома та Петті. Тільки у нас вийшло дві копії Петті: перша - тому що Петті товаришує з Алісою, друга - тому що вона товаришує з Томом. Отже, треба якось відфільтрувати дублікати. Для цього призначено фільтр dedup().

```
alice.bothE('friends').bothV.except([alice]).loop(3){ it.loops <= 2
}.dedup.name
==> Tom
==>Patty
```

Щоб зрозуміти, як були отримані ці значення, можна пройти шляхом друг->друг, скориставшись перетворенням paths().

```
alice.bothE('friends').bothV.except([alice]).loop(3){ it.loops <= 2
}.dedup.name.paths
==> [v[7], e[12][9-friends->7], v[9], e[11][9-friends->8], v[8], Tom]
==> [v[7], e[12][9-friends->7], v[9], e[11][9-friends->8], v[9], Patty]
```

Досі ми обходили граф лише у прямому напрямку. Але іноді потрібно

зробити два кроки вперед та два тому. Якщо, розпочавши з вузла Аліси, зробити два кроки вперед, а потім два кроки назад, ми повернемося у вихідний вузол.

```
gremlin> alice.outE.inV.back(2).name
==> Alice
```

І останнім ми розглянемо ще один часто використовуваний крок groupCount(), який обходить вузли і підраховує повторювані значення, зберігаючи їх в асоціативному масиві. У наступному прикладі ми обходимо всі вершини графа і підраховуємо, скільки разів зустрічаються різні значення якості name.

```
gremlin>name_map=[:]
gremlin> gVname.groupCount( name_map ) gremlin> name_map
==> Prancing Wolf Ice Wine 2007=1
==> Wine Expert Monthly=1
==> riesling=1
==> Prancing Wolf Winery=1
==> Prancing Wolf Kabinett 2002=1
==> Prancing Wolf Spatlese 2007=1
==> Alice=1
==> Tom=1
==> Patty = 1
```

У мовах Groovy та Gremlin асоціативний масив позначається символом [:] і практично не відрізняється від об'єктів у літеральній нотації {} у мовах Ruby та JavaScript. Зверніть увагу, що всі значення дорівнюють 1. Цього й слід було очікувати, оскільки в нас не було імен, що повторюються, а в колекцію V кожна вершина графа входить рівно один раз. Далі підрахуємо, скільки марок вина любить кожна зареєстрована в системі людина. Для цього потрібноотримати всі вершини, що описують улюблені вина, і підрахувати, скільки відповідає кожному імені.

```
gremlin>wines_count=[:]
gremlin> gVoutE('likes').outV.name.groupCount( wines_count ) gremlin>
wines_count
==> Alice=1
==> Tom=2
```

Як і очікувалося, Аліса любить одну марку вина, а Том дві.

Перехід на рівень Groovy

Крім кроків на Gremlin, у нашому розпорядженні широкий спектр конструкцій та методів мови Groovy. У Groovy є функція розподілу (у дусі mapreduce), яка називається collect(), і функція редукування inject(). З їхньою

допомогою можна заздалегідь сформулювати запити, як у технології mapreduce.

Нехай потрібно підрахувати, скільки вин ще не отримали оцінки. Для цього можна спочатку збудувати список значень true/false, що показують, оцінено вино чи ні. Потім цей перелік подається на вхід редуктора, який підраховує кількість true і false. На фазі розподілу функція collect використовується так:

```
rated_list = gVin('grape_type').collect{
!it.inE('reported_on').toList().isEmpty()
}
```

Тут вираз `gVin('grape_type')` повертає вершини, в які входить ребро типу grape_type. Оскільки такі ребра можуть входити лише у вершини, що описують вина, ми тим самим отримуємо список всіх вин. Далі у замиканні collect ми дивимося, чи входить у поточну вершину ребро типу reported_on. Виклик методу toList() перетворює конвеєр на цей список, який ми можемо перевірити на порожнечу. У змінній rated_list, що породжується цим кодом, буде знаходитись список значень true і false.

Щоб підрахувати, скільки вин не оцінено, редукуємо цей список, скориставшись методом inject().

```
rated_list.inject(0){ count, is_rated -> if (is_rated) {
count
} else { count + 1
}
}
```

У Groovy оператор "стрілка" (->) відокремлює вхідні аргументи замикання з його тіла. У нашому редукторі хочемо обробити значення, зібрані на етапі розподілу, і зрозуміти, оцінено поточне вино чи ні; для цього і потрібні змінні count та is_rated. Аргумент 0 inject(0) служить для ініціалізації count нулем перед першим зверненням. Потім у тілі замикання ми повертаємо поточне значення лічильника count (якщо вино оцінено), або збільшуємо його на одиницю (якщо вино не оцінено). На виході ми отримаємо кількість значень false у списку (тобто кількість неоцінених вин).

```
==> 2
```

Як з'ясовується, оцінки ще не отримали дві вина.

Маючи у своєму розпорядженні описані інструменти, ми можемо складати різні комбінації обходів та трансформацій графа. Нехай, наприклад, потрібно знайти всі пари друзів. Для цього потрібно спочатку знайти всі ребра типу friends, а потім вивести імена людей на обох кінцях ребра, скориставшись операцією transform.

```
gVoutE('friends').transform{[it.outV.name.next(), it.inV.name.next()]}
==> [Patty, Tom]
==> [Patty, Alice]
```

Тут замикання transform повертає літеральний масив ([...]) з двох елементів: вхідної та вихідної вершин ребра friend.

Щоб знайти всіх людей і вина, які вони люблять, ми перетворимо кожну знайдену людину (вершина на будь-якому кінці ребра типу friends) до списку з двох елементів: ім'я людини та список її улюблених вин.

```
gVboth('friends').dedup.transform{
[ it.name, it.out('likes').name.toList() ]
}
==> [Alice, [Prancing Wolf Ice Wine 2007]]
==> [Patty, []]
==> [Tom, [Prancing Wolf Ice Wine 2007, Prancing Wolf Kabinett 2002]]
```

Предметно-орієнтовані кроки

Gremlin дозволяє визначати також кроки, семантично пов'язані з даними, що зберігаються в графі.

Почнемо зі створення кроку різного роду, що дає відповіді на поставлене вище питання. Коли метод varietal() викликається для певної вершини, він шукає ребра типу grape_type, що виходять з неї, і слідує за ними до протилежних вершин.

Нам доведеться трохи заглибитись у Groovy, тому спочатку наведемо код кроку, а потім докладно опишемо його.

neo4j/varietal.groovy

```
Gremlin.defineStep( 'varietal', [Vertex, Pipe],
{_().out('grape_type').dedup}
)
```

Спочатку ми повідомляємо ядру Gremlin, що збираємося визначити крок з ім'ям varietal. У другому рядку ми говоримо, що новий крок потрібно приєднати до класів Vertex та Pipe (якщо сумніваєтеся, вказуйте обидва). І в

останньому рядку проводиться змістовна робота. По суті, ми створюємо замикання, що містить виконуваний кроком код. Знак підкреслення та круглі дужки представляють поточний об'єкт конвеєра. З цього об'єкта ми пройдемо до сусідніх вузлів по ребрах типу grape_type, тобто у вузол, що визначає сортові характеристики вина. І насамкінець викликаємо метод dedup, щоб усунути дублікати.

Новий крок викликається як будь-який інший. Наприклад, наступний запит повертає сорт винограду, з якого виготовлено крижане вино:

```
gVfilter{it.name=='Prancing Wolf Ice Wine 2007'}.varietal.name
==> riesling
```

Розглянемо ще один приклад. На цей раз напишемо крок для типової дії: отримати улюблені вина всіх друзів.

neo4j/friendsuggest.groovy

```
Gremlin.defineStep( 'friendsuggest', [Vertex, Pipe],
{
_().sideEffect{start = it}.both('friends').
except([start]).out('likes').dedup
}
)
```

Як і раніше, даємо новому кроку ім'я friendsuggest- І зв'язуємо його з класами Vertex і Pipe. Наш код має відфільтрувати поточну людину. Для цього ми зберігаємо поточну вершину або канал змінної (start), викликаючи функцію sideEffect{start = it}. Потім ми отримуємо всі вузли friends, крім поточного (ми не хочемо вважати Алісу другом самої себе).

І тепер новий крок можна включити в конвеєр як завжди.

```
gVfilter{it.name=='Patty'}.friendsuggest.name
==> Prancing Wolf Ice Wine 2007
==> Prancing Wolf Kabinett 2002
```

Оскільки varietal і friendsuggest - звичайні кроки, що будують канали, їх можна зчеплювати для отримання цікавіших запитів. Так, наступний запит знаходить сорти винограду, які віддають перевагу друзям Петті:

```
gVfilter{it.name=='Patty'}.friendsuggest.varietal.name
==> riesling
```

Використання наявних у Groovy засобів метапрограмування для створення нових кроків – потужний засіб розробки предметно-орієнтованих мов. Але до цього підходу, як і до Gremlin, потрібно звикнути.

Оновлення та видалення

Обходити граф і вставляти в нього дані ми навчилися, але як щодо оновлення та видалення? Просто – потрібно лише знайти вершину чи ребро, яке ви збираєтесь змінити. Давайте оцінимо рівень любові Аліси до вина Prancing Wolf Ice Wine 2007 року, приписавши їй вагу.

```
gremlin> e=gVfilter{it.name=='Alice'}.outE('likes').next() gremlin> e.weight = 95
gremlin> e.save
```

Видалити значення анітрохи не складніше.

```
gremlin> e.removeProperty('weight') gremlin> e.save
```

Об'єкт, що представляє граф, має функції для видалення вершин і ребер: removeVertex і removeEdge відповідно. Щоб знищити граф, ми можемо видалити всі вершини та ребра:

```
gremlin > gVeach { g.removeVertex (it) } gremlin > gEeach { g.removeEdge (it) }
```

Перевірити, що нічого не залишилося, можна, викликавши методи `g.V` і `g.E`. Тогож результату можна досягти за допомогою методу clear() – якщо не боїтеся.

```
gremlin> g.clear()
```

Якщо ви запустили консольну оболонку Gremlin (поза веб-інтерфейсом), то рекомендується коректно закривати з'єднання з графом методом shutdown().

```
gremlin> g.shutdown()
```

В іншому випадку можна пошкодити базу даних. Але зазвичай сервер просто надішле вам помилку при наступному підключенні до графа.

Висновок

Отже ми познайомилися з графовою базою даних Neo4j і зрозуміли, наскільки вона відрізняється від інших. Хоча ми не розповідали про специфічні патерни проєктування, але все, що можна намалювати на дошці, можна зберегти у графовій базі даних.

Простота Neo4j може вразити, якщо ви не звикли моделювати дані у вигляді графів. Незважаючи на потужний API з відкритим вихідним кодом та роки промислової експлуатації, у цієї СУБД все ще відносно мало користувачів. Ми пояснюємо це лише недостатньою інформованістю,

оскільки графові бази даних природно відбивають процес інтерпретації даних людиною. Ми представляємо сім'ї як дерев, друзів – як графи; мало хто розглядає зв'язок між людьми як самовідносні типи даних. Для деяких класів завдань, наприклад, соціальних мереж, вибір Neo4j напрошується сам собою. Але й у не настільки очевидних випадках має сенс придивитися до цієї СУБД уважніше – її міць та простота використання, можливо, здивують вас.

Завдання для індивідуальної роботи

1. Знайдіть ще оболонку для Neo4J (наприклад, оболонка Cypher на адміністративній консолі).
2. Запитайте імена всіх вузлів в іншій оболонці (наприклад, Cypher).
3. Увидаліть всі вузли та ребра з бази даних.
4. Створіть новий граф, який представляє вашу родину.

Лекція 13. REST, індекси та алгоритми у Neo4j

Створімо за допомогою REST-інтерфейсу вузли та зв'язки між ними у Neo4j, а потім побудуємо індекси та здійснимо повнотекстовий пошук. Потім ми розглянемо модуль, що підключається, який дозволяє виконувати на сервері Gremlin-запити, відправлені через REST-інтерфейс. Тим самим ми зможемо обійтися взагалі без консолі Gremlin і обмежень, що накладаються їй - навіть встановлювати Java на сервери додатків або клієнти необов'язково.

REST-інтерфейс

Як і Riak, HBase, Mongo та CouchDB, Neo4j поставляється з RESTінтерфейсом. Одна з причин, через які всі ці бази даних підтримують REST, полягає в бажанні позбавитися мовних залежностей, забезпечивши стандартний інтерфейс підключення. Ми можемо підключитися до сервера Neo4j - для роботи якого необхідна Java - з машини, на якій ніяких слідів Java немає і близько. А завдяки модулю Gremlin, що підключається, ми зможемо через REST скористатися виразною міццю лаконічного синтаксису цієї мови.

Перш за все, перевірте, що REST-сервер запущено, для чого надішліть GET-запит на базовий URL, який повертає кореневий вузол. Сервер прослуховує той самий порт, як і розглянутий вчора адміністративний веб-інтерфейс, тільки йому відповідає шлях `/db/data/`. Для надсилання REST-запитів ми скористаємося програмою curl.

```
$ curl http://localhost:7474/db/data/
{
"relationship_index":
“http://localhost:7474/db/data/index/relationship”,“node” :
“http://localhost:7474/db/data/node”, “relationship_types” :
“http://localhost:7474/db/data/relationship/types”, “extensions_info” :
“http:/ /localhost:7474/db/data/ext”, “node_index” :
“http://localhost:7474/db/data/index/node”, “extensions” : {
}
}
```

У відповідь ми отримаємо JSON-об'єкт з описом URL-адрес інших команд, зокрема для операцій з вузлами та індексами.

Створення вузлів та зв'язків за допомогою REST

Створити вузол або зв'язок за допомогою інтерфейсу Neo4j REST так

само просто, як CouchDB або Riak. Для створення вузла потрібно відредагувати адресу /db/data/node POST-запит, що містить дані у форматі JSON. Буде зручніше, якщо для кожного вузла задати властивість name, це дозволить легко отримати інформацію про вузл: досить просто викликати name.

```
$curl -i -X POST http://localhost:7474/db/data/node \
-H "Content-Type: application/json" \
-d '{"name": "PG Wodehouse", "genre": "British Humour"}'
```

У відповідь сервер поверне шлях до вузла в заголовку Location та метадані вузла в тілі (ми наводимо їх у скороченому вигляді). Всі ці дані можна прочитати, відправивши GET-запит на адресу, вказану в заголовку Location (або у властивості self в метаданих).

```
HTTP/1.1 201 Created
Location: http://localhost:7474/db/data/node/9

Content-Type: application/json

{
"outgoing_relationships":
"http://localhost:7474/db/data/node/9/relationships/out",
"data": {
"genre": "British Humour", "name" : "PG Wodehouse"
},
"traverse":
"http://localhost:7474/db/data/node/9/traverse/{returnType}",
"all_typed_relationships" :
"http://localhost:7474/db/data/node/9/relationships/all/{-
list|&|types}", "property":
"http://localhost:7474/db/data/node/9/properties/{key}", "self" :
"http://localhost:7474/db/data/node/9",
"properties": "http://localhost:7474/db/data/node/9/properties",
"outgoing_typed_relationships" :
"http://localhost:7474/db/data/node/9/relationships/out/{-
list|&|types}", "incoming_relationships":
"http://localhost:7474/db/data/node/9/relationships/in", "extensions": {
},
"create_relationship":
"http://localhost:7474/db/data/node/9/relationships",
"paged_traverse":
"http://localhost:7474/db/.../{returnType}{?pageSize,leaseTime}",
"all_relationships":
"http://localhost:7474/db/data/node/9/relationships/all",
"incoming_typed_relationships":
"http://localhost:7474/db/data/node/9/relationships/in/{-list|&|types}"
}
```

Якщо потрібні лише властивості вузла (не метадані), то відправте GET-запит на адресу, отриману дописуванням рядка / properties в кінці URL вузла. А якщо дописати ще й ім'я конкретної якості, то буде повернено його

значення.

```
$ curl http://localhost:7474/db/data/node/9/properties/genre “British Humour”
```

Одного вузла нам буде замало, тому додайте ще один із властивостями `[“name”: "JeeveTsakes Charge", "style" : "short story"]`. Оскільки оповідання «Дживз бере кермо влади у свої руки» (Jeeves Takes Charge) написав П. Дж. Вудхауз, між цими вузлами можна встановити зв'язок.

```
$ curl -i -X POST http://localhost:7474/db/data/node/9/relationships \
-H “Content-Type: application/json” \
-d '{“to”: “http://localhost:7474/db/data/node/10”, “type”: “WROTE”, “data”: {“published”: “November 28, 1916”} }'
```

REST-інтерфейс має чудову особливість: він заздалегідь, у властивості create_relationship повернутих в тілі метаданих, повідомляє, як слід створювати зв'язок. Таким чином, REST-інтерфейси є самодокументованими.

Знаходження шляху

За допомогою інтерфейсу REST можна знайти шлях між двома вузлами, відправивши POST-запит на URL /paths початкового вузла. У тілі запиту повинен бути рядок у форматі JSON, що описує, в який вузол йти, по яких ребрах слідувати і який алгоритм пошуку шляху використовувати.

Нехай, наприклад, потрібно знайти шляхи по ребрах типу WROTE з вузла 1, використовуючи алгоритм shortestPath і обмежуючись глибиною 10.

```
$ curl -X POST http://localhost:7474/db/data/node/9/paths \
-H “Content-Type: application/json” \
-d '{“to”:”http://localhost:7474/db/data/node/10”, “relationships”: {“type” : “WROTE”}, “algorithm”:”shortestPath”, “max_depth ”:10}'
[ {
"start" : "http://localhost:7474/db/data/node/9", "nodes" : [
“http://localhost:7474/db/data/node/9”,
“http://localhost:7474/db/data/node/10”
],
"length" : 1,
“relationships” : [ “http://localhost:7474/db/data/relationship/14” ],
“end” : “http://localhost:7474/db/data/node/10”
} ]
```

Допустимі також алгоритми allPaths, allSimplePaths і dijkstra. Докладні відомості про ці алгоритми можна знайти в онлайновій документації (http://api.neo4j.org/current/org/neo4j/graphalgo/GraphAlgoFactory.html), але їх детальний розгляд виходить за рамки цього курсу.

Індексування

Як і інші розглянуті бази даних, Neo4j підтримує швидкий пошук індексів. Однак тут є одна тонкість. На відміну від інших СУБД, де запит формулюється однаково незалежно від того, побудовані індекси чи ні, Neo4j застосовує інший підхід. Пов'язано це про те, що індексування – це окрема служба.

Найпростішим з усіх індексів є хеш-індекс, що складається з пар ключ-значення. У ролі ключа виступають якісь дані, що зберігаються у вузлі, а в ролі значення – REST-сумісний URL, що вказує на вузол у графі. Кількість індексів не обмежена. Назвемо індекс, що цікавить нас, «authors». У кінець URL помістимо ім'я автора, що індексується, а в якості значення передамо вузол 9 (той, в якому зберігаються відомості про Вудхауз, на вашій машині номер може відрізнятися).

```
$ curl -X POST http://localhost:7474/db/data/index/node/authors \
-H “Content-Type: application/json” \
-d '{ “uri” : “http://localhost:7474/db/data/node/9”, “key” : “name”,
“value” : “PG+Wodehouse”}'
```

Для вибірки вузла потрібно просто звернутися до індексу, причому у відповідь ви отримаєте не заданий URL, а самі дані вузла.

```
$ curl
http://localhost:7474/db/data/index/node/authors/name/PG+Wodehouse
```

Крім хеш-індексів, Neo4j підтримує інвертовані індекси для повнотекстового пошуку, що дають змогу пред'являти запити виду: «Дай мені всі книги, назви яких починаються зслова 'Jeeves'». Для побудови такого індексу необхідний весь набір даних, а чи не окрема запис, як у попередньому прикладі. Як і Riak, Neo4j будує інвертований індекс за допомогою Lucene.

```
$ curl -X POST http://localhost:7474/db/data/index/node \
-H “Content-Type: application/json” \
-d '{“name”:”fulltext”,
“config”:{“type”:”fulltext”,”provider”:”lucene”}}'
```

Цей POST-запит повертає JSON-об'єкт, що містить відомості про доданий індекс.

```
{
"template":
“http://localhost:7474/db/data/index/node/fulltext/{key}/{value}”, “provider”
: “lucene”,
"type": “fulltext”
}
```

Додати Вудхауза до повнотекстового індексу можна наступним чином:

```
curl -X POST http://localhost:7474/db/data/index/node/fulltext \
-H “Content-Type: application/json” \
-d '{ “uri” : “http://localhost:7474/db/data/node/9”, “key” : “name”, “value” : “PG+Wodehouse”}'
```

Тепер для пошуку використовується синтаксис запитів Lucene, а як шлях вказується URL-індексу.

```
$ curl http://localhost:7474/db/data/index/node/fulltext?query=name:P*
```

Індекси можна будувати і по ребрах, достатньо замінити в URL слово node словом relationship, наприклад:

```
http://localhost:7474/ db/data/index/relationship/published/date/1916-11-28.
```

REST та Gremlin

Для REST-інтерфейсу до Neo4j є модуль Gremlin, що підключається (у тій версії, з якою ми працюємо, він встановлюється за умовчанням3). Через інтерфейс REST можна посилати будь-які команди, які можна виконати в консольній оболонці Gremlin. Тим самим ви отримуєте у своє розпорядження міць та гнучкість обох інструментів. Це чудова комбінація, оскільки Gremlin найкраще підходить для складних запитів, а REST забезпечує гнучкість розгортання та мовну незалежність.

Наступний запит повертає імена всіх вершин. Потрібно лише відправити дані на URL модуля, що підключається у вигляді JSON-рядка в поле script.

```
$ curl -X POST
\http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script \
-H "content-type:application/json" \
-d '{“script”:”gVname”}'
[ "PG Wodehouse", "Jeeves Takes Charge"]
```

Хоча, починаючи з цього моменту, ми будемо використовувати Gremlin, не забувайте, що з тим самим успіхом можна було б зупинитися на REST.

Великі дані

До цього часу ми мали справу з маленькими наборами даних, тому варто дізнатися, як Neo4j поведеться, коли обсяг даних великий.

Досліджуємо набір даних про кінофільми, які можна завантажити із

сайту Freebase.com. Ми будемо працювати з набором “performance”, в якому поля розділені знаками табуляції (http://download.freebase.com/datadumps/latest/browse/film/performance.tsv). Скачайте цей набір і виконайте показаний нижче скрипт, який послідовно читає рядки і створює зв'язки між новими або існуючими вузлами (збіги шукаються на ім'я в індексі).

Для запуску цього Ruby-скрипту вам знадобляться gem-пакети `json` та `faraday`.

neo4j/importer.rb

```
REST_URL = 'http://localhost:7474/'
HEADER = { 'Content-Type' => 'application/json' }

%w{rubygems json cgi faraday}.each{|r| require r}

# підключитися до REST-сервера Neo4j
conn = Faraday.new(:url => REST_URL) do |builder| builder.adapter
:net_http
end

# цей метод шукає існуючий вузол в індексі або створює новий
def get_or_create_node(conn, index, value)
# шукаємо вузол в індексі
r = conn.get(“/db/data/index/node/#{index}/name/#{CGI.escape(value)}”)
node = (JSON.parse(r.body).first || { })['self'] if r.status == 200 unless
node
# в індексі вузол не знайдено, створюємо новий
r = conn.post(“/db/data/node”, JSON.unparse({“name” => value}), HEADER)
node = (JSON.parse(r.body) || {})['self '] if [200, 201].include? r.status
# додаємо новий вузол в індекс
node_data = "{\"uri\" : \"#{node}\", \"key\" : \"name\",
\"value\" : \"#{CGI.escape(value)}\"}”
conn.post(“/db/data/index/node/#{index}”, node_data, HEADER)
end node
end
puts "починається обробка..." count = 0
File.open(ARGV[0]).each do |line|
_, _, actor, movie = line.split("\t") next if actor.empty? ||
movie.empty?
# будуємо вузли актора та фільму
actor_node = get_or_create_node(conn, 'actors', actor) movie_node =
get_or_create_node(conn, 'movies', movie)

# створюємо зв'язок між актором та фільмом
conn.post(“#{actor_node}/relationships”,

JSON.unparse({ :to => movie_node, :type => 'ACTED_IN' }), HEADER)
puts “завантажено зв'язків: #{count}” if (count += 1) % 100 == 0
end
puts “готово!”
```

Написавши скрипт, запустіть його, вказавши на завантажений файл

performance.tsv.

```
$ruby importer.rb performance.tsv
```

Для обробки всього файлу даних може знадобитися кілька годин, але процес можна будь-якої миті перервати, обмежившись неповним списком фільмів та акторів. Якщо ви працюєте з версією Ruby 1.9, можна прискорити обробку, замінивши рядок `builder.adapter :net_http` рядком `builder.adapter :em_synchrony`, який створює неблокуюче з'єднання.

Алгоритми Gremlin

Завантаживши великий набір даних про фільми, ми на якийсь час залишимо REST-інтерфейс і повернемося до Gremlin.

Реалізуємо один із найвідоміших алгоритмів на графах: шість кроків до Кевіна Бейкона. Цей алгоритм ґрунтується на грі, сенс якої – не більше ніж за 6 переходів знайти найкоротшу відстань між загаданим актором та Кевіном Бейконом через акторів, разом із якими вони знімалися. Наприклад, Алек Гіннес знімався у фільмі «Кафка» з Терезою Рассел, а та – у фільмі «Дикість» з Кевіном Бейконом.

Для початку відкрийте консоль Gremlin та підключіться до графа.Далі створіть крок costers, код якого наведено нижче. Він схожий на створений вчора крок friendsuggest – шукає акторів, що знімалися в тих самих фільмах, що актор у вказаному вузлі (тобто розташованих на іншому кінці будь-якого ребра, що виходить із вузлів фільмів, пов'язаних із початковим вузлом актора).

```
neo4j/costars.groovy Gremlin.defineStep( 'costars',
[Vertex, Pipe],
{
_().sideEffect{start = it}.outE('ACTED_IN').

inV.inE('ACTED_IN').outV.filter{
!start.equals(it)
}.dedup
}
)
```

У Neo4j ми не «запитуємо» множину значень, а «обходимо» граф.

Чарівність цієї ідеї в тому, що зазвичай першим відвіданим буде вузол, найближчий до початкового (у термінах кількості проміжних ребер, а не виваженої відстані). Почнемо з пошуку початкового та кінцевого вузла.

```
gremlin> bacon = gVfilter{it.name=='Kevin Bacon'}.next() gremlin> elvis
= gVfilter{it.name=='Elvis Presley'}.next()
```

Далі ми шукаємо акторів, що знімалися в одному фільмі з початковим, потім - тих, хто знімався в одному фільмі з кожним зі знайдених на попередньому кроці, і т.д. якщо не знайдете, можете повторити пошук, збільшивши кількість кроків). Після чотирьох ітерацій циклу ми знайдемо всіх акторів, які «відстоять на чотири кроки». Скористаємося раніше створеним кроком costers.

```
elvis.costars.loop(1){it.loops < 4}
```

Слід залишити лише вершини на шляху до Бейкона, а решту відкинути.

```
elvis.costars.loop(1){ it.loops < 4
}.filter{it.equals(bacon)}
```

Щоб не потрапляти у вузол Кевіна Бейкон двічі, ми достроково перервемо цикл, опинившись у цьому вузлі. Іншими словами, цикл закінчується після чотирьох ітерацій або коли виявлено вузол bacon. Після цього можна вивести шляхи, якими ми потрапили у вузол bacon.

```
elvis.costars.loop(1){
it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths
```

Тепер залишилося лише взяти зі списку можливих шляхів перший елемент – це буде найкоротший шлях, обчислений раніше за всіх інших. Оператор >> таки витягує перший елемент зі списку.

```
(elvis.costars.loop(1){
it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths >> 1)
```

І нарешті, ми отримуємо ім'я кожної вершини та відфільтровуємо порожні ребра, користуючись командою Groovy grep.

```
(elvis.costars.loop(1){
it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths >> 1).name.grep{it}
==>Elvis Presley
==>Double Trouble
==>Roddy McDowall
==>The Big Picture
==>Kevin Bacon
```

Ми не знали, хто такий Родді Макдауелл, але в цьому і полягає краса

графової бази даних. Нам і не треба було цього знати, щоб отримати правильну відповідь. Можете повправлятися з Groovy, якщо хочете вивести не просто список, а щось більш витончене, але самі дані ми вже отримали.

Випадковий обхід

Коли потрібна представницька вибірка з великого набору даних, зручно скористатися «випадковим обходом». Спершу створюємо генератор випадкових чисел.

```
rand = new Random()
```

Потім відфільтровуємо необхідну частку повної множини документів. Якщо, наприклад, потрібно повернути лише близько третини приблизно від 60 фільмів,у яких зіграв Кевін Бейкон, можна залишити лише ті, котрим випадкове число виявилося менше 0.33.

```
bacon.outE.filter{rand.nextDouble() <= 0.33}.inV.name
```

В результаті має вийти приблизно 20 випадкових назв фільмів.

Якщо взяти акторів, що віддаляються на два кроки від Кевіна Бейкона, то вийде величезний список (для нашого набору даних він містить понад 300 000 імен).

```
bacon.outE.inV.inE.outV.loop(4){ it.loops < 3
}.count()
==> 316198
```

А щоб залишити приблизно один відсоток із цього списку, додамо фільтр. Але не забудьте, що сам фільтр є кроком, тому параметр методу loop слід збільшити на 1.

```
bacon.outE{ rand.nextDouble() <= 0.01

}.inV.inE.outV.loop(5){ it.loops < 3
}.name
```

Ми отримали Еліа Вуда. Очікується, що наш алгоритм пошуку акторів, близьких до Бейкона, поверне його через два кроки. І це справді так: Еліа Вуд грав у фільмі «Зіткнення з безоднею» разом із Роном Елдардом, а той грав у «Сплячих» разом із Бейконом.

Центрованість

Центрованістю називається міра відносної важливості вузла у графі. Наприклад, якщо ми хочемо виміряти важливість вузла в мережі, виходячи з

його відстані до всіх інших вузлів, то буде потрібно алгоритм обчислення центрованості.

Мабуть, найвідомішим алгоритмом обчислення центрованості є Google PageRank, але є кілька варіантів. Ми реалізуємо простий варіант центрованості власного вектора, в якому просто обчислюється кількість ребер, що входять у вузол або з нього. Ми зіставимо кожному актору кількість зіграних ним ролей.

Нам необхідний асоціативний масив, який міг би заповнити функція groupCount(), і змінна count, що відраховує кількість ітерацій, щоб вона перевищувала заданого порога.

```
role_count = [:]; count = 0 gVin.groupCount(role_count).loop(2){ count++ < 1000 }; ''
```

Ключами асоціативного масиву role_count будуть вершини, а значеннями – число ребер, інцидентних даній вершині. Для інтерпретації результату масив краще відсортувати.

```
role_count.sort{a,b -> a.value <=> b.value}
```

Останнім буде актор, із найдовшим послужним списком.У нашому наборі ця честь належить легендарному акторові озвучення Мелу Бланку, який набрав 424 пункти (всі їх можна переглянути, виконавши запит gVfilter{it.name=='Mel Blanc'}.out. name).

Зовнішні алгоритми

Написати свій алгоритм, звичайно, цікаво, але здебільшого ця робота вже кимось зроблена. Каркас Java Universal Network/ Graph (JUNG) є колекцією стандартних алгоритмів на графах та інших засобів для моделювання та візуалізації графів. Завдяки проєкту Gremlin/Blueprint стало нескладно отримати доступ до таких алгоритмів з JUNG, як PageRank, HITS, Voltage, алгоритми обчислення центрованості та інструментів представлення графа у вигляді матриці.

Для використання JUNG необхідно обернути граф Neo4j, перетворивши його на граф JUNG5. Для доступу до графа JUNG у нас є дві можливості: завантажити та встановити всі jar-файли Blueprint та JUNG у

каталог libs сервера Neo4j та перезапустити сервер або завантажити вже конфігуровану консоль Gremlin. Ми рекомендуємо другий спосіб, оскільки це позбавить вас необхідності вишукувати в мережі різні jar-файли.

У припущенні, що ви завантажили консоль gremlin, зупиніть сервер neo4j і запустіть Gremlin. Вам знадобиться створити об'єкт Neo4jGraph, передавши його конструктору шлях до підкаталогу data/graph установочного каталогу.

```
g = new Neo4jGraph('/users/x/neo4j-enterprise-1.7/data/graph.db')
```

Ми, як і раніше, позначати граф Gremlin буквою g. Об'єкт Neo4jGraph необхідно обернути об'єктом GraphJung, який ми назвемо j.

```
j = new GraphJung(g)
```

Кевін Бейкон був обраний за «центр» завдяки його грі у фільмах з іншими популярними зірками. Але взагалі йому зовсім необов'язково було грати багато ролей самому, важливо просто бути пов'язаним з тими, хто пов'язаний з багатьма іншими.

Але тоді виникає питання: а чи не можна знайти актора, який з точки зору відстані до інших акторів був би кращим за Кевіна Бейкона?

У JUNG є алгоритм оцінювання BarycenterScorer, який надає кожній вершині оцінку, з її відстані до решти вершин. Якщо Кевін Бейкон – справді найкращий вибір, його оцінка має бути найменшою, тобто він «найближче» до решти акторів.

Наш алгоритм із каркасу JUNG має застосовуватись тількидо акторів, тому сконструюємо трансформатор, який відкидатиме всі вузли, крім акторів. Клас EdgeLabelTransformer передає алгоритму лише вузли, з яких виходять ребра типу ACTED_IN.

```
t = new EdgeLabelTransformer(['ACTED_IN'] as Set, false)
```

Потім потрібно імпортувати сам алгоритм та створити його екземпляр, передавши конструктору граф GraphJung та трансформатор.

```
import edu.uci.ics.jung.algorithms.scoring.BarycenterScorer barycenter = new BarycenterScorer<Vertex,Edge>( j, t )
```

Тепер можна запитати у BarycenterScorer оцінки всіх вузлів. Знайдемо, чому дорівнює оцінка Кевіна Бейкона.

```
bacon = gVfilter{it.name=='Kevin Bacon'}.next() bacon_score =
barycenter.getVertexScore(bacon)
~0.0166
```

Отримавши оцінку Бейкон, ми можемо обійти всі вершини графа і зберегти ті, для яких оцінка менша.

```
connected = [:]
```

Застосування алгоритму BarycenterScorer до кожного актору в базі даних може тривати досить тривалий час. Тому вчинимо інакше – застосуємо його лише до акторів, які грали з Бейконом в одному фільмі. Це може тривати кілька хвилин, все залежить від потужності обладнання. Сам алгоритм BarycenterScorer працює швидко, але його треба застосувати до багатьох акторів.

```
bacon.costars.each {
score = b.getVertexScore(it); if(score <bacon_score) {
connected[it] = score;
}
}
```

Усі ключі, що опинилися в асоціативному масиві connected, представляють акторів із кращою оцінкою, ніж у Кевіна Бейкона. Але добре було б побачити когось нам знайомого, тому виведемо всіх і виберемо того, хто більше сподобається. На вашій машині результат може відрізнятись, тому що відкрита для загального користування база даних постійно змінюється.

```
connected.collect{k,v -> k.name + " => " + v}
==> Donald Sutherland => 0.00925
==> Clint Eastwood => 0.01488
...
```

Дональд Сазерленд увійшов до списку з оцінкою ~0.00925. Тому гіпотетично зіграти з друзями у гру «Шість кроків до Дональда Сазерленда» було б простіше, ніж у традиційних «Шість кроків до Кевіна Бейкона». Маючи граф j, ми можемо виконати над нашим набором даних будь-який алгоритм, наявний у JUNG, наприклад PageRank. Як і у випадку BarycenterScorer, спочатку потрібно імпортувати клас.

```
import edu.uci.ics.jung.algorithms.scoring.PageRank pr = new
PageRank<Vertex,Edge>( j, t, 0.25d )
```

Повний перелік наявних у JUNG алгоритмів можна знайти в онлайн документації у форматі Javadoc, куди постійно додаються нові алгоритми.

Висновок

Отже, ми дізналися про нові способи взаємодії з базою даних Neo4j, познайомившись із її REST-інтерфейсом. Ми бачили, як за допомогою модуля Gremlin, що підключається, виконувати написаний на Gremlin код на сервері і отримувати через REST-інтерфейс результати. Ми поекспериментували з великим набором даних та закінчили коротким оглядом кількох алгоритмів для дослідження цих даних.

Neo4j – один із найкращих зразків графових баз даних з відкритим вихідним кодом. Такі бази чудово підходять для зберігання неструктурованих даних - іноді навіть краще, ніж документні сховища. Мало того, що в Neo4j немає ні типів, ні схеми, вона ще й не накладає жодних обмежень на взаємозв'язки між даними. Поточна версія Neo4j підтримує 34,4 мільярда вузлів і стільки ж зв'язків – більш ніж достатньо більшості завдань (в одному графі Neo4j можна було б зберігати більше 42 вузлів для кожного з 800 мільйонів користувачів Facebook).

Завдання для індивідуальної роботи

1. Знайдіть прив'язку до інтерфейсу REST для своєї улюбленої мови програмування.
2. Перетворіть частину пошуку шляхів в алгоритмі «Шість кроків до Кевіна Бейкона» на окремий крок. Потім напишіть на Groovy універсальну функцію (наприклад, def actor_ path(g, name1, name2) {…}), яка приймає граф та два імені акторів і порівнює відстані від них до Бейкона.
3. Виберіть який-небудь з численних алгоритмів JUNG і застосуйте його до вузла (або до набору даних, якщо того вимагає API).
4. Установіть драйвер на свій вибір і скористайтеся ним для управління графом обраної компанії, в якому вершини представляють людей та його ролі, а ребра – робочі відносини (підпорядковується, працює разом). Якщо компанія дуже велика, обмежтеся працюючими поряд групами; якщо дуже мала, спробуйте увімкнути ще й клієнтів. Знайдіть у організації людину з «найкращими зв'язками», тобто з найменшою відстанню до інших вершин.

Лекція 14. Розподіленість та висока доступність Neo4j

Нарешті розглянемо, як пристосувати Neo4j до вирішення критично важливих завдань. Ми побачимо, що Neo4j може забезпечити надійне зберігання даних за допомогою ACID-сумісних транзакцій. Потім ми встановимо та налаштуємо високодоступний кластер Neo4j, щоб збільшити доступність під час обслуговування великого потоку запитів на читання. І, нарешті, розглянемо стратегії резервного копіювання.

Транзакції

Neo4j – база даних з підтримкою атомарних, несуперечливих, ізольованих та довговічних (ACID) транзакцій, як і PostgreSQL. Це робить її придатною для зберігання важливих даних, коли зазвичай вибирається реляційна СУБД. Як і завжди, у транзакціях Neo4j реалізовано принцип «все чи нічого». Або успішно виконуються всі операції, що входять до складу транзакції, або жодна з них.

Механізм обробки транзакцій знаходиться не на рівні Gremlin, а на нижчому – у проєкті Blueprint, що обгортає Neo4j. Конкретні деталі можуть змінюватись від версії до версії. Ми використовуємо версію Gremlin 1.3, яка базується на Blueprint 1.0. Якщо у вас встановлена інша версія того чи іншого продукту, то шукайте подробиці в документації Blueprint API.

Як і в PostgreSQL, прості однорядкові функції автоматично поринають у неявну транзакцію. Щоб продемонструвати багаторядкові транзакції, ми повинні відключити режим автоматичних транзакцій для об'єкта графа, давши Neo4j знати, що ми збираємося обробляти транзакції вручну. Для зміни режиму транзакцій функція setTransactionMode().

```
gremlin> g.setTransactionMode(TransactionalGraph.Mode.MANUAL)
```

Для початку та завершення транзакції у графі призначені методи startTransaction() та stopTransaction(conclusion). Завершуючи транзакцію, необхідно вказати, чи вона була виконана успішно. Якщо ні, то Neo4j відкотить усі команди, виконані з початку транзакції. Рекомендуємо укладати

транзакцію в блок try/catch, щоб усі винятки гарантовано призводили до відкату.

```
      g.startTransaction() try {
      // Виконати кілька операцій над графом...
g.stopTransaction(TransactionalGraph.Conclusion.SUCCESS)
      } catch(e) { g.stopTransaction(TransactionalGraph.Conclusion.FAILURE)
      }
```

Якщо ви готові вийти за рамки Gremlin і працювати безпосередньо з об'єктом Neo4j EmbeddedGraphDatabase, то можете використовувати синтаксис транзакцій, визначений Java API. Це може знадобитися, якщо ви пишете код Java або мовою, побудованою на базі віртуальної машини Java, наприклад JRuby.

```
      r = g.getRawGraph() tx = r.beginTx() try {
      // Виконати кілька операцій над графом ... tx.success ()
      } finally { tx.finish()
      }
```

За будь-якого варіанта ви отримуєте всі гарантії ACID. Навіть у випадку збою системи всі операції запису будуть відкачені після перезапуску сервера. Якщо ручне керування транзакціями не потрібно, краще залишити режим TransactionalGraph.Mode.AUTOMATIC.

Висока доступність

Режим високої доступності – це відповідь Neo4j на запитання: «Чи здатна графова база до масштабування»? Так, але з деякими обмеженнями. Запис на один підлеглий сервер не відразу синхронізується з іншими підлеглими серверами, тому протягом невеликого проміжку часу узгодженість (тобто теореми CAP) може бути відсутнім (але відновиться зрештою). Високодоступний (HA) кластер не гарантує властивостей ACID у повному обсязі. З цієї причини HA-кластери Neo4j розглядаються в основному як рішення, покликане підвищити кількість обслуговування запитів на читання. Як і в Mongo, кластери, що входять в сервер, вибирають головний вузол, на якому зберігається «золота копія» даних. Але, на відміну від Mongo, на підлеглі сервери можна писати. Ці операції запису будуть синхронізовані з головним вузлом, який поширить зміни на інші підпорядковані вузли.

HA-кластер

Для роботи з механізмом високої доступності Neo4j ми повинні спочатку налаштувати кластер. У Neo4j використовується зовнішня служба координації кластера Zookeeper – ще один прекрасний проєкт, що відбрунькувався від Apache Hadoop. Це універсальна служба для координації розподілених програм. У кластері Neo4j вона застосовується для керування життєвим циклом. З кожним сервером Neo4j пов'язаний окремий координатор, завданням якого є управління його місцем у кластері, як показано на рис. 36.

Однак, до складу Neo4j Enterprise вже входить Zookeeper, а також низка файлів для налаштування кластера. Ми збираємося запустити три екземпляри сервера Neo4j Enterprise версії 1.7. Завантажити дистрибутив для своєї операційної системи можна із сайту http://neo4j.org/download/ . Потім розпакуйте архів та створіть ще дві копії каталогу. Ми включили до складу їхніх імен цифри 1, 2, 3 і так і надалі посилатимемося на них.

```
tar fx neo4j-enterprise-1.7-unix.tar
mv neo4j-enterprise-1.7 neo4j-enterprise-1.7-1
cp-R neo4j-enterprise-1.7-1 neo4j-enterprise-1.7-2 cp-R neo4j-
enterprise-1.7-1 neo4j-enterprise-1.7-3
```

Теперу нас є три ідентичні копії бази даних.

Зазвичай слід встановити лише одну копію на кожен сервер і налаштувати кластер так, щоб усі сервери знали один про одного. Але оскільки ми запускаємо всі сервери на одній машині, то рознесли їх по різних каталогах і призначили різні порти.

Для створення кластера потрібно виконати п'ять кроків, розпочавши конфігурацію координаторів кластера Zookeeper і закінчивши конфігуруванням самих серверів Neo4j.

1. Присвоїти кожному координатору унікальний ідентифікатор.
2. Налаштувати кожен координатор, щоб він міг спілкуватися з іншими координаторами і зі своїм сервером Neo4j.
3. Запустити всі три сервери-координатори.

4. Налаштувати кожен сервер Neo4j для роботи в режимі високої доступності, призначити їм унікальні порти та повідомити про наявність координаторів.

5. Запустити всі три сервери Neo4j.

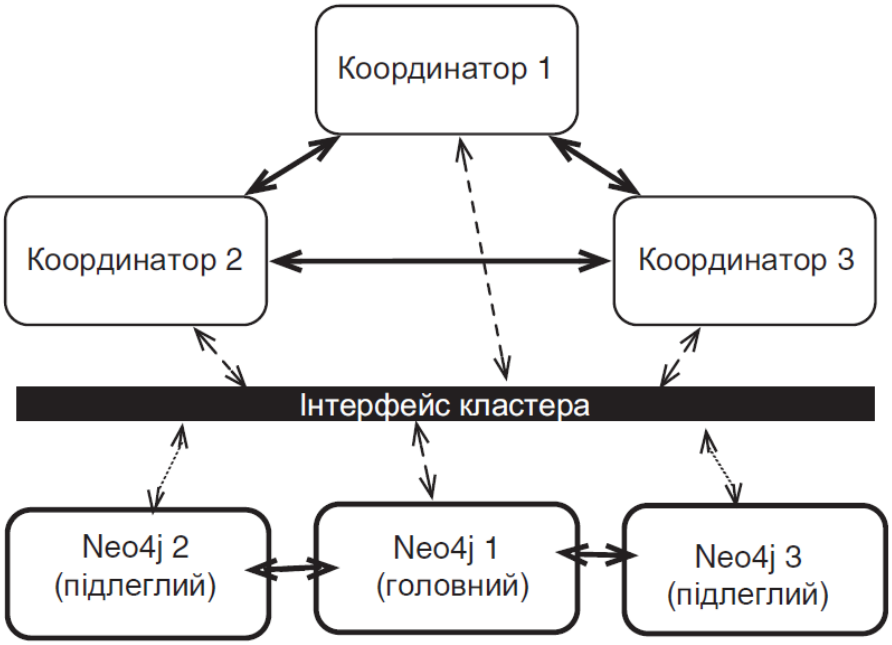


Рис. 36.Кластер із трьох серверів Neo4j та їх координаторів

Zookeeper відслідковує сервери за унікальним у межах кластера ідентифікатором. Це – єдине, що зберігається у файлі data/coordinator/myid. Для сервера 1 залишимо на увазі за замовчуванням значення 1, для сервера 2 задаємо ідентифікатор 2, а для сервера 3 - ідентифікатор 3.

```
echo “2” > neo4j-enterprise-1.7-2/data/coordinator/myid echo “3” >
neo4j-enterprise-1.7-3/data/coordinator/myid
```

Необхідно також встановити деякі внутрішні для кластера комунікаційні параметри. Кожен сервер має мати свій файл conf/coord.cfg. За замовчуванням у змінну server.1 записується ім'я сервера localhost і номери двох портів: для вибору кворуму (2888) і для вибору головного вузла (3888).

Побудова кластера

Кворумом Zookeeper називається група серверів і портів, що входять до кластеру, через які вони взаємодіють (не плутати з кворумом в Riak, де цим терміном позначається мінімальна більшість, необхідна для забезпечення узгодженості). Порт для вибору головного вузла використовується, коли

головний вузол виходить з ладу, - він потрібен для того, щоб сервери, що залишилися, могли вибрати новий головний вузол. Ми залишимо змінну server.1 без зміни та додамо змінні server.2 та server.3, вказавши наступні по порядку номери портів. Файли coord.cfg в каталогах серверів 1, 2 і 3 повинні містити ті самі рядки.

```
server.1=localhost:2888:3888 server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

Зрештою, ми повинні визначити порт, через який можна підключатися до сервера Neo4j. За замовчуванням вибирається clientPort 2181, таким ми його залишимо для сервера 1. Для сервера 2 поставимо clientPort=2182, а для сервера 3 – clientPort=2183. Якщо якісь із цих портів на вашій машині вже зайняті, можете вибрати будь-які інші, але ми далі припускатимемо, що порти задані саме так.

Координатори

Для запуску координатора Zookeeper ми скористаємося скриптом, люб'язно наданим розробниками Neo4j. Виконайте наступну команду в кожному з трьох каталогів серверів:

```
bin/neo4j-coordinator start
Starting Neo4j Coordinator...WARNING: не змінює програму користувача
[36542]... waiting for coordinator to be ready. OK.
```

Координатори тепер працюють, але Neo4j ще немає.

Запускаємо Neo4j

Далі ми повинні налаштувати сервер Neo4j для роботи в режимі високої доступності та підключити його до сервера-координатора. Відкрийте файл conf/neo4j-server.properties та додайте наступний рядок (для кожного сервера):

```
org.neo4j.server.database.mode=HA
```

Тиким чином ми говоримо, що сервер Neo4j повинен працювати в режимі високої доступності; і досі ми працювали в режимі SINGLE. Також потрібно призначити порти веб-серверам. Зазвичай порт 7474, що мається на увазі, цілком годиться, але якщо ми запускаємо три екземпляри neo4j на одній машині, то треба їх розвести по портах протоколів http/https. Серверу 1

призначимо порти 7471/7481, серверу 2 - 7472/7482, а серверу 3 - 7473/7483.

```
org.neo4j.server.webserver.port=7471
org.neo4j.server.webserver.https.port=7481
```

Нарешті, з'єднаємо кожен екземпляр Neo4j з відповідним координатором. Відкривши файл conf/neo4j.properties для сервера 1, ви знайдете кілька закоментованих рядків, що починаються словом ha. Це параметри режиму високої доступності: номер машини даного сервера в кластері, список серверів Zookeeper і порт, яким сервери neo4j можуть спілкуватися між собою. Для сервера 1 додайте файл neo4j.properties до таких параметрів:

```
ha.server_id=1
ha.coordinators=localhost:2181,localhost:2182,localhost:2183
ha.server=localhost:6001
ha.pull_interval=1
```

Параметри двох інших серверів аналогічні з однією відмінністю: для сервера 2 ha.server_id=2, а для сервера 3 ha.server_id=3. Крім того, значення ha.server також повинні відрізнятися (ми вибрали для сервера 2 порт 6002, а для сервера 3 порт 6003). Ще раз наголосимо, що при запуску серверів на різних машинах змінювати номери портів необов'язково. Таким чином, файл для сервера 2 (і аналогічно для сервера 3) має виглядати так:

```
ha.server_id=2
ha.coordinators=localhost:2181,localhost:2182,localhost:2183
ha.server=localhost:6002
ha.pull_interval=1
```

Ми задали pull_interval рівним 1, це означає, що кожен підлеглий сервер повинен перевіряти наявність оновлень на головному раз на секунду. Взагалі, задавати таке маленьке значення не варто, але ми хочемо бачити оновлення даних у прикладі відразу після їх вставки.

Налаштувавши сервери Neo4j для високодоступного кластера, ми можемо їх запустити. Запускайте кожен сервер neo4j з його власного інсталяційного каталогу.

```
bin/neo4j start
```

Щоб стежити за повідомленнями сервера, виконайте команду tail, вказавши файл журналу.

```
tail -f data/log/console.log
```

Кожен сервер приєднується до призначеного йому координатора.

Перевірка стану кластера

Координатор, запущений першим, - ймовірно, сервер 1 - стає головним сервером. Щоб переконатися в цьому, відкрийте адміністративний веб-інтерфейс приєднаного екземпляра Neo4j (Ми призначили серверу 1 порт 7471). Натисніть на посилання Server Info (Відомості про сервер) у верхній частині сторінки, а потім на посилання High Availability (Висока доступність) у бічному меню.

Властивості у списку High Availability містять інформацію про кластер. Якщо даний сервер є головним, відповідна властивість дорівнюватиме true. В іншому випадку можна визначити, який сервер обраний головним, глянувши на список InstancesInCluster, де перераховані всі підключені сервери із зазначенням ідентифікатора машини, ознаки головного сервера та іншої інформації.

Перевірка результатів реплікації

Запустивши кластер, ми можемо перевірити, чи сервери реплікуються правильно. Якщо все піде за планом, то будь-яка операція запису на підлеглий сервер має бути реплікована на головний, а звідти на всі інші підлеглі сервери. Відкривши веб-консолі для кожного із трьох серверів, ми зможемо скористатися вбудованою в адміністративний інтерфейс консоллю Gremlin. Зверніть увагу, що об'єкт графа Gremlin тепер обгортає об'єкт HighlyAvai lableGraphDatabase.

```
g = neo4jgraph[HighlyAvailableGraphDatabase[/…/neo4j-ent-1.7-2/data/graph.db]]
```

Щоб протестувати наші сервери, ми додамо до нового графа кілька вузлів, що містять назви знаменитих парадоксів. На консолі одного з підлеглих серверів запишіть у кореневий вузол феномен Зенона.

```
gremlin> root = g.v (0)
gremlin> root.paradox = "Zeno's"
gremlin> root.save
```

Тепер перейдіть на консоль головного сервера та виведіть значення парадоксу у вершині.

```
gremlin > g.V.paradox
==> Zeno's
```

Перейдіть на консоль іншого підлеглого сервера та додайте парадокс Рассела. Вивівши список, ми виявимо, що на другому підпорядкованому сервері існують обидва вузли, хоча безпосередньо ми додавали лише один.

```
gremlin> g.addVertex(["paradox": "Russell's"])
gremlin> g.V.paradox
==> Zeno's
==> Russell's
```

Якщо на якийсь підлеглий сервер зміни ще не поширилися, поверніться на сторінку Server Info, High Availability. Зверніть увагу на значення ідентифікаторів LastCommittedTransactionId для всіх серверів. Якщо вони збігаються, то дані узгоджені. Чим менше число, тим старіша версія даних на сервері.

Вибір головного сервера

Якщо зупинити головний сервер і оновити інформацію на одному з тих, що залишилися, виявиться, що вибрано новий головний сервер. Якщо знову запустити сервер, то він повернеться до кластера, але залишиться підлеглим (доки новий головний сервер працює).

Завдяки механізму високої доступності сильно навантажені системи реплікують дані на кілька серверів і завдяки цьому поділяють навантаження. Зрештою, дані на всіх серверах кластера будуть узгоджені, але є кілька прийомів, що дозволяють зменшити ймовірність читання застарілих даних у кожний момент часу, наприклад, зв'язування сеансу з одним сервером. Якщо використовувати відповідні інструменти, заздалегідь все спланувати і правильно налаштувати, можна створити графову базу даних, яка містить мільярди вершин і ребер і при цьому обслуговує стільки запитів, скільки вам необхідно. Додайте сюди регулярне резервне копіювання – і ось вам рецепт добротної виробничої системи.

Резервне копіювання

Резервне копіювання – невід'ємна складова будь-якої професійної процедури експлуатації бази даних. Хоча реплікацію можна певною мірою вважати різновидом резервного копіювання, щоденне зняття копії із зберіганням її поза приміщенням обчислювального центру настійно

рекомендується у разі аварійного відновлення. Важко планувати пожежу в серверній або землетрус, який зруйнує будівлю вщент. Видання Neo4j Enterprise включає просту програму резервного копіювання під назвою neo4j-backup.

При експлуатації HA-кластера найправильніший підхід – запускати команду повного резервного копіювання, яка скопіює базу даних з кластера в тимчасовий міткою файл на змонтованому диску. У резервному каталозі буде створено повністю працездатну копію. Якщо виникне необхідність відновлення, достатньо буде замінити каталог даних кожному сервері резервним каталогом – і можна продовжувати роботу.

Починати потрібно з повного резервного копіювання. Наступна команда копіює HA-кластер у каталог, ім'я якого закінчується сьогоднішньою датою (її повертає команда date, яка є у будь-якому варіанті *nix).

```
bin/neo4j-backup -full -from ha://localhost:2181,localhost:2182,\
localhost:2183 -to /mnt/backups/neo4j-`date +%Y.%m.%d`.db
```

Якщо сервер запущений не в режимі високої доступності, просто змініть схему ha в URI на один. Після зняття повної копії ви можете виконувати інкрементне резервне копіювання, зберігаючи лише зміни, що відбулися з моменту створення останньої копії. Якщо потрібно виконати повне копіювання на одному сервері опівночі, а потім інкрементно копіювати зміни кожні дві години, то підійде, наприклад, така команда:

```
bin/neo4j-backup -incremental -from single://localhost \
-to /mnt/backups/neo4j-`date +%Y.%m.%d`.db
```

Однак майте на увазі, що інкрементне копіювання буде працювати лише в тому випадку, якщо у вказаному каталозі вже є повна копія. Тому показана вище команда повинна запускатися того ж дня, коли було знято повну резервну копію.

Ми вивчили механізми забезпечення збереження даних у Neo4j: ACID-сумісних транзакцій, високої доступності та резервного копіювання. Важливо відзначити, що всі розглянуті інструменти включені тільки у видання Neo4j Enterprise, використання якого регламентується схемою

подвійного ліцензування – GPL/AGPL. Якщо ви не бажаєте розкривати вихідний код своєї програми, то повинні перейти на видання Community або отримати статус OEM від компанії Neo Technology, яка розробляє Neo4j.

Висновок

Neo4j - найкраща реалізація графової бази даних (досить рідкісний клас) з відкритим вихідним кодом. У графових базах даних акцент робиться на зв'язках між даними, а не на спільності значень. Моделювати дані у вигляді графів легко. Потрібно лише створити вузли та зв'язки між ними та за бажання асоціювати з ними пари ключ-значення. Запити зводяться до опису обходу графа, вирушаючи з початкового вузла.

У дистрибутив Neo4j входять інструменти для прискорення пошуку (Lucene) і прості у використанні (хоча, можливо, не цілком звичні) мовні розширення, зокрема Gremlin і REST-інтерфейс. Але Neo4j не тільки проста у використанні, а й працює швидко. На відміну від операцій з'єднання в реляційних базах даних або операцій map-reduce в інших базах обхід графа вимагає постійного часу. Пов'язані дані знаходяться лише в одному кроці – не потрібно виробляти потужне з'єднання з наступною фільтрацією результатів, як у більшості розглянутих вище СУБД. Яким би великим не був граф, перехід із вузла A у вузол B вимагає лише одного кроку, якщо між цими вузлами є зв'язок. Нарешті, у видання Enterprise включені засоби для створення високодоступних сайтів з великою кількістю запитів на читання шляхом створення HA-кластера Neo4j.

Слабкі сторони Neo4j. Neo4j не позбавлена недоліків. Ребра в Neo4j не можуть входити в ту саму вершину, з якої виходять (утворювати петлі). Нам також здається, що обраноня термінологія (вузол замість вершини і зв'язок замість ребра) тільки ускладнює взаєморозуміння. HA-кластер чудово справляється з реплікацією, але реплікувати можна лише граф цілком. Неможливо сегментувати граф, що накладає обмеження на його розмір (хоча будемо чесні – обмеження в десятки мільярдів не дуже обтяжливе). Видання Community поставляється за ліцензією GPL, але якщо вам потрібні послуги,

що входять лише у видання Enterprise (високодоступний кластер та резервне копіювання), для побудови виробничої системи, то, ймовірно, за ліцензію доведеться заплатити.

Neo4j та теорема CAP. Для тих, хто збирається будувати розподілену систему, стратегія Neo4j має властивості доступності та стійкості до втрати зв'язності (тобто відноситься до класу AP). Кожен підлеглий сервер повертає тільки ті дані, які має в даний момент, і вони протягом деякого періоду часу можуть відрізнятися від зберігаються в головному вузлі. Затримку оновлення можна скоротити, зменшивши інтервал між опитуваннями на підпорядкованому сервері, але технічно система все одно забезпечує лише узгодженість зрештою. Тому HA-кластер Neo4j рекомендується використовувати тільки в додатках, орієнтованих здебільшого на читання.

Завдання для індивідуальної роботи

1. Дайте відповідь на запитання: «Яка максимальна кількість підтримуваних вузлів?» (Підказка: загляньте в розділ «Питання та відповіді» на сайті документації.)
2. Реплікуйте Neo4J на три фізичні сервери.
3. Налаштуйте балансувальник навантаження на базі якогось веб-сервера, наприклад Apache або Nginx, і підключіться до кластера через REST-інтерфейс. Виконайте якийсь скрипт мовою Gremlin.

Лекція 15. Основи Redis

Redis (REmote DIctionary Service – віддалена служба словників) – просте у роботі сховище ключів та значень із розвиненим набором команд. Перша версія системи вийшла у 2009 році. І у швидкодії у нього практично немає суперників. Читання відбувається швидко, а запис ще швидше - на деяких еталонних тестах продемонстровано до 100 000 операцій SET за секунду. Автор Redis Сальваторе Санфіліппо (Salvatore Sanfilippo) назвав свій проєкт «Сервер структур даних», щоб підкреслити закладені в нього можливості роботи зі складними типами даних та інші особливості. Вивченням цього надшвидкого «не просто сховища ключів та значень» ми завершимо огляд найбільш відомих баз даних NoSQL.

Сховище сервера структур даних

Точно віднести Redis до якоїсь категорії досить важко. На найнижчому рівні це, звичайно, сховище ключів та значень, але це дуже такий спрощений підхід. Redis підтримує складні структури даних, хоч і не настільки, як документо-орієнтовані бази даних. Вона підтримує запити, що повертають множини, але не такого рівня детальності, як реляційні СУБД, і без підтримки типів. І, зрозуміло, вона швидка, хоча задля досягнення цієї мети жертвує довговічністю даних задля швидкодії.

Крім сервера структур даних, Redis є ще й блокуючою чергою (або стеком), а також системою публікації-підписки. У ній можна налаштовувати політики терміну зберігання, рівні довговічності та параметри реплікації. Все це робить Redis скоріше комплектом інструментальних засобів, до якого входять корисні алгоритми роботи зі структурами даних та процесами, аніж базою даних якогось певного жанру.

Великий список клієнтських бібліотек Redis дозволяє використовувати її в багатьох мовах програмування. Робота з нею не тільки проста, а й приносить задоволення.

Почавши вивчення з простих операцій CRUD, ми перейдемо до операцій над складнішими структурами даних: списками, хешами,

множинами та відсортованими множинами. Ми навчимося створювати транзакції та маніпулювати характеристиками, що визначають термін зберігання даних. Ми скористаємося Redis для створення простої черги повідомлень та досліджуємо вбудований механізм публікації-підписки. Потім ми звернемося до питання про налаштування та параметри реплікації Redis і подивимося, як підтримати баланс між довговічністю даних і швидкодією.

Бази даних часто використовуються у поєднанні – і ця тенденція набирає сили. Ми залишили Redis насамкінець, щоб показати його застосування саме в якості системи, в якій будуть задіяні Redis, CouchDB, Neo4J і Postgres, а поєднувати їх буде Node.js.

Операції CRUD та типи даних

Перш за все вивчимо командний інтерфейс Redis дослідивши багато зі 124 доступних команд. Особливий інтерес становлять витончені типи даних та способи їхнього опитування, що далеко виходять за рамки примітивного «отримати значення ключа».

Redis підтримують деякі менеджери пакетів, зокрема Homebrew для Mac, але й зібрати його з вихідного коду теж нескладно1. Ми працюватимемо з версією 2.4. Встановивши її, запустіть сервер командою

```
$ redis-server
```

За замовчуванням він працює не у фоновому режимі, але це можна виправити, додавши до кінця команди знак & або просто відкривши ще один термінал. Потім запустіть командну утиліту, яка автоматично підключиться до порту 6379 за промовчанням. Установивши з'єднання, спробуйте продзвонити сервер.

```
$ redis-cli
redis 127.0.0.1:6379 > PING PONG
```

Якщо неможливо підключитися, буде видано повідомлення про помилку. У відповідь на команду help буде виведено інструкцію з вбудованої довідки. Введіть help, потім пробіл, а потім почніть вводити якусь команду. Якщо ви не знаєте жодної команди Redis, просто натискайте клавішу Tab -

циклічно перебиратиметься список команд.

```
redis 127.0.0.1:6379> help
Type: “help @<group>” get a list of commands in <group> “help <command>” for help on <command>
“help <tab>” натиснути на list possible help topics “quit” to exit
```

Ми скористаємося Redis для побудови серверної частини системи скорочення URL-адрес – на кшталт tinyurl.com або bit.ly. Скорочувач URL-адреси – це служба, яка приймає довгу URL-адресу та зіставляє йому коротку адресу у власному домені, наприклад, `http://www.myveryververylongdomain.com/somelongpath.php` може бути скорочений до `http://bit.ly/VLD`. При переході по короткому URL система переадресує на вихідний URL, позбавляючи користувача необхідності набирати в текстових повідомленнях довгі рядки. Водночас власник служби отримує деяку статистику, зокрема про відвідуваність.

У Redis існує операція SET, яка зіставляє короткий ключ, наприклад 7wks, довге значення, наприклад `http://www.sevenweeks.org`. Ця операція приймає рівно два параметри: ключ та значення. Для отримання значення ключа потрібно виконати операцію GET.

```
redis 127.0.0.1:6379> SET 7wks http://www.sevenweeks.org/OK
redis 127.0.0.1:6379 > GET 7wks
"http://www.sevenweeks.org/"
```

Для зменшення трафіку можна скористатися операцією MSET, яка зіставляє відразу кілька ключів та значень. Так, наступна команда зіставляє значення Google.com ключ gog, а значення Yahoo.com - ключ yah.

```
redis 127.0.0.1:6379> MSET goghttp://www.google.comyahhttp://www.yahoo.com OK
```

І навпаки, MGET отримує на вході кілька ключів та повертає впорядкований список відповідних їм значень.

```
redis 127.0.0.1:6379> MGET gog yah
1) "http://www.google.com/"
2) "http://www.yahoo.com/"
```

Хоча Redis зберігає рядки, вона розпізнає цілі числа і навіть вміє виконувати з ними деякі прості операції. Якщо ми хочемо стежити, скільки коротких ключів зберігається у наборі даних, можемо створити лічильник і збільшувати його за одиницю командою INCR.

```
redis 127.0.0.1:6379> SET count 2 OK
redis 127.0.0.1:6379> INCR count (integer) 3
```

```
redis 127.0.0.1:6379> GET count “3”
```

Хоча GET повертає Count у вигляді рядка, INCR розуміє, що це ціле число і додає до нього одиницю. Спроба інкрементувати щось крім цілого закінчиться погано.

```
redis 127.0.0.1:6379> SET bad_count “a” OK
redis 127.0.0.1:6379> INCR bad_count
(error) ERR value is not integer or out of range
```

Якщо значення не можна інтерпретувати як ціле число,Redis справедливо лається. Також можна додавати (INCRBY) або віднімати (DECR, DECRBY) будь-яке ціле значення.

Транзакції

Ми вже зустрічалися з транзакціями в інших базах даних (Postgres та Neo4j), і команда Redis MULTI, що відкриває атомарний блок команд, призначена для тієї ж мети. Якщо укласти, скажімо, операції SET і INCR в один блок, то вони обидві виконаються успішно,або не виконається жодна. Часткового результату ми ніколи не побачимо.

Виконаємо перетворення на ключ ще одного URL та збільшення лічильника в рамках однієї транзакції. Транзакція розпочинається командою MULTI та завершується командою EXEC.

```
redis 127.0.0.1:6379 > MULTI OK
redis 127.0.0.1:6379> SET prag http://pragprog.comQUEUED
redis 127.0.0.1:6379> INCR count QUEUED
redis 127.0.0.1:6379> EXEC
1) OK
2) (integer) 2
```

При використанні MULTI команди не виконуються в момент введення (як у транзакціях Postgres), а встановлюються в чергу і потім виконуються послідовно.

Існує і аналог команди SQL ROLLBACK – перервати транзакцію дозволяє команда DISCARD, яка очищає чергу транзакції. Але на відміну від ROLLBACK, вона не відкочує базу даних до попереднього стану; транзакція взагалі починає виконуватися. Ефект той самий, хоча механізми зовсім різні (відкат транзакції та скасування операції).

Складові типи даних

Досі ми бачили досить лише збереження рядка і цілого числа як

значень ключів - навіть у рамках транзакції - це, звичайно, чудово, але в більшості завдань програмування та зберігання даних доводиться мати справу з більш різноманітними типами даних. Вбудовані механізми зберігання списків, хешей, множин і відсортованих множин пояснюють популярність Redis, і, познайомившись із операціями над цими типами, ви, напевно, погодитеся, що популярність заслужена.

У цих колекціях можна зберігати множину значень (до 2^32, або понад 4 мільярдів елементів). Більш ніж достатньо для того, щоб помістити всі облікові записи на Facebook до списку під одним ключем.

Хоча деякі команди Redis виглядають загадково, усі вони влаштовані за загальним зразком. Команди над множинами починаються з літери S, над хешами – з літери H, над відсортованими множинами – з літери Z. Команди над списками зазвичай починаються з літери L (від left – лівий) або R (від right – правий) – залежно від того , з якого боку списку застосовується операція (наприклад, LPUSH).

Хеш

Хеші грають у Redis роль об'єктів-контейнерів, здатних зберігати довільне число пар ключ-значення. Скористайтеся хешом для обліку користувачів, які зареєструвалися в нашій службі скорочення URL-адрес.

Хеші хороші тим, що дозволяють уникнути збереження даних із ключами, що мають штучні префікси. (Зверніть увагу, що всередині ключа використовується знак двокрапки. Це допустимий символ, який часто застосовується для логічного розбиття ключа на сегменти. Але це не більше, ніж угода, що не має в Redis глибокого сенсу.)

```
redis 127.0.0.1:6379> MSET user:eric:name “Eric Redmond” user:eric:password s3cretOK
redis 127.0.0.1:6379> MGET user:eric:name user:eric:password
1)"Eric Redmond"
2)"s3cret"
```

Замість окремих ключів ми можемо створити хеш, у якому зберігатимуться пари ключ-значення.

```
redis 127.0.0.1:6379> HMSET user:eric name “Eric Redmond” password s3cretOK
```

Щоб отримати всі значення з хешу, потрібно знати лише один ключ.

```
redis 127.0.0.1:6379> HVALS user:eric
1)"Eric Redmond"
2)"s3cret"
```

Також можна отримати всі ключі з хешу.

```
redis 127.0.0.1:6379> HKEYS user:eric
1)"name"
2)"password"
```

Або запитати лише одне значення, передавши спочатку ключ Redis, а потім ключ усередині хешу. У прикладі нижче ми отримуємо лише пароль.

```
redis 127.0.0.1:6379> HGET user:eric password "s3cret"
```

На відміну від документних сховищ типу Mongo або CouchDB, хеші в Redis не можуть бути вкладеними (як і всі інші складові типи даних, зокрема списки). Іншими словами, у хеші можна зберігати лише рядки.

Існують також команди для видалення з хешу (HDEL), збільшення цілого значення на задану величину (HINCRBY) та отримання загальної кількості елементів у хеші (HLEN).

Список

Список містить упорядкований набір значень і може виступати як у ролі черги (першим прийшов, першим обслужено), так і в ролі стека (останнім прийшов, першим обслужений). Підтримуються також більш складні операції, наприклад, вставка в середину списку, обмеження розміру списку та переміщення значень одного списку в інший.

Оскільки наша служба скорочення URL тепер вміє враховувати користувачів, чому б не надати їм можливість зберігати «списки побажань», тобто списки URL-адрес, які вони хотіли б відвідати. Призначимо списку коротких URL-сайтів, куди ми хотіли б заглянути, ключ USERNAME:wishlist і поміщатимемо значення в кінець списку (праворуч).

```
redis 127.0.0.1:6379> RPUSH eric:wishlist 7wks gog prag (integer) 3
```

Як і в більшості команд вставки в колекцію, Redis повертає кількість вставлених значень. Інакше кажучи, якщо ми вставили три значення, то отримуємо у відповідь число 3. У будь-який момент можна дізнатися про довжину списку за допомогою команди LLEN.

Команда LRANGE дозволяє отримати будь-яку частину списку,

задавши першу та останню позицію. У операціях зі списками вважається, перший елемент має індекс 0. Негативна позиція означає, що відлік потрібно вести з кінця списку.

```
redis 127.0.0.1:6379> LRANGE eric:wishlist 0 -1
1)"7wks"
2)"gog"
3)"prag"
```

Команда LREM видаляє зі списку із заданим ключем зазначені значення. Їй також необхідно передати число, що повідомляє, скільки елементів із зазначеним значенням видаляти. Якщо цей параметр дорівнює 0, то видаляються всі відповідні значення:

```
redis 127.0.0.1:6379> LREM eric:wishlist 0 gog
```

Якщо лічильник більше 0, то буде видалено не більше заданого числа збігів, а якщо менше 0, перегляд списку почнеться з кінця (з правого боку).

Щоб видалити та повернути значення у тому порядку, в якому вони додавалися (як у черзі), потрібно витягувати їх з лівого кінця списку (голови).

```
redis 127.0.0.1:6379> LPOP eric:wishlist "7wks"
```

Щоб використати список як стек, потрібно додавати значення командою RPUSH, а витягувати командою RPOP. Всі ці операції виконуються постійно.

Поводження черги можна досягти також комбінацією команд LPUSH і RPOP, а поведінки стека – комбінацією LPUSH і LPOP.

Нехай потрібно видалити значення зі списку побажань та перемістити їх до списку відвідуваних сайтів. Щоб виконати рух атомарно, ми можемо укласти команди pop і push в блок MULTI. На мові Ruby ці кроки можна було б записати, як показано нижче (командна оболонка тут не підійде, тому що нам необхідно зберегти витягнуте значення, тому ми скористалися gem-пакетом redis-rb):

```
redis.multi do
site = redis.rpop('eric:wishlist') redis.lpush('eric:visited', site) end
```

Але Redis надає також команду, яка дозволяє за одну операцію видалити (pop) значення кінця одного списку і додати (push) на початок іншого. Вона називається RPOPLPUSH (pop праворуч, push зліва).

```
redis 127.0.0.1:6379> RPOPLPUSH eric:wishlist eric:visited
```

Якщо тепер переглянути список побажань, виявиться, що елемента prag більше немає; він тепер знаходиться в списку visited. Це корисний механізм створення черг команд.

Якщо ви шукаєте в документації по Redis команди RPOPRPUSH, LPOPLPUSH та LPOPRPUSH, то з подивом виявите, що їх немає. RPOPLPUSH – єдиний варіант, тому будуйте списки відповідно.

Блокуючі списки

Отже, наша служба скорочення URL успішно стартувала і можна подумати про додавання до неї соціальних функцій – наприклад, систему коментування в реальному часі, де користувачі могли б залишати свої зауваження про відвідувані сайти.

Напишемо просту систему обміну повідомленнями, де кілька клієнтів можуть додавати коментарі, а один клієнт (аналітик) витягує повідомлення з черги. Ми хотіли б, щоб аналітик чекав на появу нових коментарів і витягував їх у міру надходження. Redis пропонує для вирішення подібних завдань кілька блокуючих команд.

Відкрийте ще один термінал та запустіть у ньому клієнт redis-cli. Цей екземпляр буде нашим аналітиком. Команда, яка чекає на надходження значення, називається BRPOP. Їй необхідно передати ключ списку, з якого витягувати значення, та таймувати в секундах; нижче ми задали п'ять хвилин.

```
redis 127.0.0.1:6379> BRPOP comments 300
```

Тепер перейдіть на першу консоль і помістіть повідомлення до списку comments.

```
redis 127.0.0.1:6379> LPUSH comments “Prag is great!I buy all my books there.”
```

Повернувшись на консоль аналітика, ви побачите, що команда повернула два рядки: ключ та витягнуте значення. Крім того, на консолі друкується час, проведений в очікуванні.

```
1)“comments”
2)“Prag is great! I buy all my books there.” (50.22s)
```

Існує також блокуюча версія операції pop зліва (BLPOP) та комбінованої операції "pop справа, push зліва" (BRPOPLPUSH).

Множина

Наш URL-адреса поступово оформляється, але добре було б додати ще засіб угруповання URL за якоюсь загальною ознакою.

Множина називається невпорядкована колекція без дублікатів. Вони добре підходять до виконання таких операцій над значеннями двох і більше ключів, як об'єднання і перетин.

Щоб розбити URL-адреси на групи із загальним ключем, ми можемо створити множину і додати до нього кілька значень командою SADD.

```
redis 127.0.0.1:6379> SADD news nytimes.com pragprog.com (integer) 2
```

Redis додав два значення. Команда SMEMBERS дозволяє витягти все множину, не гарантуючи певного порядку.

```
redis 127.0.0.1:6379> SMEMBERS news
1)"pragprog.com"
2)"nytimes.com"
```

Додамо ще одну категорію tech для сайтів технічного спрямування.

```
redis 127.0.0.1:6379> SADD tech pragprog.com apple.com (integer) 2
```

Щоб знайти, які сайти одночасно публікують новини та мають технічну тематику, виконаємо перетин обох множин командою SINTER.

```
redis 127.0.0.1:6379> SINTER news tech
1) "pragprog.com"
```

Анітрохи не складніше видалити з однієї множини значення, що зустрічаються в іншому. Щоб знайти всі новинні, але не технічні сайти, скористаємося командою SDIFF:

```
redis 127.0.0.1:6379> SDIFF news tech
1) "nytimes.com"
```

Можна також побудувати об'єднання сайтів - тобто множину як новинних, так і технічних сайтів. Оскільки це множина, дублікати автоматично видаляються.

```
redis 127.0.0.1:6379> SUNION news tech
1)"apple.com"
2)"pragprog.com"
3)"nytimes.com"
```

Цю множину значень можна відразу ж зберегти в новій множині (`SUNIONSTORE destination key [key …]`).

```
redis 127.0.0.1:6379> SUNIONSTORE websites news tech
```

За допомогою цієї команди можна реалізувати корисний прийом:

копіювання значень одного ключа до іншого ключа – SUNIONSTORE news_copy news. Аналогічні команди існує для збереження результатів перетину (SINTERSTORE) та різниці (SDIFFSTORE).

Якщо команда RPOPLPUSH переміщає значення зі списку до списку, то SMOVE робить те саме для множин, тільки її назва простіше запам'ятати.

І як LLEN повертає довжину списку, так SCARD (кардинальне число множини) підраховує кількість елементів у множині; щоправда, цю назву запам'ятати важче.

Оскільки множини не впорядковані, не існує команд для виконання операцій зліва, справа або взагалі з будь-яким зазначенням позиції. Для отримання випадкового значення з множини досить просто команди SPOP key, а видалення значень – команди SREM key value [value …].

На відміну від списків блокуючих команд для множин не існує.

Відсортовані множини

Розглянуті досі типи даних Redis легко відображаються стандартні конструкції мов програмування. Відсортовані ж множини запозичують лише потроху в інших типів. Вони впорядковані, як списки, і містять дублікатів, як множини. Вони зберігаються пари поле-значення, як у хешах, але полі – не рядок, а число, що означає порядок проходження значення у множині. Можна вважати, що відсортована множина – аналог черги з пріоритетами та довільним доступом. Але така гнучкість має ціну. Оскільки відсортовані множини підтримують упорядкованість значень, то вставка елемента проводиться за час log(N) (де N – розмір множини), а не за постійний час, як у випадку хешей та списків.

Далі ми хочемо відслідковувати популярність конкретних коротких адрес. Щоразу, як хтось відвідує URL-адресу, його оцінка збільшується. Як і у випадку хеша, для додавання елемента у відсортовану множину потрібно вказати ключ Redis і два значення: оцінку та сам елемент.

```
redis 127.0.0.1:6379> ZADD visits 500 7wks 9 gog 9999 prag
(integer) 3
```

Щоб збільшити оцінку, ми можемо або знову додати елемент, вказавши

нове значення, - при цьому оцінка оновиться, але нове значення не додасться, - або збільшити її на деяку величину, в результаті чого буде повернено нове значення.

```
redis 127.0.0.1:6379> ZINCRBY visits 1 prag
"10000"
```

Щоб зменшити значення, потрібно так само, за допомогою команди ZINCRBY додати негативну величину.

Діапазони

Для отримання значень з нашої множини відвідувань можнавиконати команду ZRANGE, яка повертає діапазон, заданий своїми межами – як і команда списку LRANGE. Але тільки у разі відсортованої множини діапазон ще й упорядковується за величиною оцінки в порядку зростання. Таким чином, щоб знайти два відвідуваних сайту (нумерація починається з 0), виконайте наступну команду:

```
redis 127.0.0.1:6379> ZRANGE visits 0 1
1)"gog"
2)"7wks"
```

Щоб отримати ще й оцінки кожного елемента, додайте уточнення WITHSCORES. Якщо потрібно відсортувати елементи в порядку зменшення, скористайтеся командою зі словом REV, наприклад ZREVRANGE.

```
redis 127.0.0.1:6379> ZREVRANGE visits 0 -1 WITHSCORES
1) "prag" 2) "10000"
3) "7wks" 4) "500"
5) "gog" 6) "9"
```

Але якщо ми вибрали відсортовану множину, то, напевно, хочемо запитувати діапазон значень оцінки, а не позицій. Команда ZRANGEBYSCORE має дещо інший синтаксис, ніж ZRANGE. За замовчуванням нижня та верхня межа включаються; щоб їх виключити, слід на початку діапазону поставити дужку, що відкриває: (. Таким чином, наступна команда поверне всі оцінки, для яких 9 <= score <= 10000:

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits 9 9999
1)"gog"
2)"7wks"
```

А така команда поверне оцінки, для яких 9 < score <= 10000:

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits (9 9999
1) "7wks"
```

Можна також встановити діапазони з нескінченними межами. Наступна

команда поверне всю множину.

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits -inf inf
```

Щоб перерахувати значення у зворотному порядку, скористайтесь командою ZREVRANGEBYSCORE.

Крім пошуку діапазону значень за рангом (індексом) або оцінкою, існують також відповідні команди видалення діапазону: ZREMRANGEBYRANK та ZREMRANGEBYSCORE.

Об'єднання

Як і для багатьох, можна створити новий ключ, який міститиме об'єднання або перетин одного або більше ключів. Це більш складна команда, тому що вона повинна не тільки об'єднати ключі – ця операція порівняно проста, – а й перерахувати оцінки, що можливо різняться. Операція об'єднання записується так:

```
ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight
...]] [AGGREGATE SUM|MIN|MAX]
```

Тут destination – ключ, у якому зберігається результат, а key – один або кілька ключів, що об'єднуються. Параметр numkeys задає кількість ключів, що об'єднуються, а параметр weight – необов'язкове число (ваговий коефіцієнт), на яке слід помножити оцінку у відповідному ключі (якщо є два ключі, то має бути і дві ваги). Нарешті, aggregate – необов'язкове правило агрегування виважених оцінок; за замовчуванням проводиться підсумовування, але можна також задати обчислення мінімуму або максимуму.

Скористайтеся цією командою для обчислення міри важливості відсортованої множини коротких адрес.

Спочатку створимо новий ключ, у якому зберігатимуться оцінки коротких адрес, обчислені за кількістю голосів. Кожен відвідувач сайту може проголосувати за або проти сайту, і його голос додається до підсумкової оцінки.

```
redis 127.0.0.1:6379> ZADD votes 2 7wks 0 gog 9001 prag
(integer) 3
```

Ми хочемо знайти найважливіші із зареєстрованих у системі сайтів,

застосовуючи для оцінки комбінацію кількості голосів та кількості відвідувань. Голоси важливіші, але й відвідування дають якийсь внесок, хоч і менший (часто людина буває настільки зачарована сайтом, що просто забуває проголосувати). Ми хочемо скомбінувати обидві оцінки та обчислити на їх основі оцінку важливості сайту. Для цього призначимо кількості голосів удвічі більшу вагу, аніж кількості відвідувань.

```
      ZUNIONSTORE importance 2 visits votes WEIGHTS 1 2 AGGREGATE SUM
(integer) 3
      redis 127.0.0.1:6379> ZRANGEBYSCORE importance -inf inf WITHSCORES
      1) "gog" 2) "9"
      3) "7wks" 4) "504"
      5) "prag" 6) "28002"
```

Ця команда має й інші застосування. Наприклад, щоб подвоїти всі оцінки у множині, ми можемо побудувати об'єднання з єдиним ключем, задавши вагу 2 і зберігши результат у вихідній множині.

```
      redis 127.0.0.1:6379> ZUNIONSTORE votes 1 votes WEIGHTS 2
      (integer) 2
      redis 127.0.0.1:6379> ZRANGE votes 0 -1 WITHSCORES
      1) "gog" 2) "0"
      3) "7wks" 4) "4"
      5)"prag" 6) "18002"
```

Для відсортованих множин є аналогічна команда (ZINTERSTORE), яка обчислює перетин.

Термін зберігання

Сховища ключів та значень типу Redis часто застосовуються як швидкий кеш для зберігання даних, які важко чи довго обчислювати щоразу заново. Завдання терміну зберігання дозволяє уникнути необмеженого зростання множини ключів, оскільки Redis автоматично видаляє пару ключ-значення після зазначеного часу.

Щоб задати термін зберігання ключа, нам знадобиться команда EXPIRE, якій передається існуючий ключ і час його життя в секундах. Нижче ми задаємо термін зберігання 10 секунд. Якщо протягом цього проміжку перевірити існування ключа за допомогою команди EXISTS, ми отримаємо відповідь 1 (true). Але якщо трохи почекати, то зрештою команда поверне 0 (false).

```
      redis 127.0.0.1:6379> SET ice “I'm melting…” OK
```

```
redis 127.0.0.1:6379> EXPIRE ice 10
(integer) 1
redis 127.0.0.1:6379> EXISTS ice
(integer) 1
redis 127.0.0.1:6379> EXISTS ice
(integer) 0
```

Установлення та завдання терміну зберігання ключів – настільки часта операція, що у Redis є для неї спеціальна команда SETEX.

```
redis 127.0.0.1:6379> SETEX ice 10 "I'm melting…"
```

Команда TTL дає змогу запросити час життя ключа. Якщо встановити термін зберігання ключа ice, як показано вище, а потім перевірити його TTL, то ми дізнаємося, скільки секунд йому залишилося жити.

```
redis 127.0.0.1:6379 > TTL ice (integer) 4
```

Будь-коли до закінчення терміну зберігання тайм-аут можна скасувати, виконавши команду PERSIST ключ.

```
redis 127.0.0.1:6379> PERSIST ice
```

Можна також вказати не відносне (у вигляді секунд, починаючи з поточного моменту), а абсолютний час видалення ключа. Для цього призначена команда EXPIREAT, яка приймає часову мітку Unix (кількість секунд з півночі 1 січня 1970 року).

Типовий прийом, що дозволяє зберігати в кеші тільки ключі, що часто використовуються, полягає в тому, щоб оновлювати термін зберігання при кожному вилученні значення. Цей алгоритм, званий MRU (most recently used – останні використані) гарантує, що затребувані ключі Redis залишить, а ті, до яких звернення виробляються рідко, після таймууту видалить звичайним порядком.

Простір імен

Досі ми мали справу лише з одним простором імен. Алеіноді потрібно розміщувати ключі в різних просторах імен – за аналогією з сегментами Riak. Наприклад, якщо ви пишете інтернаціоналізоване сховище ключів та значень, то можете зберігати переклади відповідей на різні мови у різних просторах імен. У німецькому просторі імен ключу greeting (привітання) буде зіставлено значення «guten tag», а французькому – «bonjour». Після того, як користувач вибере потрібну мову, програма буде витягувати значення з

відповідного простору імен.

У термінології Redis простір імен називається базою даних, і кожному такому простору зіставляється числовий ключ.

Поки що ми працювали з простором імен 0, що мається на увазі за умовчанням (воно також називається базою даних 0). Наступні команди порівнюють ключ greeting англійське слово hello.

```
redis 127.0.0.1:6379> SET greeting hello OK
redis 127.0.0.1:6379> GET greeting "hello"
```

Але якщо перейти на іншу базу даних командою SELECT, то цей ключ стане недоступним.

```
redis 127.0.0.1:6379> SELECT 1 OK
redis 127.0.0.1:6379[1]> GET greeting (nil)
```

А надання йому значення в новому просторі імен не змінить колишнього значення.

```
redis 127.0.0.1:6379[1]> SET greeting “guten tag” OK
redis 127.0.0.1:6379[1]> SELECT 0 OK
redis 127.0.0.1:6379> GET greeting "hello"
```

Оскільки всі бази даних працюють в тому самому екземплярі сервера, то Redis дозволяє перекидати ключі за допомогою команди MOVE. Наступні команди переміщують ключ greeting до бази даних 2.

```
redis 127.0.0.1:6379> MOVE greeting 2
(integer) 2
redis 127.0.0.1:6379> SELECT 2 OK
redis 127.0.0.1:6379[2]> GET greeting “hello”
```

Це буває корисно, коли потрібно виконувати на одному сервері Redis різні програми, дозволяючи їм обмінюватися даними.

Крім всього цього у Redis є багато інших команд, наприклад: перейменування ключів (RENAME), визначення типу значення ключа (TYPE) та видалення пари ключ-значення (DEL). Є також небезпечна команда FLUSHDB, яка видаляє всі ключі з однієї бази даних Redis, і її небезпечна родичка - команда FLUSHALL, що видаляє всі ключі зі всіх баз даних. Повний перелік команд Redis можна знайти в онлайновій документації.

Висновок

Різноманітність типів даних у Redis і підтримка складних запитів

перетворюють цю систему на щось набагато більше, ніж стандартне сховище ключів та значень. Вона може виступати у ролі стека, черги чи черги з пріоритетами, служити сховищем об'єктів (завдяки хешам) і навіть вміє виконувати складні операції над множинами (об'єднання, перетин та різниця). У системі є багато атомарних команд та механізм транзакцій для виконання багатокрокових команд. Вбудована також можливість завдання терміну зберігання ключів, що дозволяє використовувати сховище як кеш.

Redis не відчуває нестачі в командах - всього їх більше 120. Назви більшості команд пояснюють їхнє призначення, потрібно лише звикнути до думки, що деякі надлишкові літери опущені (як, наприклад, у команді INCRBY) і що математична точність іноді не так допомагає, скільки заплутує (порівняйте, наприклад, ZCOUNT – число елементів у відсортованій множині та SCARD – кардинальне число множини).

Redis вже став невід'ємною частиною багатьох систем. На його основі створено декілька проєктів з відкритим вихідним кодом – від Resque (написана на Ruby служба асинхронних черг завдань) до SocketStream (система керування сеансами Node.js). Незалежно від того, яка база даних обрана як канонічна система, подумати про використання спільно з нею Redis безсумнівно має сенс.

Завдання для індивідуальної роботи

1. Знайдіть повну документацію щодо команд Redis, де наведено зокрема оцінку складності в нотації «О-велике» (O(x)).

2. Установіть драйвер своєї улюбленої мови програмування та підключіться до сервера Redis. Вставте та інкрементуйте числове значення в рамках однієї транзакції.

3. Користуючись драйвером за своїм вибором, напишіть програму, яка читає дані з блокуючого списку і кудись виводить їх (на консоль, файл, канал Socket.io і т. д.), а також іншу програму, яка поміщає дані в цей перелік.

Лекція 16. Більш складні застосування, розподілені обчислення у Redis

Спочатку ми розглядали Redis як сервер структур даних. Тепер ми продовжимо зводити будівлю на цьому фундаменті і познайомимося з деякими більш сучасними функціями Redis, зокрема, з конвеєрами, моделлю публікації-підписки, налаштуванням системи та реплікацією. Крім того, ми навчимося будувати кластер Redis, швидко зберігати великі масиви даних та використовувати фільтри Блума.

Інтерфейс Redis

Redis, що налічує всього 20 000 рядків вихідного коду, є досить простим проєктом. І його інтерфейс також простий - у тому сенсі, що приймає буквально ті рядки, що вводяться на консолі.

Telnet

Ми можемо взаємодіяти з Redis без будь-якого командного інтерфейсу, просто надсилаючи команди через TCP-з'єднання або через telnet. Кожна команда повинна завершуватися знаками повернення каретки та переходу на новий рядок (CRLF, або \r\n).

redis/telnet.sh

```
$ telnet localhost 6379 Trying 127.0.0.1...
Connected to localhost. Escape character is '^]'. SET test hello
    +OK
GET test
    $5 hello
SADD stest 1 99
    :2
SMEMBERS stest
    *2
$1 1
$2 99
CTRL-]
```

Як бачите, ми вводимо ті самі команди, що при роботі з консоллю, тільки консоль проводила незначне форматування відповідей.

Redis повертає стан OK, додаючи на початок префікс +. Перед тим як повернути рядок *hello*, Redis послав $5, що означає «довжина наступного рядка складає 5 символів».

Числу 2, повернутому після додавання двох значень у ключ test, передує знак :, що показує, що це ціле число (успішно було додано два значення).

Запитавши два значення, ми отримали у першому рядку число 2, якому передує зірочка, – це означає, що далі йдуть два складові значення. Наступні два рядки такі ж, як при поверненні *hello*, тільки в першій ми бачимо число *1*, а в другому – рядок *99*.

Конвеєри

Можна також передавати рядки по одному, скориставшись програмою netcat (nc), яка спочатку була написана для операційної системи BSD, але тепер включається за замовчуванням багато дистрибутивів Unix. У цьому випадку ми повинні явно завершувати кожен рядок знаками CRLF (telnet робить це автоматично). Ми також очікуємо протягом однієї секунди після появи введеної команди на консолі, щоб дати серверу Redis час відповіді. У деяких, але не у всіх реалізаціях nc є прапор -q, що дозволяє обійтися без очікування – перевірте.

```
$ (echo-en “ECHO hello\r”; sleep 1) | nc localhost 6379
$5 hello
```

Цим можна скористатися, щоб організувати конвеєр команд, тобто надсилати кілька команд в одному запиті.

```
$ (echo -en "PING\PING\PING\n"; sleep 1) | nc localhost 6379
+PONG
+PONG
+PONG
```

Це набагато ефективніше, ніж відправляти по одній команді, тому такий варіант завжди слід мати на увазі – особливо під час роботи з транзакціями. Не забувайте лише завершувати кожну команду символами \r\n, які інтерпретує сервер як розмежувач.

Публікація-підписка

На попередній лекції нам удалося реалізувати примітивну блокуючу чергу за допомогою списку. Ми поміщали у чергу дані, які зчитувалися блокуючою командою pop. Це дозволило нам збудувати дуже просту модель публікації-підписки. Повідомлення в чергу можуть розміщувати різні

клієнти, а витягує їх єдиний читач у міру надходження. Але, у багатьох випадках нам необхідна зворотна поведінка, коли кілька передплатників бажають читати повідомлення, опубліковані одним видавцем, як показано на рис. 37. Redis пропонує кілька спеціалізованих команд для організації публікації-підписки.

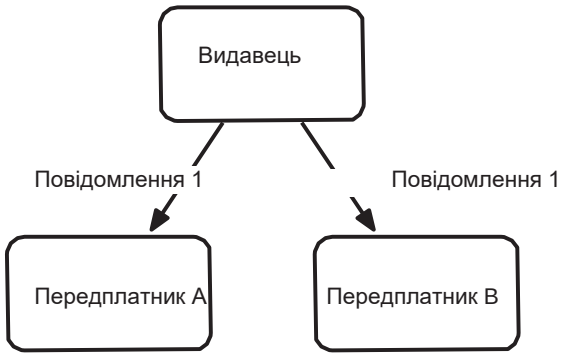


Рис. 37. Видавець надсилає повідомлення всім передплатникам

Удосконалюємо механізм коментування, розроблений вчора за допомогою блокуючих списків, дозволивши користувачеві надсилати коментар кільком передплатникам (а не тільки одному). Почнемо з передплатників, які встановлюють з'єднання із ключем, який у термінології публікації-підписки називається каналом. Запустіть ще двох клієнтів і підпишіться на канал коментарів. Передплата блокує командний інтерфейс.

```
redis 127.0.0.1:6379> SUBSCRIBE comments Reading messages... (Ctrl+C до quit)
1)"subscribe"
2)“comments”
3)(integer) 1
```

Маючи двох передплатників, ми можемо опублікувати через канал comments довільне рядкове повідомлення. PUBLISH поверне число 2, що означає, що повідомлення було отримано двома передплатниками.

```
redis 127.0.0.1:6379> Publish comments “Check out this shortcoded site! 7wks”(integer) 2
```

Обидва передплатники одержують багаточасткову відповідь (multibulk reply) (список), що складається з трьох елементів: рядок «message», ім'я каналу та саме опубліковане повідомлення.

```
1)"message"
2)“comments”
3)“Check out this shortcoded site! 7wks”
```

Коли клієнт втратить інтерес до отримання кореспонденції, він може

виконати команду UNSUBSCRIBE comments, щоб вимкнутись від каналу comments, або команду UNSUBSCRIBE без параметрів – щоб вимкнутись від усіх каналів. Відзначимо, що в оболонці redis-cli потрібно натиснути CTRL+C для розриву з'єднання.

Інформація про сервер

Перед тим як змінювати системні налаштування Redis, корисно переглянути, що видає команда INFO, оскільки в результаті зміни налаштувань змінюються і деякі значення, що їй повертаються. INFO виводить відомості про сервер, у тому числі номер версії, ідентифікатор процесу, обсяг зайнятої пам'яті та час з моменту останнього перезапуску.

```
redis 127.0.0.1:6379 > INFO
redis_version:2.4.5 redis_git_sha1:00000000 redis_git_dirty:0
arch_bits:64 multiplexing_api:kqueue process_id:54046 uptime_in_seconds:4
uptime_in_days:0 lru_clock:1807
…
```

Рекомендуємо повернутися до цієї команди пізніше, оскільки вона дає корисний миттєвий знімок стану сервера та його налаштувань. Виводиться навіть інформація про довговічність, фрагментацію пам'яті та стан реплікації.

Налаштування Redis

Досі ми запускали Redis із налаштуваннями за замовчуванням. Але взначною мірою Redis зобов'язаний своєю силою гнучкості конфігурування, що дозволяє адаптувати його до конкретного застосування. Файл redis.conf, що входить до складу дистрибутива – в системах *nix він встановлюється в каталог /etc/redis – практично не потребує пояснень, тому ми розглянемо лише його частину. Розкажемо про деякі найбільш уживані параметри по порядку.

```
daemonize no port 6379 loglevel verbose logfile stdout database 16
```

За замовчуванням параметр daemonize дорівнює no, тому сервер і запускається в пріоритетному режимі. Для тестування це зручно, але у виробничій системі не дуже. Якщо встановити значення yes, то сервер буде працювати у фоновому режимі і запише ідентифікатор свого процесу в pid-файл.

У наступному рядку задається номер порту сервера – 6379. Це

налаштування особливо корисне при запуску кількох екземплярів Redis на одній машині. Параметр loglevel за умовчанням дорівнює verbose (докладно), але у виробничій системі краще задати режим notice або warning. Параметр logfile налаштований так, що діагностична інформація друкується на stdout (стандартний висновок, тобто консоль), але в режимі демона краще вказати ім'я журналу.

Параметр database визначає кількість доступних баз даних Redis. Про те, як перемикати базу даних, ми говорили сьогодні. Якщо ви плануєте використовувати лише один простір імен, то краще встановити цей параметр один.

Довговічність

Redis підтримує кілька режимів збереження. Насамперед, це відсутність збереження взагалі, коли всі значення зберігаються в оперативній пам'яті. Якщо сервер використовується для кешування, це розумний вибір, тому що за довговічність доводиться розплачуватися затримками. Одна з особливостей Redis, що відрізняють її від інших швидких кешів, наприклад, memcached, - вбудована підтримка зберігання значень на диску. За замовчуванням пари ключ-значення зберігаються лише періодично. Команда LASTSAVE повертає тимчасову мітку Unix, що відповідає останньому успішному запису на диск. Те саме значення можна прочитати з поля last_save_time, що повертається командою INFO.

Примусово скинути дані на диск дозволяє команда SAVE (або BGSAVE, яка зберігає асинхронно у фоновому режимі).

```
redis 127.0.0.1:6379 > SAVE
```

Після цього в журналі сервера з'являться рядки:

```
[46421] 10 Oct 19:11:50 * Background saving started by pid 52123
[52123] 10 Oct 19:11:50 * DB saved on disk
[46421] 10 Oct 19:11:50 * Background saving terminated with success
```

Ще один спосіб забезпечення довговічності – зміна параметрів миттєвих знімків у конфігураційному файлі.

Миттєві знімки

Ми можемо змінити частоту збереження на диск, додавши, вилучивши

або змінивши одне з полів збереження. За замовчуванням таких полів 3, і всі вони починаються ключовим словом save, за яким слідує час у секундах і мінімальна кількість ключів, які повинні бути змінені, щоб розпочався запис на диск.

Наприклад, щоб зберігати кожні 5 хвилин (300 секунд) за умови, що змінився хоча б один ключ, потрібно написати:

```
save 300 1
```

У конфігураційному файлі налаштовані розумні умовчання, а саме: якщо змінилося 10 000 ключів, зберігати раз на 60 секунд; якщо змінилося 10 ключів, зберігати раз на 300 секунд; якщо змінився хоча б один ключ, зберігати не рідше ніж раз на 900 секунд (15 хвилин).

```
save 900 1
save 300 10
save 60 10000
```

Можна додати додаткові поля збереження, які визначають точніші пороги.

Файл із дозаписом

За умовчанням Redis забезпечує довговічність в кінцевому рахунку, тобто асинхронно скидає значення на диск з інтервалами, які визначаються налаштуваннями збереження, або за явною командою від клієнта. Це прийнятно для кешу другого рівня або сервера сеансів, але недостатньо, коли довговічність даних - неодмінна умова, наприклад, при роботі з фінансовими даними. Користувачі не зрозуміють вас, втративши гроші через аварійне завершення сервера Redis.

Redis підтримує файл з дозаписом (appendonly.aof), де зберігаються всі команди записи. Це подібність протоколювання з випереджаючим записом. Якщо сервер "впаде", не записавши значення, то при перезапуску він виконає збережені команди, відновивши останній стан. Щоб активувати цей режим, слід присвоїти значення yes параметру appendonly у файлі redis.conf.

```
appendonly yes
```

Далі ми маємо вирішити, як часто скидати команди у файл. Режим always забезпечує максимальну надійність, оскільки кожна команда

зберігається відразу. Але він і найповільніший, що знецінює ті переваги, через які люди обирають Redis. За замовчуванням встановлюється режим everysec, у якому команди записуються разів у секунду. Це прийнятний компроміс, оскільки, з одного боку, сервер працює досить швидко, а, з іншого, ви в найгіршому випадку втратите лише зміни, що відбулися протягом однієї секунди. Зрештою, в режимі no скиданням на диск керує операційна система. Але це може відбуватися не надто часто, так що файл дозапису втрачає сенс і, можливо, його тоді краще взагалі не використовувати.

```
# appendfsync always appendfsync everysec
# appendfsync no
```

У режимі дозапису є інші параметри, про які можна прочитати в самому конфігураційному файлі. За певних способів експлуатації вони можуть виявитися корисними.

Безпека

Взагалі Redis не проєктувався як абсолютно безпечний сервер, але в документації ви можете натрапити на опис параметра requirepass і команди AUTH. Їх можна благополучно ігнорувати, тому що підтримується лише завдання пароля у відкритому вигляді. Оскільки клієнт може перевірити майже 100 000 паролів на секунду, то гра не коштує свічок, і це навіть не кажучи про те, що відкриті паролі в будь-якому разі небезпечні. Якщо потрібно забезпечити безпеку Redis, краще встановити гідний брандмауер і налаштувати SSH.

Цікаво, що Redis підтримує безпеку на рівні команд шляхом заплутування, дозволяючи приховувати чи придушувати команди. Наступна команда перетворює команду FLUSHALL (видалити всі ключі з системи) у рядок, що важко вгадується c283d93ac9528f986

```
rename-command FLUSHALL c283d93ac9528f986023793b411e4ba2
```

Якщо відправити серверу команду FLUSHALL, він поверне помилку. Навпаки, секретна команда працює.

```
redis 127.0.0.1:6379> FLUSHALL
(error) ERR unknown command 'FLUSHALL'
```

```
redis 127.0.0.1:6379 > c283d93ac9528f986023793b411e4ba2 OK
```

Більше того, ми можемо взагалі заборонити деякі команди, перейменувавши їх у порожній рядок:

```
rename-command FLUSHALL “”
```

Таким чином, можна заборонити скільки завгодно команд, що дозволяє організувати певний аналог спеціалізованого оточення.

Додаткові параметри

Існує ряд спеціальних параметрів для ведення журналу повільних запитів, завдання кодування, налаштування затримки та імпорту зовнішніх конфігураційних файлів. Але попереджаємо – зустрівши в документації згадки про віртуальну пам'ять Redis, не звертайте на них уваги, оскільки у версії 2.4 вони оголошені нерекомендованими і згодом можуть бути видалені.

Для вивчення конфігурації сервера Redis пропонує чудовий інструмент еталонного тестування. За замовчуванням він підключається до порту 6379 на локальній машині і надсилає 10 000 запитів від імені 50 клієнтів, що паралельно працюють. Збільшити кількість запитів до 100000 можна за допомогою аргументу -n.

```
$ redis-benchmark -n 100000
====== PING (inline) ======
100000 повідомлень було завершено в 3.05 seconds
50 parallel clients
3 bytes payload keep alive: 1
5.03% <= 1 milliseconds
98.44% <= 2 milliseconds
99.92% <= 3 milliseconds
100.00% <= 3 milliseconds
32808.40 requests per second
…
```

Можна протестувати інші команди, наприклад SADD і LRANGE; але що команда складніше, то більше часу вона виконується.

Реплікація головний-підлеглий

Як і інші розглянуті вище бази даних NoSQL (наприклад, MongoDB і Neo4j), Redis підтримує реплікацію як головний-підлеглий. За замовчуванням сервер є головним, якщо не підпорядкувати його якомусь іншому. Дані реплікуються довільне число підлеглих серверів.

Налаштувати підлеглі сервери нескладно. Для початку потрібно скопіювати файл redis.conf.

```
$ cp redis.conf redis-s1.conf
```

У файл потрібно внести лише дві зміни:

```
port 6380
slaveof 127.0.0.1 6379
```

Якщо все піде за планом, то під час запуску підлеглого сервера у його журналі з'являться рядки виду:

```
$ redis-server redis-s1.conf
[9003] 16 Oct 23:51:52* Connecting to MASTER... [9003] 16 Oct 23:51:52 *
MASTER <-> SLAVE sync started
[9003] 16 Oct 23:51:52* Non blocking connect for SYNC оголошено.
[9003] 16 Oct 23:51:52* MASTER <-> SLAVE sync: отримувати 28 bytes from
master [9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: Loading DB in memory
[9003] 16 Oct 23:51:52* MASTER <-> SLAVE sync: Finished with success
```

У журналі головного сервера – рядок 1 slaves. Якщо виконати на головному сервері команду

```
redis 127.0.0.1:6379> SADD meetings "StarTrek Pastry Chefs" "LARPers
Intl."
```

а потім підключитися до підлеглого, то можна буде отримати список зустрічей:

```
redis 127.0.0.1:6380> SMEMBERS meetings
1)"StarTrek Pastry Chefs"
2)"LARPers Intl."
```

У виробничій системі реплікація зазвичай включається підвищення доступності чи резервного копіювання, тому підлеглі сервери запускаються різних машинах.

Завантаження даних

Ми багато говорили про те, яка Redis швидка система, але відчути це, не маючи солідного масиву даних, складно. Давайте завантажимо на сервер Redis великий набір даних. Ми збираємося завантажити з сайту Freebase.com список (http://download.freebase.com/datadumps/latest/browse/book/isbn.tsv), що містить понад 2,5 мільйона назв виданих книг, що ідентифікуються міжнародним стандартним книжковим номером (ISBN).

Попередньо встановіть gem-пакет redis для Ruby.

```
$ gem install redis
```

Існує кілька способів вставити великий набір даних. Вони варіюються за швидкістю і відповідно за складністю.

Найпростіший метод – просто обійти в циклі весь список даних та виконати для кожного значення команду SET за допомогою стандартної бібліотеки клієнтів redis-rb.

redis/isbn.rb

```
LIMIT = 1.0 / 0 # 1.0/0 в Ruby позначає нескінченність
# %w{rubygems hiredis redis/connection/hiredis}.each{|r| require r}
%w{rubygems time redis}.each{|r| require r}

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall
count, start = 0, Time.now File.open(ARGV[0]).each do |line|
count += 1
next if count == 1
isbn, _, _, title = line.split("\t") next if isbn.empty? || title ==
"\n"

$redis.set(isbn, title.strip)

# встановіть значення LIMIT, якщо не хочете завантажувати весь набір
даних break if count >= LIMIT
end

puts "#{count} елементів за #{Time.now start} секунд"

$ ruby isbn.rb isbn.tsv
2456384 елементів за 266.690189 секунд
```

Щоб прискорити вставку та не використовувати JRuby, можна встановити додатковий gem-пакет hiredis. Це написаний на драйвер C, який працює значно швидше драйвера на чистому Ruby. Потім розкоментуйте рядок hiredis require, щоб завантажити драйвер. Можливо, ви не помітите серйозного покращення на такого роду завданні, що завантажує передусім процесор, але при використанні Ruby у виробничій системі ми рекомендуємо встановити його.

Помітного прискорення можна досягти за рахунок конвеєризації. Наступний скрипт створює пакет із 1000 рядків і вставляє його повністю, передаючи конвеєру. В результаті час вставки на нашій машині зменшився більш ніж утричі.

redis/isbn_pipelined.rb

```
BATCH_SIZE = 1000
LIMIT = 1.0 / 0 # 1.0/0 в Ruby позначає нескінченність
# %w{rubygems hiredis redis/connection/hiredis}.each{|r| require r}
%w{rubygems time redis}.each{|r| require r}

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
```

```
$redis.flushall
# надіслати дані у вигляді одного пакета оновлень
def flush(batch)
$redis.pipelined do batch.each do |saved_line|
isbn, _, _, title = line.split("\t")
next if isbn.empty? || title == "\n"
$redis.set(isbn, title.strip)
end
end
batch.clear
end

batch = []
count, start = 0, Time.now File.open(ARGV[0]).each do |line|
count += 1
next if count == 1

# помістити рядки в масив
batch << line
# коли масиві виявиться BATCH_SIZE елементів, скинути його
if batch.size == BATCH_SIZE
flush(batch)
puts "#{count-1} елементів" end
# встановіть значення LIMIT, якщо не хочете завантажувати весь набір
даних
break if count >= LIMIT
end

# скинути значення, що залишилися
flush(batch)

puts "#{count} елементів за #{Time.now start} секунд"

$ ruby isbn_pipelined.rb isbn.tsv
2666642 елементів за 79.312975 секунд
```

Таким чином ми зменшили кількість з'єднань з Redis, але побудова набору даних для передачі по конвеєру має власні накладні витрати. Організуючи конвеєр у виробничій системі, поекспериментуйте, задаючи різну кількість операцій в одному пакеті.

Для користувачів Ruby принагідно зазначимо, що якщо програма працює в неблокуючому режимі, застосовуючи Event Machine, то драйвер Ruby може задіяти em-synchrony за рахунок виклику EM:: Protocols::Redis.connect.

Кластер Redis

Крім простої реплікації, багато клієнтів Redis надають інтерфейс для побудови простого "рукоробного" розподіленого кластера. Написаний на Ruby клієнт redis-rb підтримує керований кластер із узгодженим хешуванням.

Можливо, ви ще пам'ятаєте, як ми обговорювали узгоджене хешування у розділі, присвяченому Riak, де можна додавати та видаляти вузли без старіння більшості ключів. Тут ідея така сама, тільки кластер управляється клієнтом, а чи не самими серверами.

Для початку нам знадобиться ще один сервер. На відміну від реплікації головний-підлеглий, тут обидва сервери грають роль головного (як у конфігурації за замовчуванням). Ми просто скопіювали файл redis.conf і замінили номер порту на 6380. Більше із серверами нічого робити не треба.

redis/isbn_cluster.rb

```
LIMIT = 10000
%w{rubygems time redis}.each{|r| require r} require 'redis/distributed'

$redis = Redis::Distributed.new([ "redis://localhost:6379/",
"redis://localhost:6380/"
])
$redis.flushall

count, start = 0, Time.now File.open(ARGV[0]).each do |line|
count += 1

next if count == 1
isbn, _, _, title = line.split("\t")
next if isbn.empty? || title == "\n"

$redis.set(isbn, title.strip)
# встановіть значення LIMIT, якщо не хочете завантажувати весь набір
даних
break if count >= LIMIT
end
puts "#{count} items in #{Time.now start} seconds"
```

Для створення мосту між двома та більше серверами довелося внести лише незначні зміни до існуючого клієнта для завантаження набору даних про книги. По-перше, потрібно зажадати файл redis/distributed із gem-пакета redis.

```
require 'redis/distributed'
```

Потім ми замінюємо клієнт Redis на Redis::Distributed і передаємо йому масив URI-адрес серверів. У всіх URI вказана схема redis і далі ім'я сервера (localhost) та порт.

```
$redis = Redis::Distributed.new([ "redis://localhost:6379/",
"redis://localhost:6380/"
])
```

Запуск клієнта проводиться так само, як і раніше.

```
$ ruby isbn_cluster.rb isbn.tsv
```

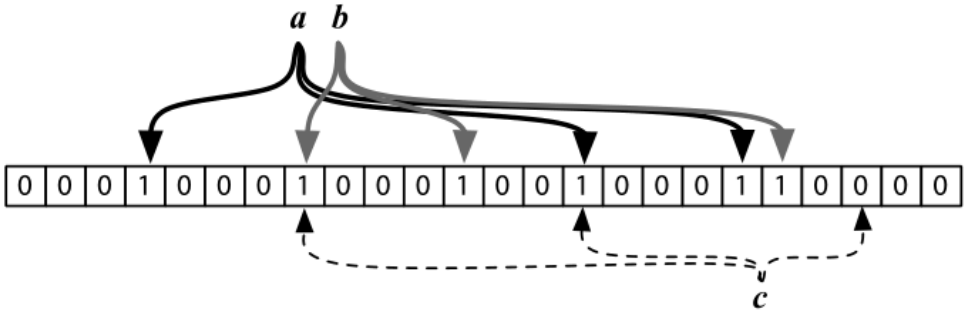
Але тепер на клієнта покладається набагато більше роботи, тому що він обчислює які ключі на яких серверах зберігаються. Переконатися в тому, що ключі дійсно зберігаються на різних серверах, можна спробувати запит ISBN з кожного сервера в командній оболонці. У відповідь на команду GET значення поверне лише один сервер. Але якщо запитувати ключі через об'єкт Redis::Distributed, то клієнт отримуватиме значення від правильного сервера.

Фільтри Блума

Стати власником унікального терміну – чудовий спосіб гарантувати успішність пошуку мережі. Давайте напишемо скрипт, який перевірятиме, чи є слово унікальним серед усіх слів, що зустрічаються в назвах книг з каталогу ISBN. Для перевірки того, чи траплялося слово раніше, можна скористатися фільтром Блума.

Фільтр Блума – це імовірнісна структура даних, яка перевіряє існування елемента у множині. Вперше ми зустрілися з нею у лекції присвяченій HBase. Фільтр Блума може повертати хибнопозитивну відповідь, але ніколи – хибнонегативну. Це корисно, якщо потрібно швидко встановити, що значення відсутнє.

Завдання доказу неіснування фільтр Блума вирішує за рахунок перетворення значення сильно розріджену послідовність бітів, яка порівнюється з об'єднанням бітів, відповідних всім існуючим значенням. Іншими словами, при додаванні нового значення проводиться операція АБО з поточною послідовністю бітів, що зберігається у фільтрі Блума. Якщо потрібно перевірити, чи існує вже дане значення, то ми виконуємо операцію І з послідовністю бітів, що зберігається. Якщо в даному значенні піднято хоча б один біт, який не піднятий у відповідному сегменті фільтра Блума, значить, це значення раніше не додавалася, тобто його явно немає у множині. Нижче ця ідея представлена графічно.



Напишемо програму, яка перебирає всі книги з набору ISBN, витягує та нормалізує назву книги, а потім розбиває її на окремі слова. Кожне слово перевіряється за фільтром Блума. Якщо фільтр повертає false, то слово ще не існує, і ми додаємо його у фільтр. Принагідно можна виводити все, що зустрілися нові слова.

```
$ gem install bloomfilter-rb

redis/isbn_bf.rb
# LIMIT = 1.0 / 0 # 1.0/0 в Ruby позначає нескінченність LIMIT = 10000
%w{rubygems time bloomfilter-rb}.each{|r| require r} bloomfilter =
BloomFilter::Redis.new(:size => 1000000)

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall

count, start = 0, Time.now File.open(ARGV[0]).each do |line|
count += 1
next if count == 1
_, _, _, title = line.split("\t") next if title == "\n"

words = title.gsub(/[^\w\s]+/, '').downcase
# виділяємо слова
words = words.split('') words.each do |word|
# пропустити слово, якщо воно вже є у фільтрі Блума
next if bloomfilter.include? (word)
# вивести слово, що не зустрічалося раніше слово
puts word
# додати нове слово у фільтр Блума
bloomfilter.insert(word)
end
# встановіть значення LIMIT, якщо не хочете завантажувати весь набір
даних
break if count >= LIMIT
end
puts "Чи містить Jabbyredis? #{bloomfilter.include?('jabbyredis')}"
puts "#{count} рядків за #{Time.now start} секунд"
```

Цей фільтр Блума на основі Redis написав на Ruby Ілля Григорик (Ilya Grigorik), але сама ідея переноситься на будь-якою іншу мову. При запуску клієнта використовується той самий файл ISBN, але ми аналізуємо тільки назви книг.

```
$ ruby isbn_bf.rb isbn.tsv
```

Спочатку буде зустрічатися багато загальновживаних слів типу and і the. Але ближче до кінця набору з'являються дедалі дивніші слова, наприклад unindustria.

Плюс цього підходу – можливість виявляти слова-дублікати. А мінус у тому, що час від часу просочуються хибно-позитивні відповіді - фільтр Блума може пропустити слово, яке ніколи не зустрічалося. Тому в реальних додатках виконується додаткова перевірка, наприклад, більш повільної бази даних у системі, куди записуються всі дані. Але це відбуватиметься досить рідко у припущенні, що розмір фільтра досить великий, а цей розмір можна оцінити заздалегідь.

SETBIT та GETBIT

Як ми вже говорили, принцип роботи фільтра Блума заснований на маніпуляціях з бітами в розрідженому векторі біта. У реалізації фільтра Блума в Redis, якою ми щойно скористалися, застосовуються дві команди, які недавно з'явилися, які і виконують ці маніпуляції: SETBIT і GETBIT.

Як і у всіх командах Redis, назва SETBIT говорить сама за себе. Команда встановлює один біт деякої позиції бітового вектора, причому позиції нумеруються з нуля. Цей спосіб часто застосовується для швидкої позначки в багатопараметричних системах - простіше завести кілька бітів, ніж описувати рядками.

Якби нам знадобилося створити систему обліку приправ до гамбургерів, то ми могли б зіставити кожній приправі позицію біта, наприклад: кетчуп = 0, гірчиця = 1, цибуля = 2, салат-латук = 3. Тоді гамбургер із гірчицею та цибулею можна було описати бітовим вектором 0110, який у командному рядку формується так:

```
redis 127.0.0.1:6379> SETBIT my_burger 1 1
(integer) 0
redis 127.0.0.1:6379> SETBIT my_burger 2 1
(integer) 0
```

Потім ми могли б перевірити, чи маємо додаватися до мого гамбургера латук або гірчиця. Якщо повернутий 0, то відповідь – ні, якщо 1 – так.

```
redis 127.0.0.1:6379> GETBIT my_burger 3
(integer) 0
redis 127.0.0.1:6379> GETBIT my_burger 1
(integer) 1
```

У реалізації фільтра Блума ця ідея використовується шляхом представлення значення у вигляді бітового вектора. При зверненні до методу insert() викликається команда SETBIT X 1 кожної позиції X, у якій біт піднятий, а за зверненні до include?() існування перевіряється шляхом виклику GETBIT X – і повертається false, якщо хоча б однієї позиції GETBIT повертає 0 .

Фільтр Блума чудово справляється із завданням зменшення надлишкового трафіку з більш повільною системою, що стоїть за ним, чи це повільна база даних, обмежений ресурс або мережний запит. Якщо є повільна база IP-адрес і потрібно відстежувати нових користувачів сайту, то можна попередньо перевірити за допомогою фільтра Блума, чи існує вже IP-адреса в системі. Якщо фільтр повернув false, можна з упевненістю сказати, що адреси ще немає, і відреагувати відповідним чином. Якщо ж фільтр повернув true, то адреса, можливо, існує, а, можливо, і немає, тому необхідне додаткове звернення до основного сховища. Саме тому так важливо правильно обчислити розмір фільтра – при відповідному розмірі фільтр Блума може зменшити частоту помилок, тобто ймовірність хибно-позитивної відповіді (але не виключити їх зовсім).

Висновок

Сьогодні ми продовжили вивчення Redis, перейшовши від простих операцій до питання про те, як вичавити максимум продуктивності з і так дуже швидкої системи. Вчора ми переконалися, що Redis забезпечує швидкі та гнучкі структури даних та прості маніпуляції з ними, але він також успішно справляється і з більш складними завданнями, надаючи вбудовані функції публікації-підписки та бітові операції. Крім того, він налаштовується в широких межах, підтримуючи різноманітні параметри довговічності та реплікації, що дозволяє адаптувати його до різних потреб. Нарешті, для Redis

є ряд написаних сторонніми розробниками розширень, зокрема фільтри Блума та кластери.

На цьому завершуємо розгляд основних операцій сервера структур даних Redis. Завтра ми займемося іншою справою, а саме скористаємося Redis як наріжним каменем у системі багатостороннього зберігання даних, куди увійдуть ще CouchDB та Neo4j.

Завдання для індивідуальної роботи

1. Знайдіть відомості про різні патерни обміну повідомленнями і з'ясуйте, скільки з них реалізовано в Redis.

2. Запустіть скрипт завантаження даних про ISBN, відключивши миттєві знімки та дозапис у файл. Потім спробуйте запустити його, задавши параметр appendfsync рівним always, і подивіться на різницю в швидкості.

3. Візьміть свій улюблений каркас для розробки веб-застосунків і спробуйте з його допомогою розробити просту службу скорочення URL-адрес, використовуючи Redis як сховище даних. Сайт повинен пропонувати поле для введення URL-адреси та реалізовувати переадресацію з короткого URL на вихідний. Підвищіть надійність зберігання за допомогою кластера Redis з кількох вузлів з реплікацією головний-підлеглий.

Лекція 17. Комбінування Redis з іншими базами даних

Сьогодні ми завершимо розгляд бази даних Redis у взаємодії з іншими базами даних. Як ми дізналися – кожна база даних має свої сильні сторони, тому багато сучасних систем розвиваються в напрямку моделі багатостороннього зберігання (polyglot persistence), коли використовується кілька баз даних і кожна грає свою роль. Ми побудуємо проєкт, у якому CouchDB виступатиме у ролі системи, куди записуються вихідні дані (канонічний джерело даних), Neo4j керуватиме зв'язками, а Redis – використовуватиметься завантаження даних, і кешування. Відразу скажемо, що це лише один з варіантів демонстрації спільної роботи кількох баз даних, де особливі вміння кожної роблять свій внесок у досягнення спільної мети.

Служба багатостороннього зберігання

Наша служба багатостороннього зберігання працюватиме як фронтальна система для зберігання інформації про музичні гурти. Ми хочемо зберігати список назв груп, імена виконавців, що брали участь у них, і ролі кожного виконавця в групі – від провідного вокаліста до акомпаніатора на клавітарі. Кожна з трьох баз даних – Redis, CouchDB та Neo4j – відповідатиме за певні аспекти системи.

На Redis у нашій системі покладаються три важливі функції: допомога у заповненні бази CouchDB даними, кешування нещодавно внесених до Neo4j змін та швидкий пошук за неповними значеннями. Швидкість Redis та здатність зберігати різні структури

Дані дозволяють успішно застосувати її для заповнення системи даними, а вбудований механізм терміну зберігання - те що потрібно для організації кешування.

CouchDB буде у нас системою первинного запису, тобто авторитетним (канонічним) джерелом даних. Оскільки CouchDB – документна база, в ній зручно зберігати дані про групи, в які вкладені відомості про виконавців та їх ролі. А наявним у CouchDB інтерфейсом Changes API ми скористаємося для

синхронізації з нашим третім джерелом даних.

Neo4j виступатиме у ролі сховища зв'язків. Хоча прямі запити до CouchDB цілком можливі та розумні, графова база даних набагато перевершує всіх конкурентів у простоті та швидкості обходу зв'язків між вузлами. Ми зберігатимемо в ній зв'язки між групами, їхніми членами та ролями окремих членів.

Пояснимо, що під багатомовним програмуванням розуміється використання в одному проєкті кількох мов програмування. Раніше вважалося нормальним розробляти проєкт якоюсь однією мовою загального призначення. Але багатомовне програмування корисне, тому що дозволяє задіяти сильні сторони кожної мови. Наприклад, мова Scala краще пристосована для організації транзакцій без стану на стороні веб-сервера, а на Ruby простіше писати бізнес-логіку. При сумісному використанні виникає синергетичний ефект. Однією з широко відомих багатомовних систем є Twitter. Деякі з розглянутих нами баз даних власними силами підтримують багатомовне програмування – так Riak дозволяє писати mapreduce-функції на JavaScript і Erlang,

Аналогічно багатостороннє зберігання дозволяє задіяти сильні сторони різних баз даних у одній системі, що різко контрастує зі звичною практикою використання єдиної, зазвичай реляційної, СУБД. Найпростіший варіант такої системи вже широко застосовується: сховище ключів та значень (типу Redis) використовується як кеш запитів щодо відносно повільної реляційної бази даних (наприклад, PostgreSQL). Проте, як ми бачили в попередніх лекйіях, реляційні бази далеко не оптимальні для вирішення широкого класу завдань, що постійно розширюється, наприклад обходу графа.

У нашій системі кожна база даних має свою роль, але безпосередньо вони між собою не контактують. Для заповнення баз даних, організації взаємодії між ними та як простий фронтальний сервер ми будемо використовувати каркас Node.js, заснований на JavaScript.

Заповнення даними

Наше перше завдання – заповнити сховища необхідними даними. Ми зробимо це за два кроки: спочатку заповнимо даними Redis, а потім – наше канонічне джерело, CouchDB.

Як і раніше, скачаємо набір даних із сайту Freebase.com. Цього разу ми візьмемо набір group_membership з табуляторами як роздільники (http://download.freebase.com/datadumps/latest/browse/music/group_membership.tsv). У цьому файлі дуже багато інформації, але нас цікавитимуть лише поля member (ім'я виконавця), group (назва групи) та roles (ролі у групі, перераховані через кому). Наприклад, Джон Купер (John Cooper) був у групі Skillet провідним вокалістом (Lead vocalist), басистом (Bassist) та грав на акустичній гітарі (Acoustic guitar).

```
/m/0654bxy John Cooper Skillet Lead vocalist,Acoustic guitar,Bass 1996
```

Зрештою ми хочемо помістити відомості про Джона Купера та інших учасників групи Skillet в один документ CouchDB з наведеною нижче структурою. Цей документ буде доступний за URL-адресою http://localhost:5984/bands/Skillet.

```
{
"_id": "Skillet",
"name": "Skillet"
"artists": [
{
"name": "John Cooper", "role": [
“Acoustic guitar”, “Lead vocalist”,


"Bass"
]
},
...
{
"name": "Korey Cooper", "role": [
“backing vocals”, “Synthesizer”, “Guitar”,
"Keyboard instrument"
]
}
]
}
```

У файлі зберігаються відомості більш ніж про 30000 груп та 100000 виконавців. Це не так багато, але як відправна точка для побудови своєї системи цілком годиться. Зазначимо, що документовано ролі не всіх

виконавців. Набір даних неповний, але про це можна буде подумати згодом.

Трансформація даних

Ви, мабуть, запитаєте, чому потрібно спочатку завантажувати дані в Redis, а не відразу в CouchDB. Граючи роль посередника, Redis наділяє плоскі TSV-дані структурою, тому подальше завантаження до іншої бази відбувається швидше. Ми плануємо створити по одному запису для кожної групи, і Redis дозволить зробити це за один прохід по TSV-файлу (у якому одна група згадується стільки разів, скільки в ній є учасники – кожен учасник представлений в окремому рядку). Якщо додавати учасників по одному в CouchDB, то виникне пробуксування при оновленні, оскільки при вставці двох учасників однієї групи ми намагатимемося одночасно створити або оновити один і той же документ, внаслідок чого системі доведеться повторювати вставку, коли одна зі спроб завершиться помилкою, виявленою механізмом контролю версій, вбудованим у CouchDB.

Ця стратегія має один недолік – обмеження, пов'язане з тим, що Redis зберігає весь набір даних у пам'яті. Втім, цю проблему можна вирішити за допомогою простого кластера із узгодженим хешуванням, з яким ми познайомилися другого дня.

Завантаживши дані, переконайтеся, що встановлений Node.js та його менеджер пакетів Node Package Manager (npm). Після цього встановіть три пакети NPM: redis, csv та hiredis (необов'язковий написаний на C драйвер для Redis; він здатний суттєво прискорити взаємодію у Redis).

```
$npm install hiredis redis csv
```

Потім переконайтеся, що сервер Redis прослуховує стандартний порт 6379, або змініть функцію createClient() у всіх скриптах, прописавши в ній правильний порт. Для завантаження даних у Redis запустіть показаний нижче скрипт Node.js з того ж каталогу, в якому знаходиться файл TSV (ми припускаємо, що він називається group_membership.tsv). Усі написані на JavaScript скрипти досить об'ємні, тому повністю ми їх не наводимо. Повний код можна завантажити із сайту Pragmatic Bookshelf. Тут ми зупинимося

тільки на основних моментах. Завантажте та виконайте наступний файл:

```
$node pre_populate.js
```

Цей скрипт перебирає всі рядки TSV-файлу, витягуючи ім'я виконавця, назву групи та ролі виконавця у цій групі. Потім ці дані додаються до бази Redis (порожні значення пропускаються).

У Redis ключ групи має вигляд “band:BandName”. Скрипт додає ім'я виконавця у множину. Таким чином, ключ "Band: Beatles" буде посилатися на множину значень ["John Lennon", "Paul McCartney", "George Harrison", "RingoStarr”]. Так само, ключі, що відповідають виконавцям, містять назву групи та множину ролей. Наприклад, ключ “artist:Beatles: Ringo Starr” містить множину [“Drums”] (Ударні).

Інший код потрібний просто для підрахунку числа оброблених рядків та виведення результатів на екран.

redis/pre_populate.js

```
      csv().
      fromPath( tsvFileName, { delimiter: '\t', quote: '' }). on('data',
function(data, index) {
      var
      artist = data[2], band = data[3],
      roles = buildRoles(data[4]);
      if( band === '' || artist === '' ) { trackLineCount();
      return true;
      }
      redis_client.sadd('band:' + band, artist); roles.forEach(function(role)
{
      redis_client.sadd('artist:' + band + ':' + artist, role);
      });
      trackLineCount();
      }).
```

Переконатися в тому, що цей скрипт дійсно завантажив дані в Redis, можна запустити програму redis-cli та виконавши команду RANDOMKEY. Ми очікуємо отримати ключ, який починається з band: або artist:. Значення влаштовує нас будь-яке, крім (nil).

Завантаживши дані в Redis, одразу переходьте до наступної частини. Якщо в цей момент зупинити Redis, то дані, швидше за все, будуть втрачені, якщо ви, звичайно, не встановили режим довговічності, відмінний від замовчування, або не виконали команду SAVE.

Вставка у канонічну систему

CouchDB має роль канонічної системи, в яку записуються дані. У будь-

якому конфлікті між Redis, CouchDB чи Neo4j виходить переможцем CouchDB. Хороша канонічна система повинна зберігати всі дані, необхідні перебудови будь-якого джерела даних на підвідомчої «території».

Перевірте, що CouchDB прослуховує стандартний порт 5984, або змініть номер порту в рядку require('http').createClient(5984, 'localhost') показаного нижче скрипта. Redis повинен бути у тому стані, у якому ми залишили його у попередньому розділі. Завантажте та виконайте наступний файл.

```
$node populate_couch.js
```

Якщо спочатку ми читали TSV-файл і завантажували дані з нього в Redis, то тепер читатимемо дані з Redis і завантажувати їх у CouchDB. Нам не знадобляться спеціальні драйвери для CouchDB, тому що взаємодія з нею відбувається через простий REST інтерфейс, а в Node.js вже вбудовано бібліотеку для роботи з протоколом HTTP.

У послідовній програмі код, який запитує базу даних, чекає на відповідь і обробляє результати, пишеться наступним чином:

```
results = database.some_query() for value in results
# обробити кожне значення
end
# цей код виконується лише після завершення циклу обробки результатів.
```

У паралельно-організованій програмі цикл передається у вигляді функції або блоку коду. Поки база даних займається своєю справою, переважна більшість програми продовжує працювати. І лише після того, як буде отримано результат, передана функція чи блок виконується.

```
database.some_query do |results| for value in results
# обробити кожне значення
end
end
# цей код продовжує працювати, поки база даних виконує запит.
```

Паралельна програма роботи з базою даних – справа принципово складна. Втім, це дуже ефективний метод звернення до баз. Майже у всіх популярних мовах програмування є та чи інша бібліотека, що разпаралелює код. У Ruby є EventMachine, в Python - Twisted, в Java - бібліотека NIO, в C # - Interlace, а в JavaScript - Node.js.

У наведеному нижче блоці ми виконуємо команду Redis KEYS bands:*,

щоб отримати список усіх назв груп. Якби набір даних був по-справжньому великим, можна було б обмежити запит (наприклад, опитати спочатку ключ bands:A*, щоб вибрати лише назви, починаються на a, тощо. буд.). Потім для кожної групи ми просимо множину виконавців і виділяємо з ключа назву групи, видаляючи префікс bands:.

redis/populate_couch.js

```
redisClient.keys('band:*', function(error, bandKeys) { totalBands =
bandKeys.length;
var
readBands = 0, bandsBatch = [];
bandKeys.forEach(function(bandKey) {
// підрядок 'band:'.length містить назву групи
var bandName = bandKey.substring(5);
redisClient.smembers(bandKey, function(error, artists) {
```

Потім ми отримуємо ролі кожного виконавця, Redis повертає їх як масиву масивів (ролі кожного виконавця в окремому масиві). Для цього ми можемо зібрати команди SMEMBERS в одному масиві roleBatch та виконати їх пакетно в рамках однієї команди MULTI. Фактично, це один конвеєрний запит виду:

```
MULTI
SMEMBERS “artist:Beatles:John Lennon” SMEMBERS “artist:Beatles:Ringo
Starr”
EXEC
```

Далі ми створюємо пакет із 50 документів CouchDB. Ми вирішили вчинити саме так, тому що потім передаємо весь пакет команді CouchDB /_bulk_docs, завдяки чому вставка здійснюється дуже швидко.

redis/populate_couch.js

```
redisClient. multi(roleBatch). exec(function(err, roles)
{
var
i = 0,
artistDocs = [];

// побудувати піддокументи для виконавців
artists.forEach( function(artistName) {
artistDocs.push({ name: artistName, role : roles[i++] });
});

// додати новий документ, що описує групу, пакет,
// який буде виконано пізніше
bandsBatch.push({
_id: couchKeyify( bandName ), name: bandName,
artists: artistDocs
});
```

Заповнивши базу даних про музичні групи, ми маємо місце, де

зберігаються всі необхідні системі дані. Ми знаємо назви гуртів, імена музикантів, що беруть участь у них, та їх ролі в групі. Можна дослідити щойно заповнену канонічну базу даних CouchDB, надіславши запит за адресою http://localhost:5984/_utils/database.html?bands.

Сховище зв'язків

Наступною в нашому списку є служба на основі Neo4j, яку ми плануємо використовувати для відстеження зв'язків між виконавцями та їх ролями. Звичайно, можна було б опитувати CouchDB безпосередньо, створивши представлення, але сформулювати складні запити щодо зв'язків так не вдасться. Якщо Уейн Койн із Flaming Lips втратить перед концертом свій терменвокс, то він зможе позичити його у Чарлі Клоузера з Nine Inch Nails, який також грає на терменвоксі. Так само можна було б знайти музикантів, які мають одночасно кілька талантів, навіть якщо в різних групах вони виступають у різних амплуа, – для цього досить простого обходу вузлів.

Підготувавши вихідні дані, ми тепер маємо подбатипро синхронізацію Neo4j з CouchDB у разі, коли дані в канонічній системі змінюються. Ми збираємося вбити одним пострілом двох зайців, написавши службу, яка передаватиме в Neo4j всі зміни, що відбуваються в CouchDB з моменту створення бази.

Ми також хочемо занести у Redis ключі всіх груп, виконавців та ролей, щоб можна було швидко знаходити ці дані згодом. На щастя, це ті дані, які ми вже помістили в CouchDB, що позбавляє нас від окремого кроку для початкового заповнення Neo4j і Redis.

Перевірте,що Neo4j прослуховує порт 7474, або змініть номер порту функції createClient(). CouchDB та Redis мають бути вже запущені. Завантажте та виконайте нижчезазначений файл. Цей скрипт буде працювати, поки ви не перервете його явно.

```
$node graph_sync.js
```

Це сервер, який безперервно опитує CouchDB щодо змін, що відбулися (як це робиться, ми бачили в розділі, присвяченому CouchDB). Виявивши

зміну, ми робимо дві речі: копіюємо дані в Redis і Neo4j. Наступний код заповнює Redis, виконуючи каскад функцій зворотного дзвінка. Спочатку копіюється група із ключем “band-name:Band Name”. І далі так само копіюється ім'я виконавця та його ролі.

Це дозволить нам шукати по неповних рядках. Наприклад, команда KEYS band-name:Bea* поверне такі групи: Beach Boys, Beastie Boys, Beatles тощо.

redis/graph_sync.js

```
function feedBandToRedis(band) { redisClient.set('band-name:' +
band.name, 1); band.artists.forEach(function(artist) {
redisClient.set('artist-name:' + artist.name, 1);
artist.role.forEach(function(role){
redisClient.set('role-name:' + role, 1);
```

У наступному блоці ми заповнюємо Neo4j. У файлі neo4j_caching_client.js, який можна завантажити з сайту цієї книги, знаходиться відповідний драйвер. Він використовує вбудовану в Node.js бібліотеку для роботи з HTTP, щоб підключитися до REST-інтерфейсу Neo4j; додатково ми обмежили кількість з'єднань, що одночасно відкриваються клієнтом. Наш драйвер також використовує Redis, щоб відстежувати зміни, зроблені у графі Neo4j, не надсилаючи окремого запиту. Це третє застосування Redis у нашій системі; перше – трансформація даних заповнення бази CouchDB, друге – швидкий пошук за назвами груп.

У цьому коді створюються вузли груп (якщо ще не створені), потім вузли виконавців (якщо необхідно), та був ролі. На кожному кроці принагідно створюються нові зв'язки, так що вузол The Beatles пов'язується з вузлами Джона, Пола, Джорджа та Рінго, які у свою чергу пов'язані з вузлами ролей.

redis/graph_sync.js

```
function feedBandToNeo4j(band, progress) { var
lookup = neo4jClient.lookupOrCreateNode, relate =
neo4jClient.createRelationship;

lookup('bands', 'name', band.name, function(bandNode) {
progress.emit('progress', 'band'); band.artists.forEach(function(artist) {
lookup('artists', 'name', artist.name, function(artistNode){
progress.emit('progress', 'artist'); relate(bandNode.self, artistNode.self,
'member', function(){
```

```
      progress.emit('progress', 'member');
      });
      artist.role.forEach(function(role){ lookup('roles', 'role', role,
function(roleNode){
      progress.emit('progress', 'role');relate(artistNode.self, roleNode.self,
'plays', function(){
      progress.emit('progress', 'plays');
```

Запустіть цю службу в окремому вікні. Будь-яка зміна в CouchDB, за якої додається новий виконавець або нова роль існуючого виконавця призводить до створення нового зв'язку в Neo4j і, можливо, нових ключів у Redis. Поки ця служба працює, всі три бази залишаються синхронізованими.

Зайдіть у веб-консоль CouchDB та відкрийте групу. Внесіть до бази даних якусь зміну, наприклад, додайте до групи нового учасника (зробіть себе учасником «Бітлз»!) або призначте нову роль виконавцю. Слідкуйте за тим, що виводить скрипт graph_sync. Потім відкрийте консоль Neo4j та спробуйте знайти нові зв'язки у графі. Якщо ви додали нового учасника групи, то має з'явитися зв'язок між його вузлом та вузлом групи. У поточній реалізації зв'язку не видаляються, але додавання операції Neo4j DELETE не вимагатиме повної обробки скрипту.

Веб-служба

Ми збираємося написати простий веб-додаток, який дозволить користувачам шукати музичні гурти. Для будь-якої групи будуть у вигляді посилань перераховані її учасники, а клацання на учасника виведе відомості про нього, точніше, всі його амплуа в групі. Крім того, для кожної ролі виконавця будуть виведені всі інші виконавці, що є в системі, які грають таку ж роль у своїй групі.

Наприклад, пошук гурту Led Zeppelin поверне Джиммі Пейджа (Jimmy Page), Джона Пола Джонса (John Paul Jones), Джона Бонема (John Bonham) та Роберта Планта (Robert Plant). Клацнувши на Джиммі Пейджа, ми дізнаємося, що він грає на гітарі, а заразом отримаємо купу інших гітаристів, наприклад The Edge з групи U2.

Щоб спростити створення веб-програми, встановимо ще два пакети: bricks (простий веб-каркас) і mustache (бібліотека для роботи з шаблонами).

```
$ npm install bricks mustache
```

Переконайтеся, що всі бази даних працюють, після чого запустіть веб-сервер. Завантажте та виконайте наступний файл:

```
$node band.js
```

Сервер прослуховує порт 8080, тому якщо ви зайдете в браузері на адресу http://localhost:8080/, то побачите просту пошукову форму.

Нижче наведено код створення веб-сторінки з інформацією про групу. Кожен URL виконує у нашому крихітному HTTP-сервері свою роль. URL http://localhost:8080/band приймає як параметр ім'я групи.

redis/bands.js

```
appServer.addRoute("^/band$", function(req, res) { var
bandName = req.param('name'),
bandNodePath = '/bands/' + couchUtil.couchKeyify( bandName ),
membersQuery = 'gV[[name:"'+bandName+'"]]'
+ '.out("member").in("member").uniqueObject.name';getCouchDoc(
bandNodePath, res, function( couchObj ) {
gremlin( membersQuery, function(graphData) { var artists = couchObj &&
couchObj['artists'];
var values= { band: bandName, artists: artists, bands: graphData }; var
body = '<h2>{{band}} Band Members</h2>';
body += '<ul>{{#artists}}';
body += '<li><a href="/artist?name={{name}}">{{name}}</a></li>'; body +=
'{{/artists}}</ul>';
body += '<h3>You may also like</h3>'; body += '<ul>{{#bands}}';
body += '<li><a href="/band?name={{.}}">{{.}}</a></li>'; body +=
'{{/bands}}</ul>';
writeTemplate(res, body, values);
```

Якщо ви введете у формі назву групи Nirvana, серверу буде відправлено запит на адресу http://localhost:8080/ band?name=Nirvana. Наведена вище функція відобразить HTMLсторінку (її шаблон знаходиться у зовнішньому файлі template.html). На цій сторінці будуть виведені дані про всіх виконавців, що входять до групи, взяті безпосередньо з CouchDB. Крім того, показано список рекомендованих груп; для його отримання ми виконуємо Gremlin-запит до графа Neo4j. Для групи Nirvana цей запит має такий вигляд:

```
gVfilter{it.name=="Nirvana"}.out("member").in("member").dedup.name
```

Іншими словами, виходячи з вузла Nirvana ми отримуємо унікальні назви груп, учасники яких пов'язані з учасниками групи Nirvana. Наприклад, Дейв Грол (Dave Grohl) грав як у Nirvana, так і у Foo Fighters, тому група Foo Fighters буде включена до списку. На наступній сторінці –

http://localhost:8080/artist – виводяться відомості про виконавця.

redis/bands.js

```
appServer.addRoute("^/artist$", function(req, res) { var
artistName = req.param('name'),


rolesQuery = 'gV[[name:"'+artistName+'"]].out("plays").role. \
uniqueObject ',
bandsQuery = 'gV[[name:"'+artistName+'"]].in("member").name. \
uniqueObject ';
gremlin( rolesQuery, function(roles) { gremlin( bandsQuery,
function(bands) {
var values = { artist: artistName, roles: roles, bands: bands }; var
body = '<h3>{{artist}} Performs these Roles</h3>';body += '<ul>{{#roles}}';
body += '<li>{{.}}</li>'; body += '{{/roles}}</ul>';
body += '<h3>Play in Bands</h3>'; body += '<ul>{{#bands}}';
body += '<li><a href="/band?name={{.}}">{{.}}</a></li>'; body +=
'{{/bands}}</ul>';
writeTemplate(res, body, values);
```

Тут ми виконуємо два Gremlin-запити. Перший виводить усі ролі виконавця, другий – список груп, у яких брав участь. Наприклад, для Джеффа Уорда (Jeff Ward) буде показано, що він грає на ударних (Drummer) і брав участь у групах Nine Inch Nails і Ministry.

Зауважте, що на двох описаних вище сторінках ми виводимо перехресні посилання. Посилання у списку виконавців на сторінці /bands ведуть на сторінку /artist вибраного виконавця та навпаки. Але можна трохи спростити пошук.

redis/bands.js

```
appServer.addRoute("^/search$", function(req, res) { var query =
req.param('term');
redisClient.keys("band-name:"+query+"*", function(error, keys) { var
bands = [];
keys.forEach(function(key){ bands.push(key.replace("band-name:", ''));
});
res.write( JSON.stringify(bands) ); res.end();
```

Тут ми просимо у Redis усі ключі, що відповідають початковій частині рядка, як, наприклад, "Bea*" вище. Redis виводить дані у форматі JSON. Шаблон template.html включає код jQuery, який наділяє пошукове поле функцією автозавершення.

Розвиток веб-служби

Цей порівняно невеликий скрипт реалізує лише необхідні функції. Його можна розвинути у багатьох напрямках. Наприклад, до складу

рекомендованих входять лише групи першого порядку (ті, у яких брали участь музиканти, які входять до поточної групи). Але можна отримати цікаві результати, написавши запит для отримання груп другого порядку, наприклад:

```
gVfilter{it. name=='Nine Inch Nails'}.out('member').in('member').dedup.
loop(3){ it.loops <= 2 }.name.
```

Крім того, ми не маємо форми, яка дозволяє змінити інформацію про групу. Додати таку функціональність не надто складно, тому що ми вже написали код заповнення CouchDB у скрипті populate_couch.js, а будь-яка зміна CouchDB автоматично відображається в Neo4j і Redis, якщо працює служба graph_sync.js.

Якщо вправи з багатостороннім зберіганням вас зацікавили, можна піти ще далі. Наприклад, додати сховище даних на основі PostgreSQL7 для перетворення цих даних на схему типу «зірка», що дозволяє аналізувати дані з різних вимірів, наприклад: найпоширеніший музичний інструмент чи порівняння середньої кількості учасників групи із середнім числом інструментів. Можна також додати сервер Riak для зберігання зразків творів групи або сервер HBase для побудови системи обміну повідомленнями, за допомогою якої користувачі можуть відстежувати історію своїх симпатій та антипатій, або розширення MongoDB, що приносить географічний аспект.

Або можете взагалі переписати проєкт іншою мовою, з використанням іншого веб-каркасу чи набору даних. Можливостей розвитку стільки, скільки існує комбінацій баз даних та технологій їх створення, тобто декартовий добуток усіх проєктів з відкритим вихідним кодом.

Висновок

Сьогодні ви відчули, як виглядає майбутнє систем управління даними у світі, що рухається у напрямку від однієї великої реляційної СУБД до моделі з кількома спеціалізованими базами даних. Ми об'єднали ці бази за допомогою коду, що разпаралелює хід виконання коду, і варто зазначити, що саме в цю сторону зміщуються переваги розробників у частині взаємодії з

базами даних.

Важливість Redis у цій моделі не слід недооцінювати. Безумовно, Redis не має ніякої функціональності, якої не могла б надати будь-яка з розглянутих баз даних окремо, натомість у ній є швидкі структури даних. Ми змогли перетворити плоский файл у ряд осмислених структур даних, що необхідно як із заповненні системи даними, і під час їх передачі. Причому реалізувати це вдалося легко та швидко. Навіть якщо в цілому модель багатостороннього зберігання не викликала у вас інтересу, при розробці будь-якої системи є сенс мати на увазі Redis.

Redis – компактне сховище ключів та значень (або структур даних), що споживає небагато ресурсів. Воно схоже на багатофункціональні інструменти, що включають ніж, відкривалку, штопор та інші пристосування. Як і вони, Redis придатна для вирішення найрізноманітніших, часто несподіваних завдань. До того ж, сховище Redis працює швидко, просто в експлуатації і забезпечує таку довговічність даних, яку ви налаштуєте. Redis рідко використовується як автономна база даних, частіше вона входить до складу багатосторонньої екосистеми, де застосовується для трансформації даних, кешування запитів або управління чергами повідомлень (завдяки вбудованим блокуючим командам).

Сильні сторони Redis. Очевидна перевага Redis- швидкодія, чим можуть похвалитися багато подібних сховищ ключів і значень. Але, на відміну від більшості таких сховищ, Redis ще вміє зберігати складові значення– списки, хеші та множини – та застосовувати до них спеціальні операції. Мало того, в Redis ще можна тонко налаштовувати довговічність даних, віддаючи перевагу надійності зберігання або швидкості. Вбудована реплікація типу головний-підлеглий – ще один зручний спосіб підвищити довговічність, не приносячи швидкодії в жертву повільного дозапису файлу при кожній операції. До того ж, реплікація підвищує продуктивність систем, у яких основне навантаження посідає операції читання.

Слабкі сторони Redis. Своєю швидкодією Redis зобов'язана насамперед

тому, що зберігає всі дані в пам'яті. Хтось назве це обманом, тому що база даних, яка взагалі не звертається до диска, природно працюватиме швидко. Будь-якому сховищу в оперативній пам'яті притаманна проблема довговічності даних; якщо зупинити базу, не скинувши дані на диск, дані будуть втрачені. Навіть якщо встановити синхронний режим дозапису у файл при кожній операції, все одно при відтворенні файлу відновлення даних з терміном зберігання, що минув, не цілком надійно – це характерно для будь-якого механізму відтворення подій, прив'язаних до часу. Втім, щиро кажучи, ця проблема радше теоретична, ніж практична.

Redis також не підтримує набори даних, розмір яких перевищує обсяг доступноїоперативної пам'яті (підтримка віртуальної пам'яті Redis буде виключена). В даний час розробляється кластер Redis, який дозволить вийти за межі обмеження на обсяг ОЗП в одному комп'ютері, але поки що користувачі, які бажають організувати кластер, повинні самі написати відповідний клієнт (як драйвер на Ruby, з яким ми працювали другого дня).

Завдання для індивідуальної роботи

1. Змініть програму імпорту так, щоб до складу даних про групу включалися ще дати приходу та догляду кожного учасника. Збережіть ці відомості в піддокументі виконавця у CouchDB та відображайте на сторінці виконавця.

2. Додати в системі ще й базу даних Riak для зберігання кількох зразків творчості групи в GridFS, так що користувач може прослухати одну-дві композиції. Якщо для групи зберігається хоча б одна композиція, увімкніть посилання на неї до веб-програми. Забезпечте синхронізацію між Riak та CouchDB.

Лекція 18. Порівняння різних NoSQL баз даних

Опрацювавши матеріал про різноманітні NoSQL бази даних, ви могли оцінити переваги цих систем. Ми вважаємо, що майбутнє систем управління даними належить моделі багатостороннього зберігання (коли в одному проєкті використовується відразу кілька баз даних), а монопольне переважання універсальних реляційних СУБД зійде нанівець.

Ми вже вивчили деталі кожної бази та відзначили деякі подібності та відмінності. Тепер поговоримо про їх внесок у загальну панораму систем зберігання даних, що постійно розширюється.

Ми бачили, що за способом зберігання бази даних можна приблизно розбити на п'ять жанрів: реляційні, сховища ключів та значень, стовпцеві, документні та графові. Ненадовго затримаємось і згадаємо, чим вони відрізняються один від одного і для яких завдань підходять чи не підходять – коли їх має сенс використовувати, а коли треба обирати інші типи баз даних.

Реляційні бази даних

Це класичний та найпоширеніший варіант. Реляційні системи управління базами даних (РСУБД) засновані на теорії множин, дані в них зберігаються у вигляді двовимірних таблиць, що складаються із рядків і стовпців. Реляційні бази суворо контролюють типи даних і зазвичай допускають зберігання чисел, рядків, дат та неінтерпретованих двійкових об'єктів (BLOB'ів), хоча, як бачили, PostgreSQL підтримує такі розширення, як масив і куб.

Переваги. Завдяки структурованій природі даних реляційні базимає сенс використовувати, коли структура даних наперед відома, а способи їх можливого використання – ні. Іншими словами, ви готові заплатити авансом за складність організації, сподіваючись, що в майбутньому це окупиться гнучкістю запитів. Багато завдань бізнесу зручно моделюються таким чином: замовлення, постачання, облік складських запасів, кошики покупок та інші. Заздалегідь невідомо, як згодом потрібно витягувати дані (наприклад,

скільки замовлень ми обробили в лютому?), але дані за своєю природою регулярні, і контролювати цю регулярність дуже корисно.

Недоліки. Якщо структура даних змінна чи характеризується глибокої вкладеністю, то реляційна СУБД – найкращий помічник. У ній необхідно заздалегідь визначити схему, що дуже важко зробити, якщо всі записи структурно відрізняються один від одного. Подумайте, як би ви почали розробляти базу даних для опису всіх живих істот у природі. Спроба створити повний список ознак ("має волосся", "кількість ніг", "відкладає яйця" тощо) приречена на провал. У такому разі потрібна база даних, яка не накладає таких суворих обмежень на дані, що зберігаються.

Сховища ключів та значень

Сховище ключів та значень – найпростіша з усіх розглянутих нами моделей. У ньому прості ключі відображаються на, можливо, складніші значення, як у гігантській хеш-таблиці. Завдяки відносній простоті бази даних цього жанру забезпечують максимальну гнучкість реалізації. Пошук у хеш-таблиці проводиться швидко, тому у разі Redis швидкодія була поставлена в основу. Крім того, пошук у хеш-таблицях легко розподіляється на кілька машин, і в Riak цей факт використаний для побудови простих в управлінні кластерів. Зрозуміло, простота може бути мінусом, якщо до моделювання даних пред'являються складні вимоги.

Переваги. Оскільки потреба підтримувати індекси відсутня абоневелика, то при проєктуванні сховищ ключів та значень часто закладається горизонтальна масштабованість, дуже висока швидкодія або те й інше разом. Особливо гарні вони для завдань, де між даними немає великої кількості зв'язків. Наприклад, у веб-застосунку цьому критерію відповідають сеанси користувачів; дії одного користувача в сеансі практично не пов'язані з діями інших користувачів.

Недоліки. Через відсутність індексів та засобів сканування КЗ-сховища мало чим допоможуть, якщо потрібно пред'являти запити, відмінні від найпростіших операцій CRUD (створення, читання, оновлення, видалення).

Стовпцеві бази даних

Стовпцеві бази даних (також бази даних із сімействами стовпців) мають багато спільних рис із КЗ-сховищами та РСУБД. Як і КЗ-сховищах, значення запитуються за ключом. Як і в реляційних базах, значення групуються в нуль або більше стовпців, хоча в кожному рядку може бути заповнена довільна кількість стовпців. Але на відміну від того й іншого, у стовпцевих базах дані зберігаються по стовпчикам, а не рядкам. Додавання нових стовпців обходиться дешево, версіонування реалізується тривіально, і зберігання відсутніх значень місце не витрачається. Розглянута база даних HBase – класичний приклад цього жанру.

Переваги. Стовпцеві бази даних зазвичай розроблялися з метою забезпечити горизонтальну масштабованість. Тому вони особливо хороші для завдань, пов'язаних із «великими даними», коли база розміщується в кластері з десятків, сотень, а то й тисяч вузлів. Крім того, в них зазвичай вбудовується підтримка стиснення та версіонування. Канонічний приклад завдання, орієнтованої зберігання даних по стовпцям, – індексування веб-сторінок. Сторінки в Інтернеті містять багато тексту (і тут дуже корисно стиснення), певною мірою взаємопов'язані і змінюються з часом (ось де на допомогу приходить версіонування).

Недоліки. У різних стовпцевих баз даних різні можливості і, отже, різні недоліки. Але всім їм властива одна спільна характеристика: краще проєктувати схему з урахуванням майбутніх запитів. Це означає, що потрібно апріорі уявляти, як будуть використовуватися дані, а не тільки те, що вони містять. Якщо способи використання даних заздалегідь невідомі – наприклад, потрібні довільні звіти, то стовпцева база даних – не оптимальний варіант.

Документні бази даних

У документній базі даних об'єкт, що зберігається, може містити довільний набір полів і навіть допускає вкладеність будь-якого рівня. Зазвичай такі об'єкти подаються в нотації JavaScript Object Notation (JSON), прийнятої як MongoDB, так і CouchDB, хоча ця вимога жодною мірою не є

концептуально значущою. Оскільки документи не пов'язані один з одним так тісно, як у реляційних базах даних, їх порівняно просто сегментувати та реплікувати на кілька серверів, тому часто зустрічаються розподілені реалізації. Система MongoHQ ставить за мету підвищити доступність за рахунок створення ЦОД, здатних підтримувати гігантські набори даних масштабу веб. З іншого боку, CouchDB орієнтована на простоту та довговічність даних, а доступність досягається за рахунок реплікації типу головний-головний на вузли з високим рівнем автономності.

Переваги. Документні бази даних хороші для завдань, де предметна область характеризується високим рівнем мінливості. Якщо наперед невідомо, як виглядають дані, то документна база – саме те, що потрібно. Крім того, документи за своєю природою часто добре лягають на об'єктно-орієнтовані моделі програмування. Отже, зменшується неузгодженість інтерфейсів під час переходу від моделі бази даних до об'єктної моделі програми.

Недоліки. Якщо ви звикли до складних запитів із з'єднаннями в реляційній базі даних з добре нормалізованою схемою, то в документній базі цих можливостей не вистачатиме. Документ зазвичай містить усю або більшу частину інформації про об'єкт. Якщо в реляційній базі природно прагнення нормалізувати дані з метою усунення дублювання, що може призвести до неузгодженості, то документних базах денормалізоване зберігання – це норма.

Графові бази даних

Графові бази даних – порівняно новий, що швидко розвивається клас систем, у яких наголос робиться скоріш встановлення довільних зв'язків між даними, ніж фактичні значення. Прикладом може бути проєкт із відкритим вихідним кодом Neo4j, який дедалі частіше використовується у багатьох соціальних мережах. На відміну з інших стилів баз даних, де колекції подібних об'єктів групуються у загальних контейнерах, у графових базах дані зберігаються у вільнішій формі; обробка запиту зводиться до прямування по

ребрах, що з'єднує дві вершини, інакше кажучи, до обходу вузлів. У міру використання у дедалі більшій кількості проєктів графові бази даних починають застосовуватися як для побудови простих соціальних додатків, але й більш специфічних цілей, наприклад: системи рекомендації, списки управління доступом і географічні дані.

Переваги. Графові бази даних начебто спеціально придумані для додатків, які характеризуються наявністю мережі даних. Типовий приклад – соціальна мережа, де вузли репрезентують користувачів, між якими існують різні зв'язки. Моделювання таких даних за допомогою інших засобів найчастіше є складною проблемою, але графові бази даних справляються з цим завданням на ура. До того ж вони чудово лягають на об'єктно-орієнтовану систему. Все, що можна змоделювати на дошці, можна змоделювати за допомогою графа.

Недоліки. Через наявність великої кількості зв'язків між вузлами графові бази даних зазвичай погано пристосовані до розміщення кількох машинах. Швидкий обхід графа був би неможливим, якби довелося надсилати по мережі запити іншим вузлам, тому графові бази даних масштабуються погано. Швидше за все, графова база буде лише однією з частин вашої системи, в якій зберігаються лише зв'язки, тоді як основні дані знаходяться в іншому місці.

Висновок

На початку курсу ми вже говорили, що дані – марні, поки з них не вилучено інформацію (більш грубо можна сказати, що сьогодні в даних закопані великі гроші). Від того, яку базу даних ви оберете, залежить простота збору, зберігання, аналізу та очищення даних.

Визначитися з вибором бази даних нерідко виявляється складніше, ніж просто вирішити який жанр найкраще підходить для даних конкретної предметної області. Очевидно, що соціальний граф найпростіше зберігати у графовій базі даних, але якщо йдеться про Facebook, то даних надто багато –

у графовій базі вони просто не помістяться. Швидше, ви зупинитися на якійсь системі, пристосованій до «великих даних», наприклад, HBase або Riak. Таким чином, ви будете змушені вибрати стовпцеву базу або сховище ключів та значень. Або взяти інший приклад: реляційна база даних начебто ідеально підходить для реалізації банківських транзакцій, але ж і Neo4j підтримує ACID-транзакції, тому вибір є.

Ці приклади показують, що у виборі бази даних – чи кількох баз даних, – оптимальних для предметної області, слід брати до уваги не лише жанр. Взагалі, зі зростанням обсягу даних, гранична ємність деяких баз стає обмеженням. Стовпцеві бази даних часто розподіляються по кількох ЦОД і підтримують великі дані максимального розміру, тоді як ємність графових баз найменша з усіх. Втім, це правило не є універсальним. Так, Riak – великомасштабне сховище ключів та значень, призначене для сегментування на сотні та тисячі вузлів, тоді як Redis проєктувалася для роботи в одному вузлі, але з можливістю реплікації типу головний-підлеглий та сегментування, реалізованого на рівні клієнта.

При виборі бази даних потрібно враховувати такі характеристики, як довговічність, доступність, узгодженість, масштабованість і безпека. Ви повинні вирішити, чи потрібна можливість пред'являти довільні запити, чи буде достатньо технології mapreduce. Ви віддаєте перевагу HTTP/REST-інтерфейсу або потребуєте драйвера для спеціалізованого двійкового протоколу? Навіть такі, на перший погляд, другорядні питання, як існування програми масового завантаження даних, можуть виявитися важливими у конкретному випадку.

У сучасних програмах проблема масштабованості змістилася переважно в область управління даними. Ми досягли точки, коли вибір мови програмування, каркасу, операційної системи і навіть обладнання та порядку експлуатації (завдяки хостингу віртуальних машин та хмарі) настільки спростився та здешевився, що практично перестав бути проблемою та визначається особистими уподобаннями в тій же мірі, що й необхідністю.

Якщо ви хочете створювати додатки, що масштабуються, то повинні замислитися перш за все про вибір бази (або баз) даних – саме тут тепер знаходиться вузьке місце. Допомогти вам зробити правильний вибір – саме в цьому полягає головна мета цього курсу.

Список використаної літератури

1. David Thomas та Andrew Hunt. Programming Ruby: The Pragmatic Programmer's Guide. Addison-Wesley, Reading, MA, 2001.

2. Bruce A. Tate. Seven Languages в Seven Weeks: Прагматична техніка для розробок програмування Languages. The Pragmatic Bookshelf, Raleigh, NC та Dallas, TX, 2010.

3. Эрик Редмонд, Джим. Р. Уилсон Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL. Под редакцией Жаклин Картер / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2013. – 384с.: ил.

4. Кайл Бэнкер MongoDB в действии / пер. с англ. Слинкина А.А. М.: ДКМ Пресс, 2010. 394 с.

5. Редмонд Уилсон: Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL / пер. с англ. М.: ДМК-Пресс, 2017. 384 с.

6. Робинсон Ян, Вебер Джим, Эифрем Эмиль Графовые базы данных: новые возможности для работы со связанными данными / пер. с англ. Р.Н. Рагимова; науч. ред. А. Н. Кисилев. 2-е изд. М.: ДМК Пресс, 2016. 256 с.

7. Фаулер Мартин, Садаладж Прамодкумар Дж. NoSQL: новая методология разработки нереляционных баз данных. Пер. с англ. М.: ООО «И.Д. Вильямс», 2013. 192 с.