빅데이터 시대를 위한 NoSQL 핵심 가이드

저자 _ Tatsuya Sasaki
편역 _ 손은영
감수 _ LG CNS DB 관리팀

1판 1쇄 발행일 _ 2011년 12월 30일

펴낸이 _ 장미경
펴낸곳 _ 로드북
편집 _ 임성춘
디자인 _ 이호용(표지), 박진희(본문)

주소 _ 서울시 관악구 신림동 1451-15 101호
출판 등록 _ 제 2011-21호(2011년 3월 22일)
전화 _ 02)874-7883
팩스 _ 02)6280-6901
정가 _ 25,000원(미정)
ISBN _ 978-89-966598-6-0 93560 (미정)

NoSQL DATABASE FIRST GUIDE by Tatsuya Sasaki
Copyright © 2011 by Tatsuya Sasaki

All rights reserved.
First Published in Japan in 2010 by Shuwa System Co., Ltd.
This Korean edition is published by arrangement with Shuwa System Co., Ltd.
Tokyo in care of Tuttle-Mori Agency, Inc., Tokyo through EntersKorea Co., Ltd. Seoul.

이 책의 한국어판 저작권은 (주)엔터스코리아를 통한 일본의 Shuwa System Co., Ltd와의 독점 계약으로 로드북이 소유합니다.
신 저작권법에 의하여 한국 내에서 보호를 받는 저작물이므로 무단전재와 무단복제를 금합니다.

책 내용에 대한 의견이나 문의는 출판사 이메일이나 블로그로 연락해 주십시오.
잘못 만들어진 책은 서점에서 교환해 드립니다.

이메일 chief@roadbook.co.kr
블로그 www.roadbook.co.kr
Q&A roadbook.zerois.net/qna

편역자의 글

NoSQL이라는 단어가 여기저기서 들리기 시작하더니 올해부터는 빅 데이터Big Data와 함께 빠지지 않는 화두가 되고 있습니다. 여기저기서 입에 올리고는 있지만, NoSQL이 무엇인지 찾아봐도 NoSQL 데이터베이스의 하나인 Cassandra, MongoDB, HBase 등에 대해 언급된 자료는 있지만 한데 잘 정리된 자료를 찾기는 쉽지 않습니다.

도대체 NoSQL 데이터베이스가 뭐길래 빅데이터를 언급할 때마다 등장하고 새로운 시스템을 만들 때 언급이 되는 걸까요? 어떤 것인지, 왜 그리 화두가 되는 건지, 도대체 어디에 도입해야 할지, NoSQL이 대세라는데 덮어놓고 쓰면 되는건지, 그러면 그 중 어떤 NoSQL을 도입해야 하는지, 기존의 데이터베이스와는 어떻게 다른지 등등 질문도 많고 궁금해하는 분들이 많을 거라고 생각합니다. 이 책을 읽으면서 완벽하게는 아니더라도 이러한 질문에 어느 정도 답을 얻을 수 있었습니다.

이 책에서는 NoSQL 데이터베이스를 분류 별로 설명해 줄 뿐만 아니라 해당 언어를 알지 못하더라도 직관적으로 이해하기 쉬운 루비Ruby를 사용하여 설치 및 실제 이용 사례를 다양하게 보여주고 있어 개념을 좀 더 쉽게 알 수 있습니다. 또한 여러 NoSQL 데이터베이스에 대해 다루기 때문에 서로 비교를 통해 해당 데이터베이스를 더 잘 알기 쉽도록 도와 줍니다. 기업환 경에 도입하려고 하는 부분부터 신규 애플리케이션을 개발하려고 하는데 NoSQL에 관심을 가지게 된 개발자까지 다양한 분들에게 NoSQL을 소개하는 입문서로, 책을 보시는 분들에게 도움이 되면 좋겠습니다.

과연 책이 나올 수 있을까 하던 시기가 있었는데 어느새 조만간 책이 나올 때가 되었습니다. 좋은 기회를 주신 임성춘 편집장님과 이춘식 팀장님께 먼저 감사드립니다. 같이 NoSQL을 스터디하며 번역하는 데 많은 도움을 준 우리 팀원분들 고맙습니다. 그리고 언제나 힘이 되어 주는 부모님과 동생에게 고맙고 사랑한다고 말씀드리고 싶습니다.

2011년 12월
손은영

지난 수십 년 동안 정보시스템의 데이터 모델을 수립하기 위한 방법으로 관계형 데이터베이스는 가장 널리 활용된 기술이며, 대부분의 개발자는 데이터베이스라는 단어를 관행적으로 관계형 데이터 모델에 기반한 데이터구조의 수립을 연상하곤 합니다. 하지만, 소셜 네트워킹 서비스Social networking service를 필두로 한 웹2.0의 확산에 따른 소규모, 다량 데이터의 분산화라는 특성을 가지는 데이터 모델의 요구 변화를 전통적인 관계형 데이터 모델에 따른 데이터 관리 방식으로 수용하는 데는 서서히 한계점이 드러나고 있습니다.

NoSQL은 이러한 정보시스템 데이터 모델의 변화를 수용할 수 있는 기술적 대안으로 서서히 응용 분야가 확대되고 있는 데이터 스토어 기술이며, 국내에도 많은 개발자가 관심을 기울이고 있는 상황입니다. 이 책은 관계형 데이터 모델에 익숙한 개발자들이 NoSQL에 대한 기초적인 개념과 대표적인 NoSQL 데이터베이스의 활용법을 빠르게 학습하고 새로운 정보시스템의 개발에서 응용할 수 있는 적절한 출발점을 제공합니다.

이 책의 서두에서 언급하는 바와 같이 NoSQL은 Not Only SQL이라는 말의 약어입니다. 다시 말해서, 기존 관계형 데이터베이스에 대한 대체 기술이라기보다는, 관계형 데이터베이스에 어울리지 않는 특성을 가진 데이터를 효과적으로 관리할 수 있도록 하는 보완 수단으로서의 데이터스토어 기술이라고 할 수 있습니다. 그러한 점에서 책의 전반부에서 기본적인 개념과 관계형 데이터베이스와의 비교를 통해서 개념을 설명하고, 세부적인 NoSQL 기술에 대한 내용으로 진행하고 있는 점은 효과적인 접근 방향이라고 생각합니다.

책의 중반부는 memcached, Tokyo Tyrant, Redis, MongoDB, Cassandra등의 대표적인 NoSQL 기술을 개발자들이 흔히 사용하는 리눅스 운영체제 환경에서 실제로 동작시키고, 루비Ruby 언어를 이용해서 실습할 수 있도록 예제를 제공함으로써 손쉽게 개념을 이해할 수 있도록 가이드하고 있습니다. 다만, 번역서라는 특성을 반영하듯 국내에서 사용자가 많지 않은 루비를 이용해서 예제가 작성되고 있는 점은 다소 어색하게 느낄지도 모릅니다. 하지만, 파이썬만큼이나 손쉽게 학습할 수 있으면서도, 강력한 인터프리터 언어인 루비를 접할 수 있는 기회를 제공한다는 점에서, 이 역시도 긍정적인 평을 할 수 있을 것 같습니다.

추가로 도움이 될 수 있는 주문을 한다면, "관계형 데이터베이스"를 이용하면서 가지게 된 습관을 조금은 허물고, 이 책을 읽는다면 더욱 얻게 되는 것이 많지 않을까 합니다. 물론, 새로운 기술을 익히는 과정에서 기존의 경험은 좋은 비교 수단을 제공한다는 점에서 의미가 있다고 생각합니다. 신기술이 반드시 정답일 수는 없습니다. 하지만, 기존 기술의 바탕에서 정보시스템에 대한 새로운 요구 변화에 동력을 제공하기 위한 중요한 씨앗으로서 신기술의 적용은 항상 고민해야 할 주제인 것 같습니다. NoSQL기술을 통한 차세대 정보시스템 데이터 모델의 수립을 위한 좋은 출발점으로 이 책은 적절한 선택이 될 것이라고 생각합니다.

미국의 개척시대 때 많은 사람들이 황량한 땅 미국으로 모여들었다. 아직 개척되지 않은 황량한 땅에 모여드는 사람들의 수많은 이유가 있었지만, 서부 지역을 중심으로 황금이 발견되었다는 소식으로 많은 사람이 금을 캐고자 고향을 버리고 그곳으로 이동하였던 때가 있었다. 이 당시에 금맥이라도 발견한 사람이 있으면 그 사람은 엄청난 부를 소유한 사람이 될 수 있었다.

인간친화적인 화면의 인터페이스와 디자인을 적용하고 사람이 편리하게 사용할 수 있도록 중앙집중화된 애플리케이션을 제공하는 것을 개척한 애플은 아이폰이라는 금광을 캐어 엄청난 부와 명성을 쌓아가고 있다. 사람이 생각하는 것, 사람이 바라는 것, 사람이 좋아하는 것을 분석하여 그것의 가치를 찾아낸 결과물이라 할 수 있을 것이다.

고대로부터 인간이 문명과 함께 발전하면서 가장 비약적으로 발전을 이룬 것 중에 하나가 바로 기록하는 문화라 할 수 있다. 과거에는 단지 기록하는 것 자체가 엄청난 변화였고 그것이 주는 가치가 엄청난 것이었다고 하면 최근에는 수많은 기록물 속에서 진주보다 값진 의미를 찾아내는 것이 핵심 화두라 할 수 있다.

IT를 활용한 다자간의 커뮤니티 소셜 네트워크 서비스SNS는 이미 상상 이상의 데이터를 쏟아내고 있고 그것을 분석하는 누군가에게는 다양한 기회의 요소로 활용될 수 있는 정보의 보고가 되고 있다. 바로 사람이 바라는 것이 무엇인지를 이제는 요행을 바라는 것이 아닌 데이터 분석을 통해 그것을 찾아 낼 수 있는 기반이 된 것이다. 스마트폰의 데이터, 웹 환경에서 이용되는 패턴들, 각종 프로그램이 실행되면서 쌓이는 로그들, 이동형 디바이스들 속에서 나오는 수많은 데이터들, 염색체, 우주 변화 정보 등 수없이 많은 데이터가 이제는 빠른 분석을 기다리는 듯 누적될 수 있는 환경이 된 것이다. 이것을 효과적으로 효율적으로 분석하는 누군가는 마치 금맥을 발견하는 것처럼 이 사회 속에서 강자로 우뚝 설 수가 있다고 할 수 있다.

그러나, 어마어마한 데이터를 분석하기에는 기존 IT 환경이 데이터를 빠른 시간 안에 저장하고 이것을 분석하는 데 많은 한계점이 있을 수밖에 없는 기술적인 특성을 안고 있다. 그렇기 때문에 세계적인 검색 기업인 Google 등은 빅 데이터Big data를 처리하는 다른 방식 즉, NoSQL 방식의 기술을 적용하여 어마어마한 데이터를 처리하여 빠른 성능으로 검색 서비스 및 많은 서비스를 제공하고 있는 것이다.

이미 우리나라에서도 이러한 기법들이 실용화되어 사용되고 있으며 이는 몇몇 특별한 패턴의 업무 환경뿐만 아니라 일반 기업이나 기관 등에서도 업무 특성에 따라 적용하기 위해 많은 검토를 하고 있는 것이 사실이다.

시대에 적합하고, 적용에 대한 통찰Insight을 구체적으로 발견할 수 있도록 도와줄 수 있는 이 책은 NoSQL 군의 다양한 데이터베이스의 특징, 그리고 적용 방법 등을 구체적으로 제시하고 있다. 이 책이 시중에 나오기 전에 데이터베이스를 전문적으로 다루고 있는 LG CNS DB 관리팀에서는 이 책을 통해 NoSQL에 대해서 사전에 충분히 학습을 전개하였다. 여기에서 형성된 기본적인 이해와 실질적인 적용 사례 등을 바탕으로 빅 데이터 처리의 구체화된 적용을 추진하고 있으므로, 이 책이 주는 NoSQL에 대한 기본적인 이해와 적용에 있어 좋은 책임을 입증하였다고 할 수 있다.

데이터로 인한 변화의 시기에 빅 데이터, NoSQL에 갈급(渴急)한 많은 사람에게 '목마른 사슴에게 시냇물이 나타나는 듯'한 책이 바로 이 책이 되기를 바란다.

2011년 12월
이춘식

시작하며

최근, 『NoSQL』이라는 단어를 많이 보게 됩니다. 이 단어는 대체 어떤 의미일까요? 또, 『NoSQL』을 사용하면 어떤 부분이 좋을까요?

데이터스토어라고 하면 일반적으로는 관계형 데이터베이스를 떠올릴 것입니다. 하지만, 관계형 데이터베이스도 만능은 아니며, 처리에 약점이 있는 부분도 있습니다. 그러한 관계형 데이터베이스의 약점을 채워주기 위해 등장한 것이 이 책에서 다루는 NoSQL 데이터베이스입니다. NoSQL 데이터베이스를 적재적소에 이용함으로써, 관계형 데이터베이스에서는 시간이 걸릴 것을 빠른 시간 안에, 유연하게 처리할 수 있습니다.

책에서 가정하는 이용 분야

NoSQL 데이터베이스는 아래 두 분야에서 주로 이용될 것으로 보입니다.

- 관계형 데이터베이스에서는 다루기 어려운 처리(대용량 데이터의 쓰기 등)를 위해
- 관계형 데이터베이스에서도 가능한 처리를 위한 데이터베이스 선택시 고려 대상 중 하나

NoSQL 데이터베이스는 '관계형 데이터베이스에서는 다루기 어려운 처리(대량 데이터의 쓰기 등)를 대체'하는 식으로 이용할 수 있으나, 이렇게 중요한 용도로 효과적으로 이용하기는 굉장히 어렵습니다. 이 책에서는 간단하게 '관계형 데이터베이스에서도 가능한 처리의 데이터베이스 선택시 고려 대상 중 하나로 이용'하는 경우를 주로 설명하도록 하겠습니다.

대상 독자

이 책은 관계형 데이터베이스를 업무에 1년 이상 사용한 경험이 있는 엔지니어나 프로그래머를 대상으로 하고 있습니다. 데이터 양의 증가에 따라 관계형 데이터베이스의 검색이나 갱신 처리 속도가 급격하게 악화되는 것을 체감했다면, 빠른 성능을 제공하는 NoSQL 데이터베이스 도입의 장점을 알 수 있을 것이라고 생각합니다.

이 책에서는 그런 분을 대상으로 NoSQL 데이터베이스와 NoSQL 데이터베이스를 실제 환경에서 사용할 때의 장점을 소개합니다.

내용에 대해

이 책의 구성에 대해서 간단히 설명하겠습니다.

먼저 1장은 도입부분으로 『NoSQL』은 도대체 무엇인가, 그 기원을 설명합니다. 그후 관계형 데이터베이스가 등장한 역사적 경위와 관계형 데이터베이스의 장•단점, 그리고 '왜 NoSQL 데이터베이스가 필요한가'라는 배경에 대해 소개하고 어떻게 NoSQL 데이터베이스가 등장하게 되었는지를 이야기합니다.

NoSQL 데이터베이스에는 어떤 종류와 특징이 있는지에 대해서도 이 장에서 간단하게 다룹니다. 또한 '관계형 데이터베이스와 NoSQL 데이터베이스를 어떤 식으로 구분하여 사용하면 좋을까'라는 NoSQL 데이터베이스 도입시의 준비에 대해서도 이 장에서 다루므로 참고하기 바랍니다.

2장에서는 NoSQL 데이터베이스로 memcached, Tokyo Tyrant, Redis, MongoDB, Cassandra, HBase에 주목하여 각 NoSQL 데이터베이스를 사용하는 배경, 특징과 사용 케이스, 실제 이용할 때 어떻게 동작하는지를 소개합니다. 이 장을 읽으면, 각 NoSQL 데이터베이스의 기초적인 사용 방법은 알 수 있을 것입니다.

3장에서는 2장에서 다룬 NoSQL 데이터베이스의 이용 분야와 실제로 구현된 코드에 대해 보다 구체적으로 소개합니다. memcached, Tokyo Tyrant, Redis, MongoDB라는 NoSQL 데이터베이스가 구체적이며 실제 업무에 있을 법한 사례를 해결할 수 있다는 것을 이해하여 데이터베이스 선택시 고려 대상의 하나가 될 수 있다면 좋을 것입니다.

덧붙여서 이 장에서 다루는 사례는 모두 관계형 데이터베이스에서도 가능하지만, NoSQL 데이터베이스를 사용했을 때 응답시간이 빨라진다든지, 운영 부담이 줄어드는 잇점이 있습니다.

4장에서는 마찬가지로 4가지 NoSQL 데이터베이스에 대해 퍼포먼스 비교를 하고 있습니다. 기본적인 CRUD 처리에 관련된 퍼포먼스뿐 아니라, Tokyo Tyrant의 addint 메소드와 incr 메소드의 퍼포먼스 비교, Redis의 리스트형에 대한 추가와 삭제 성능 비교, MySQL의 JOIN과 MongoDB의 embed의 퍼포먼스 비교와 같이 보다 구체적인 경우의 성능을 비교하고 있습니다.

마지막으로 5장에서는 NoSQL 데이터베이스를 이용할 때의 우려점과 이 우려를 해결할 수 있는 HandlerSocket이라는 모듈을 소개합니다. 이것은 NoSQL 데이터베이스와는 조금 다릅니다만, 굉장히 재미있는 기능으로, '이렇게 사용할 수도 있구나'라고 알아주시면 기쁠 것입니다.

2011년
Tatsuya Sasaki

실습 안내

1) 아래와 같은 박스 안에 표시된 것은 모두 명령어입니다. #으로 표시된 줄은 주석입니다.

```
rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/epel-release-5-4.noarch.rpm
yum repolist
```

2) $가 들어있는 박스는 $ 다음이 명령어를 의미합니다. 그 외의 내용은 $ 다음 명령어 실행시 나오는 결과값입니다.

```
$ ps aux | grep memcached
memcached -d -p 11211 -u memcached -m 64 -c 1024 -P /var/run/memcached/memcached.pid
```

3) 아래와 같은 경우, $ 다음의 명령어를 사용하여 실행한 telnet 혹은 테스트 쉘에서 볼드체로 표시된 명령어를 입력하면 일반 글씨체의 실행 결과가 나타납니다.

```
# localhost의 11211번 포트에 접속
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

set counter 0 0 1
1
STORED
# 더하기 연산 (빼기는 decr 명령어)
incr counter 1
```

명령어 직접 입력

4) 아래와 같이 require나 $KCODE, class로 시작하는 루비 예제는 'rb' 확장자로 끝나는 파일로 저장해서 아래 형태로 실행하시면 됩니다. 루비 예제에서는 p 등으로 출력하는 프로그램 코드 옆에 '# 출력 결과' 형태로 출력 결과가 표시됩니다.

```
vi test.rb

$KCODE = 'u'

require "rubygems"

# 데이터 읽어오기
p cache['key1']      # 123
p cache['key2']      # "abcde"
p cache['key3']      # ["hoge", "fuga"]
p cache['key4']      # {:foo=>1, :bar=>"a"}

ruby test.rb
```

3장부터 사용되는 ruby on rails는 관련 도서를 참고하기 바랍니다.

5) 참고로 회색 박스는 아래와 같이 실행 결과값이나 로그를 표시할 때도 사용됩니다.

```
# 대상 memcached 정보
INFO -- : memcache-client 1.8.5 ["localhost:1978", "localhost:1979", "localhost:1980"]
DEBUG -- : Servers now: [<MemCache::Server: localhost:1978 [1] (NOT CONNECTED)>,
<MemCache::Server: localhost:1979 [1] (NOT CONNECTED)>, <MemCache::Server:
localhost:1980 [1] (NOT CONNECTED)>]

# 데이터 입력
DEBUG -- : set key1 to <MemCache::Server: localhost:1980 [1] (CONNECTED)>: 5
DEBUG -- : set key2 to <MemCache::Server: localhost:1979 [1] (CONNECTED)>: 9
DEBUG -- : set key3 to <MemCache::Server: localhost:1979 [1] (CONNECTED)>: 16
DEBUG -- : set key4 to <MemCache::Server: localhost:1978 [1] (CONNECTED)>: 19

# 데이터 출력
DEBUG -- : get key1 from <MemCache::Server: localhost:1980 [1] (CONNECTED)> 123
DEBUG -- : get key2 from <MemCache::Server: localhost:1979 [1] (CONNECTED)> "abcde"
DEBUG -- : get key3 from <MemCache::Server: localhost:1979 [1] (CONNECTED)> ["hoge",
"fuga"]
DEBUG -- : get key4 from <MemCache::Server: localhost:1978 [1] (CONNECTED)> {:bar=>"a",
:foo=>1}
```

```
"localhost:6379: key_18, key_9, key_20"
"localhost:6380: key_10, key_1, key_11, key_2, key_3, key_4, key_5, key_6, key_7,
key_12, key_13, key_14, key_15, key_16, key_17"
"localhost:6381: key_8, key_19"
```

6) 예제 소스 다운로드 안내

2장과 3장의 ruby 예제는 별도의 예제 소스 파일로 다운로드할 수 있습니다.

예제 소스 다운로드 : www.roadbook.co.kr/56

목차

편역자의 글 3
베타 리뷰어 글 4
감수자의 글 6
저자 서문 8
실습안내 10

Chapter 1. NoSQL 데이터베이스의 기초지식

1-1 관계형 데이터베이스와 NoSQL 데이터베이스 18
1-1-1 NoSQL이란? 18
1-1-2 관계형 데이터베이스의 간략한 역사 19
1-1-3 데이터베이스의 분류 19
1-1-4 관계형 데이터베이스의 장점 22
1-1-5 관계형 데이터베이스의 약점 23
1-1-6 NoSQL 데이터베이스 28

1-2 NoSQL 데이터베이스에는 어떤 것이 있는가? 32
1-2-1 key-value형 스토어 33
1-2-2 문서형 데이터베이스 35
1-2-3 컬럼형 데이터베이스 35

1-3 NoSQL 데이터베이스의 도입은 어떻게 할 것인가? 37
1-3-1 어디까지나 고려 대상의 하나 37
1-3-2 어디까지 믿을 수 있는가? 39

Chapter 2. NoSQL 데이터베이스의 종류와 특징

2-1 memcached(휘발성 key-value형 스토어) 42
2-1-1 memcached란? 42
2-1-2 왜 memcached를 사용하는가? 42
2-1-3 특징과 사용 케이스 44
2-1-4 도입 절차 51
2-1-5 동작 확인 53
2-1-6 각 언어용 라이브러리 62
2-1-7 관련 툴 63

2-2 Tokyo Tyrant (영속성 key-value형 스토어) 71
2-2-1 Tokyo Tyrant란? 71
2-2-2 왜 Tokyo Tyrant을 사용하는가? 71
2-2-3 특징과 사용 케이스 72
2-2-4 도입 절차 77
2-2-5 동작 확인 79
2-2-6 각 언어용 라이브러리 88
2-2-7 관련 툴 88

2-3 Redis(휘발성/영속성 key-value형 스토어) 92
2-3-1 Redis란? 92
2-3-2 왜 Redis를 사용하는가? 92
2-3-3 특징과 사용 케이스 101
2-3-4 도입 절차 106
2-3-5 동작 확인 107
2-3-6 각 언어용 라이브러리 117

2-4 MongoDB(문서형 데이터베이스) 118
2-4-1 MongoDB란? 118
2-4-2 왜 MongoDB를 사용하는가? 118
2-4-3 특징과 사용 케이스 120
2-4-4 도입 절차 125
2-4-5 동작 확인 126
2-4-6 각 언어용 라이브러리 139
2-4-7 관련 툴 140

2-5 Cassandra(key-value형 스토어) 142
2-5-1 Cassandra란? 142
2-5-2 왜 Cassandra를 사용하는가? 142
2-5-3 특징과 사용 케이스 143
2-5-4 도입 절차 149
2-5-5 동작 확인 151
2-5-6 각 언어용 라이브러리 155
2-6 HBase(컬럼형 스토어) 157
2-6-1 HBase란 157
2-6-2 왜 HBase를 사용하는가? 157
2-6-3 특징과 사용 케이스 158
2-6-4 도입 절차 162
2-6-5 동작 확인 164
2-6-6 각 언어용 라이브러리 171

Chapter 3. NoSQL 데이터베이스를 사용해 본다 173

3-1 memcached의 구체적인 이용 예 174
3-1-1 샘플① 관계형 데이터베이스의 Cache 174
3-1-2 샘플② 음원 랭킹 사이트 184
3-1-3 샘플③ 외부 API의 cache 192
3-2 Tokyo Tyrant의 구체적인 이용 예 193
3-2-1 샘플① 페이지 별 Access 수 표시 193
3-3 Redis의 구체적인 이용 예 205
3-3-1 샘플① 타임라인 형식의 웹 어플리케이션 205
3-3-2 샘플② 검색 이력 기능 222
3-4 MongoDB의 구체적인 이용 예 230
3-4-1 샘플① 앙케이트 응답 데이터 보존 230
3-4-2 샘플② 분석 데이터 스토리지 247
3-5 Cassandra의 구체적인 이용 예 249
3-5-1 샘플① 사용자 검색 정보 스토리지 249
3-6 HBase 구체적인 이용 예 251
3-6-1 샘플① 대형 SNS 서비스의 주 스토리지 251

Chapter 4. 퍼포먼스 검증

4-1 기본적인 입력과 검색 성능 254
4-1-1 테스트 케이스 254
4-1-2 준비 258
4-1-3 입력 성능 259
4-1-4 검색 성능 260
4-2 구체적인 경우의 성능 비교 262
4-2-1 Tokyo Tyrant의 addint 메소드와 incr 메소드 262
4-2-2 Redis에서 리스트형 데이터의 입력과 삭제 264
4-2-3 MySQL의 JOIN과 MongoDB의 embed 266

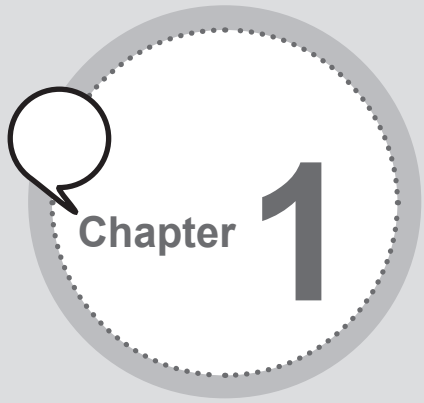
Chapter 5. NoSQL인 관계형 데이터베이스

5-1 NoSQL 데이터베이스에 대해 272
5-1-1 각각의 특징 272
5-1-2 운영시의 비용 및 적은 노하우 274
5-1-3 MySQL을 NoSQL화 하는 방법 275

5-2 HandlerSocket을 테스트해 본다 276
5-2-1 특징 276
5-2-2 HandlerSocket을 MySQL에 설치 277
5-2-3 동작 확인 281
5-2-4 성능은 어떨까? 288

부록 기본 환경 구축 가이드(CentOS 5.4 & Ruby 1.8.7) 292

찾아보기



NoSQL 데이터베이스의 기초지식

1장에서는 관계형 데이터베이스를 설명하고, 왜 NoSQL 데이터베이스가 등장하게 되었는지를 관계형 데이터베이스의 장점과 약점을 통해 알아봅니다.

그리고 NoSQL 데이터베이스에는 어떤 종류가 있는지, 관계형 데이터베이스와 NoSQL 데이터베이스를 어떻게 구분해서 사용하면 좋을지를 다룹니다.

1-1 관계형 데이터베이스와 NoSQL 데이터베이스

1-1-1 NoSQL이란?

『NoSQL』을 알고 계십니까? 요즘 주목을 받고 있는 용어입니다. 『NoSQL』이라는 단어를 보고, 『SQL』은 더 이상 필요없다 (No! SQL)과 같은 과격한 의미라고 생각할지도 모르지만 실제로는 『Not Only SQL』의 약어입니다. 즉, **『관계형 데이터베이스[1]가 맞으면 관계형 데이터베이스를, 적합하지 않으면 무리하게 관계형 데이터베이스에 얽매이지 말고 용도에 맞는 데이터 스토어[2]를 사용하자』**는 것입니다.

이런 생각에서 관계형 데이터베이스가 취약한 부분을 보충하기 위한 여러 NoSQL 데이터베이스가 등장하였습니다.

이 책은 NoSQL 데이터베이스에 대해 소개하는 책입니다만, 그 전에 관계형 데이터베이스에 대한 이해가 필요합니다. 먼저 관계형 데이터베이스의 등장과 역사, 분류, 특징에 대해 알아보겠습니다.

1 Oracle, Microsoft SQL Server, DB2, PostgreSQL 등이 있으며, 이 책에서는 MySQL을 주로 의미합니다.

2 데이터 스토어란 일정 용도를 위한 데이터가 모인 데이터의 저장소를 의미합니다. 데이터 스토어는 데이터베이스뿐만 아니라 데이터를 모아 둔 일반 파일도 포함합니다. _ 역자주

1-1-2 관계형 데이터베이스의 간략한 역사

1969년 에드거 F. 코드가 획기적인 논문을 발표하여 관계형 데이터베이스를 제창하였습니다. 그러나 이 논문은 사내 저널(『IBM Research Report』)에 실려, 일반인의 주목을 끌지는 못했다고 합니다. 하지만 다음해인 70년, 이번에는 권위있는 학술지 『Communications of ACM』에 "A Relational Model of Data for Large Shared Data Banks" (대규모 공유 데이터뱅크의 데이터 관계 모델)라는 제목으로 다시 게재되어 주목을 받았습니다.

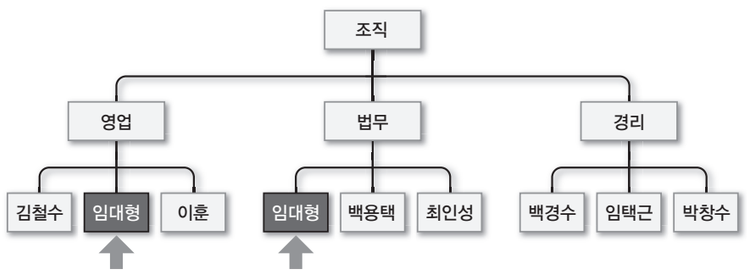
코드가 제창한 이 관계형 모델이라는 개념이 현재의 관계형 데이터베이스의 기초입니다. 처음에는 H/W가 지원해주지 못해 성능이 좋지 않았고 실용화도 되지 못했습니다. 이 후 H/W의 진화와 함께, 조작의 용이성 등 여러 운영 상의 장점을 바탕으로 관계형 데이터베이스는 일반적으로 널리 사용하게 되었습니다.

1-1-3 데이터베이스의 분류

데이터베이스는 데이터 모델(데이터를 어떤 방법으로 표현할 지를 정한 것)에 따라 주로 『계층형』, 『네트워크형』, 『관계형』의 세 가지로 분류됩니다.

계층형 데이터베이스

초기의 데이터베이스는 계층형 데이터베이스라 하여, 데이터의 관계성을 단순한 트리구조로 정의하였습니다. 프로그램에서는 트리 구조의 정의를 이용하여 데이터에 접근합니다. 이 구조에서는 부모(상위 레코드)는 복수의 자식(하위 레코드)을 가지지만 자식은 하나의 부모만 갖게 됩니다. 그러므로 굉장히 단순한 구조이긴 하지만, 복잡한 데이터를 다룰 때는 데이터가 중복(하나의 데이터가 데이터베이스에 복수 존재)되게 되어, 데이터의 중복 현상이 문제가 되었습니다.



임대형씨가 영업과 법무에 모두 소속되어 있는 경우 곤란하다.

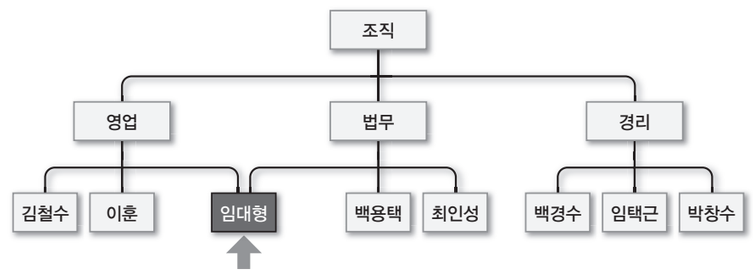
그림 1.1 계층형 데이터베이스 예

계층형 데이터베이스에는 데이터 구성이 계층 구조로 표현되어 있기 때문에, 이것을 이용하여 효율적인 검색이 가능하다는 장점이 있지만, 반대로 보면 효율적으로 검색하기 위해서는 데이터 구조를 이해하고 있어야만 합니다. 당연히, 계층 구조를 변경시킨 경우에는, 이 데이터베이스를 이용하고 있는 프로그램을 변경에 맞춰 수정해야 합니다.

네트워크형 데이터베이스

먼저 이야기한 것처럼, 계층형 데이터베이스에서는 데이터의 중복이 문제가 됩니다. 여기서 이 문제를 해결하기 위해 등장한 것이 네트워크형 데이터베이스입니다. 기본적으로 계층형 데이터와 비슷한 구조를 하고 있으나, 각 데이터가 그물코처럼 연결되어 있어 네트워크형이라고 불립니다.

계층형 데이터베이스에는 데이터의 관계성을 부자 관계로만 표현 가능했던 것에 반해, 네트워크형의 데이터베이스에는 자식도 여러 부모를 가질 수 있어서, 데이터의 중복 현상을 없앨 수 있게 되었습니다.



임대형씨가 여러 부모(영업과 법무)를 가집니다.

그림 1.2 네트워크형 데이터베이스 예

하지만 네트워크형 데이터베이스에서는 데이터 사이의 관계가 복잡해 데이터의 구조를 쉽게 바꿀 수 없다는 문제가 있습니다. 또한, 계층형과 마찬가지로 데이터 구조를 이해하고 있지 않으면 데이터에 접근이 불가능하여, 데이터 구조에 강한 의존성을 가지고 있습니다.

여기서 잠깐

이제 계층형, 네트워크형 데이터베이스는 쓰이지 않나요?

1980년대 초까지 주로 사용되었습니다. 계층형 데이터베이스로는 IBM IMS가 있으며, 네트워크형 데이터베이스는 IDL, TOTAL 등이 있으나 현재 시장에서는 관계형 데이터베이스가 주로 사용되고 있습니다.

관계형 데이터베이스

이후에 등장한 것이 코드가 제창한 관계형 모델을 바탕으로 하는 관계형 데이터베이스입니다. 관계형 데이터베이스에서는 모든 데이터는 행과 열의 이차원 표 형식으로, 사람이 직감적으로 이해하기 쉬운 형태로 표현됩니다. 네트워크형 데이터베이스에서는 데이터 구조의 변경이 어렵다는 문제가 있었습니다만, 관계형 데이터베이스에서는 값에 복수의 데이터를 연관시키는 것(여러 컬럼을 가짐)이 가능하기 때문에 각 데이터는 독립되어 있고, 데이터 구조를 비교적 간단하게 변경할 수 있습니다.

2-1 memcached (휘발성 key-value형 스토어)

2-1-1 memcached란?

memcached는 Danga Interactive에서 개발한 오픈소스 소프트웨어입니다. NoSQL 데이터베이스 중 휘발성 key-value형으로 분류됩니다.

공식 사이트 http://memcached.org/

2-1-2 왜 memcached를 사용하는가?

응답 속도가 빠름

많은 웹 어플리케이션에서는 관계형 데이터베이스에 데이터를 두고 필요한 데이터를 읽어 사용자의 브라우저에 표시해 줍니다. 관계형 데이터베이스에 보관된 데이터 양이 적을 때는 금방 결과값이 나와서 아무 문제도 없습니다만, 관계형 데이터베이스의 데이터가 늘어난다든지 복잡한 데이터 연산을 한 값을 반환해야 할 때는 응답 속도(사용자가 웹 어플리케이션에 접속하여 그 결과가 나올 때까지의 속도)가 느려지게 됩니다. 이 때 사용자가 응답을 기다려 줄까요?

Aberdeen Group의 2008년 조사에 따르면 응답 속도가 1초 느려지면 사이트의 접속수가 11%, 매상이나 신청이 7%, 고객 만족도가 16% 저하된다고 합니다.[1] 구체적인 사례로 Google의 "응답 속도가 0.5초 느려지면 검색수가 20% 적어진다", Amazon의 "응답 속도가 0.1초 느려지면 매출이 1% 감소한다" 등이 알려져 있습니다. 응답 속도는 사용자의 행동과 매상에 그대로 연결되는 굉장히 중요한 지표입니다.

캐시로 이용

그렇다면 어떻게 해서 응답 속도를 높일까요? 간단한 처리라면 관계형 데이터베이스에 인덱스를 사용하여 속도를 빠르게 하는 방법이 있습니다. memcached보다는 느리지만, 제대로 인덱스를 이용하면 관계형 데이터베이스도 충분히 빨라집니다.

하지만 여러 테이블에 흩어져 있는 데이터를 계산해야 하는 경우에는 어떻게 해야 할까요? 관계형 데이터베이스를 사용할 경우, 각 테이블에서 데이터를 읽어와서 마지막에 합칩니다. JOIN을 사용하는 복잡한 처리가 매번 필요해질 수도 있습니다. 사전에 배치를 돌려서 데이터를 작성해둘 수도 있습니다만, 이럴 경우 관리 테이블이 늘어 운영에 드는 수고가 늘어나게 됩니다.

"데이터 자체는 관계형 데이터베이스에서 수십초~수분 내에 준비할 수 있으나, 리얼타임 서비스로는 느리다." 이럴 때 활약하는 것이 memcached입니다. 관계형 데이터베이스에서 데이터를 읽어와서 memcached에 데이터를 두어 캐시로 사용하면, 대량 데이터에 대한 요청이 있어도 빠르게 응답할 수 있습니다. 동일한 요청(2회 이후)에는 memcached에 데이터가 있는 이상, 관계형 데이터베이스에 갔다오지 않고 바로 memcached에서 처리할 수 있습니다. 단, memcached에서는 데이터가 사라질 가능성이 있으므로 memcached가 중지되어 데이터가 유실되더라도 곧 다시 만들 수 있는 경우에 이용하는 것이 좋을 것입니다.

그렇다면 캐시로 사용할 경우 원본 데이터 값이 바뀌었을 때 memcached 상의 데이터는 어떻게 해야 하는지 궁금하신 분들도 계실 것입니다. 그럴 경우에는 소스 코드에 별도

1 출처 : http://www.gomez.com/wp-content/downloads/aberdeen-gomez-best-in-class.pdf

로 로직을 구현하거나 expires를 사용하는 등의 대처법을 생각할 수 있습니다. 보다 자세한 내용은 3장에서 설명하겠습니다.

2-1-3 특징과 사용 케이스

특징

memcached는 Hash(연상 배열) 형식으로 데이터를 관리할 수 있는 key-value형 스토어입니다. 데이터는 모두 메모리에 보관됩니다.

memcached와의 통신은 간단한 텍스트 방식을 이용해서 이뤄집니다. 데이터의 조작도 key에 대응하는 value를 보존하는 것만으로 단순합니다. telnet으로 memcached에 접속하여 데이터를 입출력하는 것이 가능합니다. 하지만, 텍스트 프로토콜이라서 구조체 데이터는 다루지 못하고 문자열 데이터만 value로 가질 수 있습니다.

물론, 언어별로 준비된 라이브러리를 사용하여 memcached를 이용할 경우, value에 문자열이나 숫자뿐만 아니라 배열이나 Hash와 같은 구조체 데이터를 입출력할 수 있습니다. 하지만, 앞서 말한 것처럼 텍스트 프로토콜로 통신하고 있기 때문에 시리얼라이즈하여 바이트열로 변환해야 합니다. 보통 이런 처리는 라이브러리 내에서 이뤄지기 때문에 이용시 고려할 필요는 없습니다만, 입력시 시리얼라이즈되고 출력시 디시리얼라이즈되어 복원됩니다. 예를 들어 Ruby의 경우에는 입력시 Marshal.dump되고 출력시 Marshal.load[2]됩니다.

시리얼라이즈의 언어 의존

시리얼라이즈가 각 언어에 의존하고 있기 때문에, 한 언어로 시리얼라이즈된 후 다른 언어로 디시리얼라이즈는 불가능합니다(제대로 복원되지 않기 때문에 데이터가 깨집니다). 만약 그런 처리가 필요한 경우에는 JSON이나 MessagePack 등을 이용하여 언어에 의존하지 않는 포맷으로 명시적으로 시리얼라이즈/디시리얼라이즈 해줘야 합니다.

2 Ruby에서 데이터를 바이트로 변환(Marshal.dump)하고 변환된 바이트열을 데이터로 되돌리는(Marshal.load) 함수

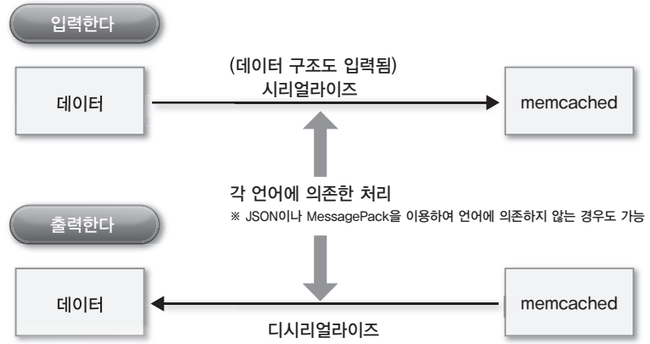


그림 2.1 memcached의 시리얼라이즈/디시리얼라이즈

장점

memcached의 장점은 역시 압도적인 속도입니다. 데이터가 모두 메모리 위에 있기 때문에 디스크 I/O를 발생시키지 않아서 관계형 데이터베이스에 비해 상당히 빠르게 작동합니다. 메모리 데이터에 대한 접속이 디스크 데이터에 대한 접속보다 10만배 ~ 100만배 빠르기 때문입니다.

간과하기 쉬운 부분이지만 가볍게 사용할 수 있는 것 또한 큰 장점입니다. key와 value라는 Hash 형식으로 데이터를 다루기 때문에, Hash를 사용해 본 경험이 있다면 수월하게 이용할 수 있어, 사용 방법 때문에 막히는 일은 없을 것입니다. memcached를 정지시키면 모든 데이터가 날아가기 때문에 무언가 이상한 상황에 직면한다 해도 memcached를 재기동하면 시작 상황으로 돌아갈 수 있습니다. 아무리 우수한 툴이라도 사용법이 어렵거나 작업에 수고가 많이 들어 다른 툴을 택한 경험은 누구나 가지고 있을 것입니다.

다수의 웹 어플리케이션에서 실제로 사용하고 있어서 노하우가 많이 공개되어 있습니다. 따라서 장애가 적은 것도 좋은 점입니다.

데이터 양이 늘어나, memcached의 메모리에 데이터를 모두 올리지 못하고 memcached를 여러 대로 분산하여 구성해야 할 수도 있습니다. memcached는 여러 대로 구성하는 것도 굉장히 간단합니다. memcached를 여러 대로 구성할 때 데이터의 분배는 Consistent Hashing이라는 방법을 이용합니다. 이 알고리즘은 memcached의 클라이언트 라이브러리에 포함되어 있습니다.

Consistent Hashing

여러 대의 서버를 이용하는 것뿐이라면, 데이터 key에 대한 Hash 값을 계산하여 이것을 서버 수로 나눈 나머지를 가지고 어느 서버에 어떤 데이터를 둘 지 간단히 결정할 수 있습니다. 예를 들어 memcached 서버를 4대 이용하는 경우, key의 Hash 값이 516113735라면, 516113735 % 4의 나머지는 3이 됩니다. 나머지는 0, 1, 2, 3 중 하나가(각각 4대의 서버에 대응) 되므로 key에 따라 4대의 서버 중 어느 서버에서 다룰 지가 자동으로 결정됩니다.

하지만, 위 방법을 사용한다면 서버를 추가해서 서버 수가 달라졌을 때 문제가 생깁니다. 서버가 한 대 추가되어 다섯 대가 된다면 어떻게 될까요? 위와 같은 key에서, 516113735 % 5의 나머지는 0이 되어, 데이터가 저장되는 서버는 아까와는 다른 서버가 됩니다(3에서 0으로). 이런 식으로 데이터를 다루는 서버가 서버가 추가될 때마다 바뀌기 때문에 이 방법은 간단하고 이해하기 쉽지만 적합하지 않습니다. 데이터가 저장되는 서버가 달라지면 데이터를 읽으려고 해도 읽을 수 없게 됩니다(이것을 cache miss라고 합니다). 여기서 서버 대수의 증감시 cache miss 문제를 해결하기 위해 나온 것이 Consistent Hashing이라는 데이터 분배 방식입니다.

이 방식에서는 우선 각 서버에 대해 Hash 값을 계산하여 그 값을 원주 위에 배치합니다. 그리고 각 데이터에 대해서도 key의 Hash 값을 계산하여 key의 Hash 값에서 좌측으로 가장 가까운 Hash 값을 가지고 있는 서버가 그 데이터를 담당합니다(입력 출력이 이 서버에서 이루어짐). 다음 그림을 보면 이해하기 쉬울 것입니다.

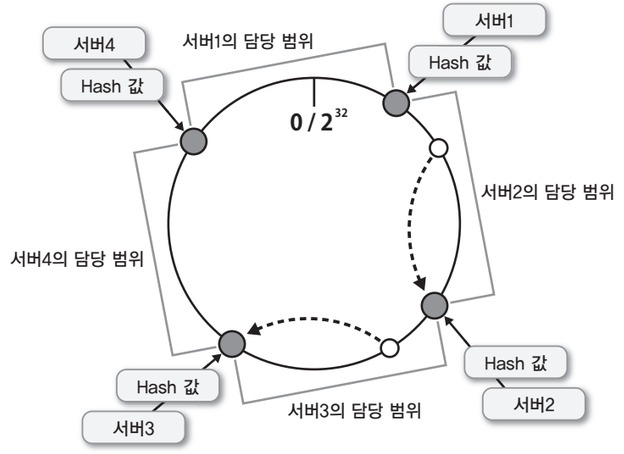


그림 2.2 Consistent Hashing 방식

이런 방식이라면, 서버 증감시 영향이 줄어듭니다.

구체적으로 생각해 봅시다. 예를 들어 앞의 예에 서버5를 추가해서 서버3과 4 사이의 Hash 값으로 해 봅시다. 이 경우, 서버1, 서버2, 서버3의 담당 범위는 바뀌지 않습니다. 각 서버가 담당하는 데이터는 지금까지와 같은 서버에서 계속 처리됩니다. 유일하게 영향을 받는 것은 서버4의 담당 범위뿐입니다.

서버5를 추가하여 서버3과 서버5의 사이가 지금까지는 서버4가 담당했으나, 서버5의 담당 범위가 됩니다. 이 범위의 데이터는 서버5 추가 전에는 서버4에서 처리했으나 서버5를 추가하였으니 요청을 서버5로 하면 됩니다. 그 결과 이 범위의 데이터에 대해서는 일시적으로 cache miss가 다수 발생할 것입니다. 하지만, 서버5와 서버4의 사이는 여전히 서버4의 담당 범위가 됩니다. 따라서 비교적 일부에만 영향을 주므로 cache miss가 많이 줄어들게 됩니다.

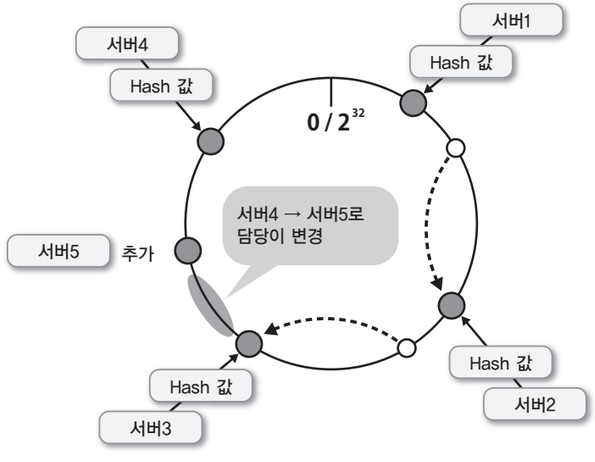


그림 2.3 서버 추가시 동작

약점

memcached의 약점은 무엇이 있을까요? 역시 신경이 쓰이는 건 『데이터의 휘발성(데이터가 날아갈 가능성이 있음)』입니다.

memcached에서는 메모리에 데이터를 두기 때문에 memcached가 갑자기 중지되었을 때 모든 데이터가 사라져 버립니다. 그러므로 사라지면 안 되는 데이터 용으로 사용하는 것은 위험이 큽니다. 어디까지나 원본 데이터는 별도로 두고 원본 데이터의 복사본이나, 원본 데이터에서 계산한 결과를 memcached에 두고 사용하는 용도에 좋습니다.

또한 데이터 출력을 위해 key를 지정해야 한다는 제약이 있습니다. 예를 들어, “data_1”이라는 key의 데이터를 취득할 경우, “값을 포함한다”는 LIKE 검색은 불가능합니다. “data_1”과 완벽하게 일치되도록 지정해야 합니다. SQL로 표현해 보면 아래와 같이 간단한 검색만 가능하기 때문입니다.

```
SELECT value FROM table WHERE key = XXX;
```


그러므로 관계형 데이터베이스에서는 간단한 처리가 memcached에서는 불가능할 때가 있습니다. 이러한 경우에는 프로그램 코드에서 처리해 줘야 합니다.

예를 들어 일시를 key로 하는 데이터를 가지고 있는 경우, 관계형 데이터베이스에서는 BETWEEN ‘2011-02-01’ AND ‘2011-02-05’라는 조건을 지정해서 가져올 수 있으나, memcached에서는 같은 값을 출력하려고 할 때, key로 ‘2011-02-01’, ‘2011-02-02’, ‘2011-02-03’, ‘2011-02-04’, ‘2011-02-05’를 각각 지정하여 데이터를 가져와야 합니다.

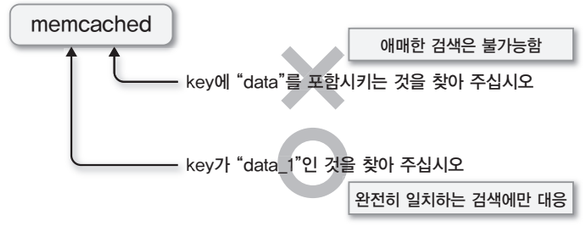


그림 2.4 memcached에서 데이터를 검색할 때 주의할 점

memcached 중지 대비책

접속이 많은 경우, memcached가 중지되었을 때 부하 대책이 필요할 수 있습니다. memcached가 중지되어 데이터가 사라졌을 때, 그때까지 관계형 데이터베이스를 거치지 않고 memcached에서 처리했던 대량의 요청이 한번에 관계형 데이터베이스에 몰리게 되어, 관계형 데이터베이스에 큰 부담를 주게 됩니다. 이 결과 응답이 느려지고, 최악의 경우 서버가 다운될 수도 있습니다.

뒤에 얘기할 repcached를 사용해서 가용성을 높인다든지, memcached를 여러 대 두어 각 memcached에서 다루는 데이터를 줄이는 (1대가 정지되어도 영향이 적어지도록) 등의 조치를 하는 것이 좋을 것입니다.

도입 사례

도입 사례를 몇 가지 소개하겠습니다. 공식 사이트에 소개된 사례 중 극히 일부로, 실제로는 이 이외에도 많은 웹 어플리케이션에서 이용하고 있습니다. 대량의 접속을 처리하는 웹 어플리케이션에서도 문제없이 사용하고 있습니다.

- Wikipedia
- Twitter
- Mixi
- Flickr
- Youtube

응답이 느린 처리를 개선하기 위해 최근에는 많은 웹 어플리케이션에서 도입이 쉬운 memcached를 사용하고 있습니다.

사용 케이스

memcached는 주로, 관계형 데이터베이스에서 가져온 데이터의 캐시로 사용됩니다.[3]

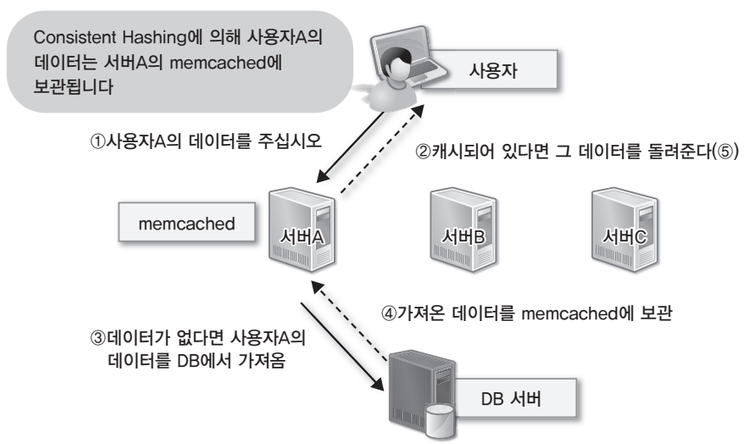


그림 2.5 memcached의 일반적인 이용 방법

3 사용자의 세션 정보를 취급하는 경우도 있습니다만, 데이터가 사라진 경우 로그인이 해제되는 등 영향을 줄 수 있으므로 주의해야 합니다.

구체적으로는 아래와 같은 경우에 도움이 됩니다.

- 시간이 오래 걸리는 SQL을 요청하는 페이지가 있어 응답이 느림
- 빈번하게 요청되는 페이지가 있어 부하가 걸림

물론 이런 경우 요구되는 데이터를 사전에 배치 등을 통해 만들어 두는 것으로 어느 정도 대응할 수 있을 것입니다. 하지만 배치로 처리한다면 그만큼 테이블이 증가하여 운영이나 관리에 부담이 발생합니다. 데이터를 계속해서 보관해야 하는 경우에는 배치로 처리하여 관계형 데이터베이스에서 관리하는 것이 좋을 것입니다. 하지만 일시적으로 이용하고 이후 지워진다고 해도 수 분 안에 만들 수 있는 데이터라면 memcached로 충분합니다.

3장에서는 보다 구체적으로 랭킹 데이터나 외부 API와의 연계에 이용되는 예를 소개하겠습니다.

2-1-4 도입 절차

설치

설치도 간단합니다. CentOS 환경에서는 yum을 사용하여 한번에 설치할 수 있습니다. 하지만 CentOS의 표준 repository에는 memcached가 없으므로[4] 설치 전에 yum의 repository에 추가해야 합니다. 이번에는 EPEL[5]을 이용하겠습니다.

▶ EPEL 설치

```
rpm -Uvh http://download.fedora.redhat.com/pub/epel/5/i386/epel-release-5-4.
                                                              noarch.rpm
yum repolist
```

4 참고 URL: http://www.atmarkit.co.jp/flinux/rensai/linuxtips/794uofrepo.html

5 Extra Packages for Enterprise Linux, http://fedoraproject.org/wiki/History_and_Philosophy_of_EPEL

이것으로 EPEL이라는 repository가 사용가능해졌습니다. 이제 memcached를 아래 명령어로 설치합니다.

```
yum install memcached
```

책에서 이용한 버전은 1.4.5입니다. 물론 소스를 다운로드하여 컴파일해서 설치할 수도 있습니다. 원하는 방식으로 설치해 주십시오.

기동

memcached를 기동합니다.

```
sudo /etc/init.d/memcached start
```

프로세스를 확인해 봅시다. memcached로 시작하는 프로세스가 기동되어 있을 것입니다.

```
$ ps aux | grep memcached
memcached -d -p 11211 -u memcached -m 64 -c 1024 -P /var/run/memcached/memcached.pid
```

각 옵션의 의미는 아래와 같습니다.

표 2.1 memcached 기동 옵션

옵션	설명	디폴트
-d	데몬화시켜 백그라운드에서 기동	-
-p	사용할 포트 번호	11211
-u	사용자명(root로 실행시에만)	nobody
-c	최대 접속수	1024
-m	사용할 수 있는 최대 메모리(단위는 MB)	64

가지고 있는 데이터의 사이즈가 -m 옵션으로 지정한 값보다 커지면 LRU(Least Recently Used)인 데이터를 지웁니다. 가장 적게 사용되는 데이터부터 삭제하므로, 캐시로 이용되는 memcached의 용도에 맞는 방식입니다.

사용하는 메모리 양을 늘리거나, 사용 포트 번호를 변경하기 위해서 /etc/init.d/memcached의 CACHESIZE나 PORT를 변경하면 됩니다. 아래와 같이 직접 지정해서 기동하는 방법도 있습니다.

```
memcached -d -p 11212 -u nobody -c 1024 -m 128
```

2-1-5 동작 확인

telnet으로 확인한다

memcached는 디폴트 설정에서는 11211번 포트를 사용합니다. 사용한다는 것은 해당 포트를 열어서, 그 포트로 들어오는 접속을 기다리고 있다는 의미입니다. 그러므로 telnet을 이용해서 memcached가 사용하는 포트에 접속하여 이용할 수 있습니다. 시험해 봅시다.

```
# locahost의 11211번 포트에 접속
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

# 데이터 입력
# set <key> <flag> <expires> <byte> enter
# <value>
set foo 0 10 3
bar
STORED

# 데이터 출력
```

```
get foo
VALUE foo 0 3
bar
END
```

표 2.2 데이터 입력시 인수

이름	설명	주의점
flag	데이터를 압축할지 여부. 0 : 압축하지 않음, 1 : 압축	–
expires	데이터를 언제까지 보존할지를 UNIX 타임스탬프/현재부터 시간(초)으로 지정	현재부터의 시간(초)은 30일(60 x 60 x 24 x 30 = 2,592,000)을 넘을 수 없음. 0을 지정하면 유효기한 없음(영원히 보존)이 된다
byte	value로 보존되는 데이터의 byte 크기를 지정	–

telnet으로 local에 기동한 memcached의 11211번 포트로 접속하여 데이터 입력과 출력을 해 보았습니다. 데이터 입력시 조금 어려울지도 모르겠지만, 처음에는 데이터가 아니라 데이터의 byte 크기를 지정해 주십시오. 다음 행에 실제 데이터를 입력합니다.

이 외에도 memcached에서는 incr, delete, append, flush_all 등 여러 명령어를 사용할 수 있습니다.

```
# locahost의 11211번 포트에 접속
$ telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.

set counter 0 0 1
1
STORED

# 더하기 연산(빼기는 decr 명령어)
incr counter 1
2
```



```
incr counter 2
4

# 데이터 삭제
delete counter
DELETED

# 데이터 삭제 확인
get counter
END

set test 0 0 4
test
STORED

# 데이터 끝부분에 추가(앞부분 추가는 prepend)
# append <key> <flag> <expires> <byte> enter
# <value>
append test 0 0 3
add
STORED

get test
VALUE test 0 7
testadd
END

# memcached의 모든 데이터 삭제
flush_all
OK
```

Ruby에서 사용해 본다(1대)

그러면 이번에는 Ruby 프로그램에서 memcached를 사용해 봅시다. 이번에는 memcache-client라는 gem 라이브러리를 이용합니다. 설치는 아래와 같습니다. 버전은 1.8.5입니다.

```
gem install memcache-client
```

우선 간단한 데이터 입력과 출력을 시험해 봅니다.

```
$KCODE = 'u'

require "rubygems"
require "memcache"

server = ['localhost:11211']        # 이용할 memcached를 지정
option = {}

cache = MemCache.new(server, option)

# 데이터 입력
cache['key1'] = 123                          # 숫자
cache['key2'] = "abcde"                      # 문자열
cache['key3'] = %w(hoge fuga)                # 배열
cache['key4'] = {:foo => 1, :bar => "a"}     # hash

# 데이터 읽어오기
p cache['key1']      # 123
p cache['key2']      # "abcde"
p cache['key3']      # ["hoge", "fuga"]
p cache['key4']      # {:foo=>1, :bar=>"a"}
```

어떻습니까? Ruby를 사용해 보지 않은 분도 다루기가 간단하다는 것을 알 수 있을 것입니다. cache라는 인스턴스에 대응하는 적당한 key로 숫자, 문자열, 배열, Hash 등 여러 value를 입력하고 출력합니다. 숫자는 숫자, 문자열은 문자열, 배열은 배열, Hash는 Hash로 제대로 출력되는 것을 확인할 수 있습니다.

참고로 constructor에는 Hash 형식으로 옵션을 줄 수 있습니다. 설정할 수 있는 옵션은 아래와 같습니다.

표 2.3 memcache-client의 constructor에 주는 옵션

이름	설정되는 내용	초기값
namespace	데이터가 입력되는 namespace key 앞에 붙음	nil
readonly	출력 전용 입력하면 MemCacheError	false
multithread	thread-safe	true
failover	서버 다운시 다른 서버에서 처리할지	true
timeout	타임아웃 설정 nil로 설정하면 타임아웃이 없어짐	0.5초
logger	로그를 출력할 logger 지정	nil
no_reply	입력시 응답을 받을지 여부. false로 응답을 받지 않으면 퍼포먼스가 좋아짐 ※memcached 1.2.5 이후 이용 가능	false
check_size	value가 1MB보다 크면 MemCacheError	true
autofix_keys	key가 250문자 이상이거나 스페이스를 포함한 경우 key 대신 Digest::SHA1.hexdigest(key)를 이용한다	false

실제로 memcached의 처리는 Hash와 굉장히 비슷합니다. 실제로 비교해 보면 Hash와 같은 느낌으로 사용하는 것을 알 수 있습니다.

```
# memcached의 경우
cache = MemCache.new(['localhost:11211'])
cache['key'] = 'value'
p cache['key']   # "value"

# Hash의 경우
hash = {}
hash['key'] = 'value'
p hash['key']    #"value"
```

memcache-client에는 데이터의 입력과 출력에 각각 두 가지 방식이 있습니다. 데이터 입력은 set 메소드와 [ ]=메소드, 데이터 출력은 get 메소드와 [ ]메소드입니다(정확히 얘기하면 [ ]=와 [ ]메소드는 set과 get 메소드의 Alias입니다).

기본적으로 마음에 드는 쪽을 사용하면 됩니다만, 데이터 입력에는 set 메소드와 [ ]=메소드가 조금 다르기 때문에 주의가 필요합니다. 데이터 입력시 set에서는 expires를 자유롭게 설정할 수 있는 것에 반해, [ ]=메소드에서는 expires를 설정할 수 없어, 0으로 지정(즉, 반영구적으로 데이터 보존)하게 됩니다. 이러한 동작은 라이브러리에 따라 다르므로 구체적인 내용은 각 라이브러리의 도큐먼트를 확인하는 것이 좋습니다.

set 메소드를 사용해서 데이터를 입력할 경우, expires를 설정할 수 있다는 것을 확인해 보겠습니다. expires를 10초로 설정해서 10초 후에 데이터가 없어짐을 확인할 수 있습니다.

```
require "rubygems"
require "memcache"

cache = MemCache.new(['localhost:11211'])

cache.set('key', 'value', 10)    # expires는 10초
p cache['key']       # 'value'

sleep 10

p cache['key']       # nil (10초 후 expire됨)
```

Ruby에서 사용해 본다(여러 대)

여기까지는 memcached를 1대만 사용하였습니다. 다음으로는 memcached를 여러 대에 두고 이용해 봅시다.

memcached를 3대 이용하면 Consistent Hashing에 의해 데이터가 3대에 나눠집니다. 여러 대를 사용하는 것의 장점으로는 먼저 memcached에 할당되는 메모리가 각 서버에서 64MB(디폴트 설정일 경우)이므로, 합계 192MB(디폴트 설정일 경우)로 3배로 늘어납니다. 그러므로 1대일 때보다 3배의 데이터를 memcached의 메모리에 올릴 수 있습니다. 또한, 1대가 정지되더라도 일부 데이터에만 영향을 주게 됩니다. 1대로만 운영한다면 memcached가 정지되었을 때, memcached 상의 데이터가 모두 사라져버리지만 3대로 운영하고 있을 때는 데이터가 분산되어 있기 때문에 1/3의 데이터만 사라지게 됩니다. 이 1/3의 데이터는 Consistent Hashing에 의해 남은 두 대로 다시 분배됩니다.

그러면 실제로 시험해 보겠습니다. 먼저, 서비스 포트를 각각 할당하여(이번에는 11211, 11212, 11213) memcached를 3개 기동합니다.

```
memcached -d -p 11211 -u nobody -c 1024 -m 64
memcached -d -p 11212 -u nobody -c 1024 -m 64
memcached -d -p 11213 -u nobody -c 1024 -m 64
```

다음으로는 프로그램 코드의 constructor 인수를 통해 아래와 같이 배열에 설정하면 됩니다. memcached를 여러 대 사용하기 위해 설정할 것은 기본적으로 이 것뿐입니다. 단, 라이브러리에 Consistent Hashing이 포함되어 있어야 사용이 가능하므로 그 부분을 주의해 주시기 바랍니다. 지금 사용중인 memcache-client는 버전이 1.6.0으로 Consistent Hashing을 사용할 수 있습니다.[6] 단순하게 서버를 늘린 것만으로 Consistent Hashing을 통해 cache miss를 줄이면서 scale-out을 할 수 있습니다.

```
$KCODE = 'u'

require "rubygems"
require "memcache"
require "logger"

server = [
   'localhost:11211',
   'localhost:11212',
   'localhost:11213'
]
option = {
   :logger => Logger.new(STDOUT)
}

cache = MemCache.new(server, option)
```

6 https://github.com/mperham/memcache-client/blob/master/History.rdoc