번역중- **NoSQL** 데이터 모델링 기법

- 글을 소개해주신 분 - http://www.facebook.com/groups/engfordev/360167394035092/
- 원문 주소 - http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/
- 번역해서 커뮤니티 블로그에 올려도 된다고 저자의 허락을 받은 댓글
- http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/#comment-621
- 수정가능 -> 메모삽입 가능으로 바꿨습니다~
- 수정할 부분이 보이시면 이곳으로 와주세요~

작업 일지

- 0423 - 스피드 번역으로 약 70% 정도 1차 번역했음.
- 0425 - 3,4 페이지 보완했음.
- 여러 멤버가 개선작업을 해주고 계심..
- 0429 - 일단 1차 마무리해서 발행하기. 미완성인 채이더라도.

번역에 참여하실 예정인 분 **<-** 다른 분들도 써주세요~

Type1. 스피드하게 번역하는 역할 **(**영**->**한**)**

- Hong Hwanmin
- Dae-Seon Moon
- rkJun
- nasol
- Jaehyun Jang (@SilvesterJang)
- Justin Yoo (@justinchronicle)

Type2. 스피드하게 번역하는 역할**(**일문 버전을 번역기로 돌리고 한국어를 다듬는 역할**)**

- Dae-Seon Moon
- rkJun

Type3. 요기조기 왔다갔다 하면서 영어 관련 질문에 답변댓글 다는 역할

- nassol
- Jaehyun Jang (@Silvesterjang)

Type4. 요기조기 왔다갔다 하면서 **NoSQL,** 개발이나 용어관련 질문에 답변하는 역할

- 박범진(생각해보니 제가 NoSQL을 모르네요 ㅋ)
- 권민택(저도 멀라여 ㅋ)
- 정정식

Type5. 응원하고 홍보하는 역할

- nassol

참여하는 방법

- 번역에 참여하실 예정인 분에, 역할별로 이름을 써주세요~
- 만만한 부분을 대략 고르셔서 번역해주세요~

- 편하게 번역해주세요~
- 스피드 번역이니 만큼, 표현하나하나에 얽매이지 마시고,
- 중요한 정보 위주로 전달하는데 중점을 둬 주세요~

- 단, 막히는 부분은, 끙끙 앓지 마시고, 메모로 질문을 써 주시고, 넘어가주세요~
- 질문을 쓴 경우에는 **QQ.** 질문 쓰시공.. **Ctrl+1** 누르셔서 서식을 바꿔주세요~

- 애매한 부분은 원문을 분홍색으로 폰트색 바꾸시고 넘어가 주세요~

- 참고로.. 우선 번역기 돌려보실 분은,
- 일문판이 있으니 - https://gist.github.com/2396234
- 요것에서 번역하시려는 부분 일부분을 복사 & 번역기를 돌리시고, 수정하시면 됩니다 :)

함께 결정한 사항

- Full text search engine 표시방법 - 전문(Full text) 검색 엔진
- http://www.facebook.com/groups/engfordev/367042396680925/ -> 바로 적용하기는 어렵더라도, 한국어로 표현하는 것에 대한 아이디어를 모아 두었음.
-

NoSQL Data Modeling Techniques

Posted on March 1, 2012

29

NoSQL databases are often compared
by various non-functional criteria,
such as **scalability, performance, and consistency.**

시스템을 평가하는 기준은 크게 두 가지로 나눌 수 있다. 1) '무엇'을 할 수 있는지를 보는 '기능적 기준과 2) '얼마나 잘'할 수 있는지를 보는 '비기능적' 기준 (확장성, 성능 등)

NoSQL 데이터 베이스를 비교할 때에는, '비기능적' 기준을 가지고 비교한다.

확장성, 성능, 일관성과 같은 다양한 '비기능적' 기준으로 NoSQL 데이터베이스들끼리 비교하는 것은 종종 있는 일이다.

This aspect of NoSQL is **well-studied** both in practice and theory
because
specific non-functional properties
are often the **main justification for NoSQL usage**
and fundamental results on distributed systems
like the CAP theorem apply well to NoSQL systems.

NoSQL의 이러한 측면(non-functional criteria)은 이론과 실전 측면에서 연구가 잘 되어 있다. 왜냐하면 기능과 무관한 특정 속성 때문에 NoSQL을 사용하기 때문이다. 그리고 CAP정리를 비롯하여 분산시스템의 본질에 대한 연구결과가 NoSQL 시스템에도 잘 적용되기 때문이다. 때문이다.

t)s)

At the same time,
NoSQL data modeling is **not so well studied** and **lacks the systematic theory**
found in relational databases.

하지만 관계형 DB와는 다르게 NoSQL 데이터 모델링은 잘 연구가 되어 있지 않고, 체계적인 이론이 부족하다.

// 이 글에서 다루려는 내용은 뭐냐면**~**

In this article I provide a **short comparison of NoSQL system families**
from the data modeling point of view
and digest several common modeling techniques.

이 글에서는**, NoSQL** 시스템군을**,** 데이터 모델링의 관점에서 간략히 비교하고**,**
흔히 쓰는 모델링 기법을 몇 개 자세히 다룰 것이다**.**

I would like to thank Daniel Kirkdorffer
who reviewed the article and cleaned up the grammar.

이 글을 검토해주고 문법적으로 교정해준 Daniel Kirkdorffer에게 감사드린다.

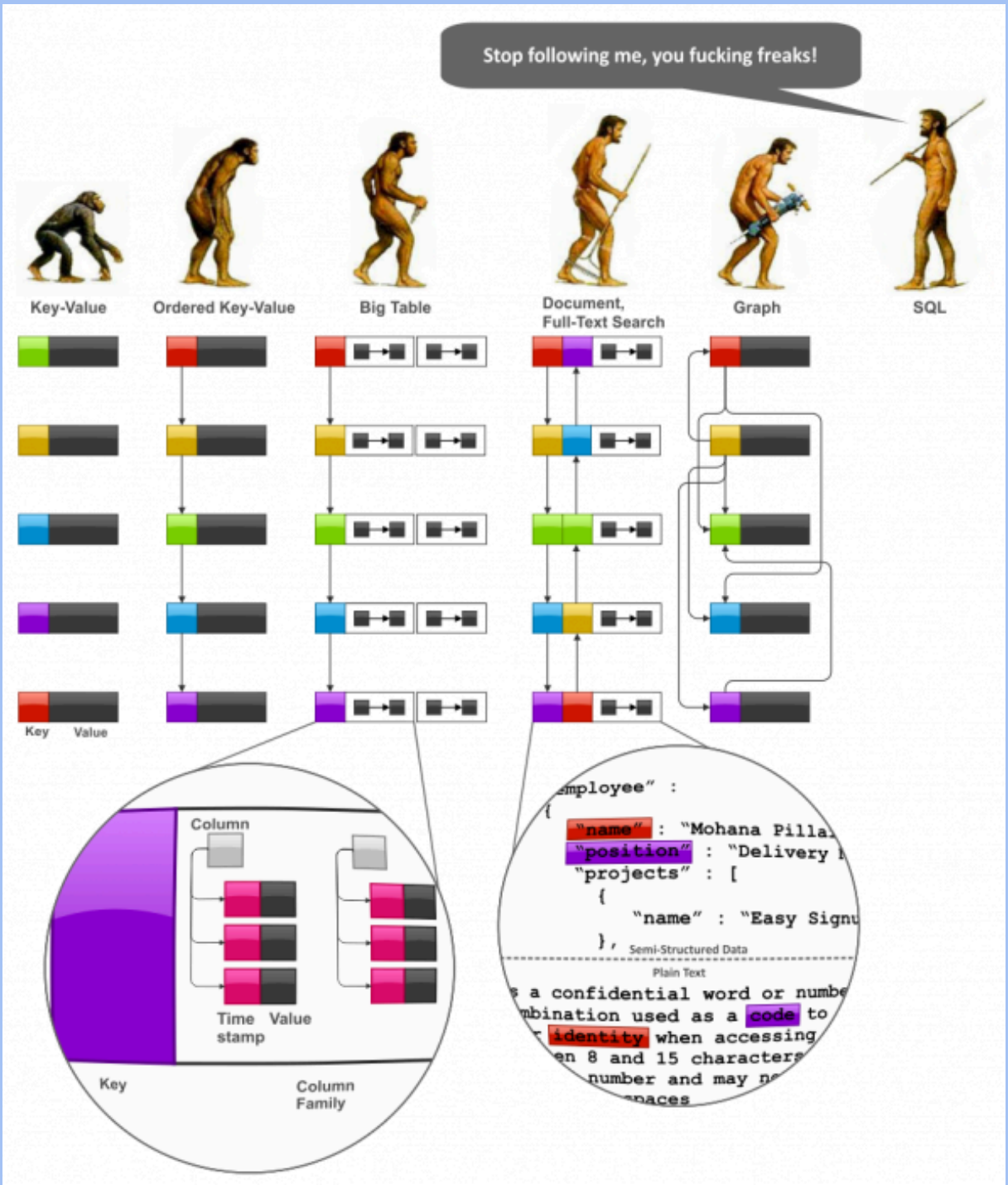
To explore data modeling techniques,
we have to start with a more or less **systematic view of NoSQL data models**
that preferably reveals trends and interconnections.

데이터 모델링 기법을 살펴보려면,
NoSQL 데이터 모델을 체계적으로 살펴보는 것부터 시작해야 한다.

The following figure depicts imaginary “evolution”
of the major NoSQL system families,
namely,
Key-Value stores,
BigTable-style databases,
Document databases,
Full Text Search Engines,
and Graph databases:

아래 그림을 보면 주요 NoSQL 시스템군의 진화를 볼 수 있다.
이름하여, 키-밸류 스토어(Key-value store), 빅테이블형 데이터베이스, 다큐먼트 데이터베이스, 전문(Full Text) 검색엔진, 그래프 데이터베이스 :

0425 - 여기까지 보완했음.



NoSQL Data Models

First, we should note that
SQL and relational model in general
were designed long time ago
to interact with the end user.

This user-oriented nature
had vast implications:

우선, 이점에 주목하자.
SQL과 관계형 모델은 오래 전에 설계되었는데,
엔드유저와 상호작용하는 것을 염두에 두고 설계되었다.

이것은 다음의 두 가지를 의미한다.

- The end user is often interested in **aggregated reporting information,**
  not in separate data items,
  and SQL pays a lot of attention to this aspect.
  최종사용자는 개별 데이터가 아니라, 정보를 모아서 보고 받는 것에 관심이 있고, SQL은 이런 측면에 많이 초점을 맞추고 있다.

  관심을 할애한다.

- No one can expect human users
  to explicitly control concurrency, integrity, consistency, or data type validity.
- 어느 누구도 인간이 병렬성, 무결성, 일관성, 데이터 타입 적합성을 정확히 제어할 수 있다고 기대하지 않는다.

  That's why SQL pays a lot of attention
  to transactional guaranties, schemas, and referential integrity.

위에 언급한 부분이 SQL이 트랙잰션 보증, 스키마, 참조 무결성에 대해 많은 주의가 필요한 이유다.

제안>> 그래서 SQL에서는 트랜잭션 보장, 스키마, 참조 무결성에 대해 주의를 많이 기울인다.

On the other hand,
it turned out that software applications
are not so often interested in in-database aggregation and able to control,
at least in many cases, integrity and validity themselves.

엔드유저와는 다르게, 소프트웨어 어플리케이션은 데이터베이스 내에서 이미 집계한 데이터를 받는 것에는 별로 관심이 없다. 그리고 소프트웨어 어플리케이션은 많은 경우, 무결성과 적합성을 제어할 수가 있다.

Besides this, **elimination of these features**
had an extremely i**mportant influence**
on the performance and scalability of the stores.

그 밖에도, 이 데이터베이스 내에서 데이터를 집계하는 기능을 제거한 것은 저장소의 성능과 확장성에 있어서 엄청나게 중요한 영향을 끼쳤다.

And this was where a **new evolution of data models began:**
바로 이 지점에서 데이터 모델의 새로운 혁명이 시작되었다 :

- Key-Value storage is a very simplistic, but very powerful model. Many techniques that are described below are perfectly applicable to this model.

- Key-Value storage는 아주 간단하지만 강력한 모델이다. 밑에서 서술한 많은 기법들은 Key-Value Storage모델이 완벽하게 적용된다..

- One of the most significant shortcomings of the Key-Value model is a poor applicability to cases that require processing of key ranges. Ordered Key-Value model overcomes this limitation and significantly improves aggregation capabilities.

  - Key-Value 모델의 가장 큰 단점중에 하나는 Key 범위들을 처리하는 것을 요구하는 경우에 대한 빈약한 적용성이다. 정렬된 Key-Value 모델은 이러한 한계를 극복하고 집계 능력을 상당히 향상시킨다.

  - 제안>> Key-Value 모델의 단점 중 하나는 Key 범위를 처리해야 하는 경우에는 잘 적용되지 않는다는 점이다.

- Ordered Key-Value model is very powerful, but it **does not provid**e any **framework for value modeling.** In general, value modeling can be done by an **application**, but **BigTable-style databases** go further and model values as a map-of-maps-of-maps, namely, column families, columns, and timestamped versions.
- 정렬된 Key-Value 모델은 강력하긴 하지만 값을 구조화하는 것에 대한 프레임워크를 제공하지 않는다. 보통, 어플리케이션에서 값을 구조화할 수 있다. 그러나 빅테이블형 데이터베이스에서는 정렬된 Key-Value 모델보다 한 걸음 더 나아가서, 맵안의 맵안의 맵

형태, 즉 컬럼군, 컬럼, 타임스탬프 버전의 형태로 밸류(값)을 구조화한다.

-
-
- 는 더 나아가서 모델의 value들을 마치 매핑하거나 (으아 모르게씀 ㅜㅜ) (뒷 부분의 의미를 잘 모르겠어서 패스했어요~)
  (제안>>그러나 BigTable-style 데이터베이스는 더 나아가서 맵안의 맵안의 맵 형태, 즉, 컬럼군, 컬럼, 타임스탬프 버전 형태로 밸류(값)를 모델링한다.)
- >> 고맙습니다~붙여서 써봤어요~

- **Document databases** advance the BigTable model **offering two significant improvements**. The first one is **values with schemes of arbitrary complexity**, not just a map-of-maps. The second one is **database-managed indexes**, at least in some implementations. Full T**ext Search Engines** can be considered a **related species** in the sense that they also offer **flexible schema and automatic indexes. The main difference** is that Document database **group** indexes by fie**ld name**s, as opposed to Search Engines that **group** indexes by fie**ld value**s. It is also worth noting that some Key-Value stores like Oracle Coherence gradually move towards Document databases **via addition of indexes and in-database entry processors.**

- 다큐먼트 데이터베이스는 빅테이블형 모델에 비해 두 가지 면에서 낫다. 첫번째는 값의 스키마의 복잡한 정도가 임의성을 띤다는 점이다. 두번째는, 데이터베이스에서 인덱스를 관리한다는 것이다. 적어도 몇몇 구현부분에서는 말이다. 전문(Full Text) 검색 엔진이 유연한 스키마와 automatic 인덱스를 제공하는 점을 보면, 전문(Full text) 검색 엔진은 다큐먼트 데이터베이스와 친척쯤이라고 볼 수 있다. 다큐먼트 데이터베이스와 전문(Full Text) 검색엔진의 주요한 차이점은 이러하다. 다큐먼트 데이터베이스는 필드명을 가지고 인덱스를 그룹으로 묶는 반면에, 전문(Full Text) 검색 엔진의 경우에는 필드의 값을 가지고 인덱스를 그룹으로 묶는다. 오라클 Coherence 등등의 몇몇 키-밸류 스토어 등등이 인덱스 기능과 데이터베이스 내장 엔트리 처리기를 갖추면서 다큐먼트 데이터베이스로 옮겨가고 있다는 점도 눈여겨 볼 만하다.

- Finally, Graph data models can be considered as a side branch of evolution that origins from the Ordered Key-Value models. Graph databases allow one model business entities very transparently (*this depends on that*), but hierarchical modeling techniques make other data models very competitive in this area too. Graph databases are related to Document databases because many implementations allow one model a value as a map or document.
- 마지막으로 그래프 데이터 모델은 정렬된 키-밸류 모델에서 곁가지로 진화해 온 결과라고 볼 수도 있다. 그래프 데이터베이스를 사용하면 비즈니스 엔터티를 매우 투명하게 모델링할 수

있다. 그러나, 위계질서가 있는 모델링 테크닉을 사용하는 다른 데이터 모델도 비즈니스 엔터티를 투명하게 모델링하는 분야에서 매우 경쟁력이 있다. 그래프 데이터베이스는 다큐먼트 데이터베이스하고도 연관이 있다. 왜냐하면 구현할 때, 많은 경우, 값을 맵이나 다큐먼트로 모델링할 수 있기 때문이다.

General Notes on NoSQL Data Modeling

NoSQL 데이타 모델링에 대한 짤막한 언급

요건 어떨까요**? >> NoSQL** 데이터 모델링에 에 대해 전반적으로 언급한 것

// 이 부분 다음 부터는 데이터 모델링 기법과 패턴에 대해서 세부적인 얘기를 할 것**..**

The rest of this article describes concrete data modeling techniques and patterns. As a preface, I would like to provide a few general notes on NoSQL data modeling:

이 글의 나머지는 구체적인 데이타 모델링 테크닉과 패턴에 대해 설명한다. 우선, 몇가지 데이타 모델링에 대한 일반적인 사항에 대해 언급하겠다.

- NoSQL data modeling often starts from the application-specific queries as opposed to relational modeling:
- NoSQL 데이타 모델링은 관계형 데이타 모델링과 반대로 종종 응용프로그램에 특화시킨 쿼리문으로부터 출발한다.

- 제안**>>** 관계형 데이터 모델링과는 다르게 NoSQL 데이터 모델링을 할 때에는 종종 특정 응용프로그램에서만 쓸 수 있는 쿼리문으로부터 출발한다.

    - Relational modeling is typically driven by the structure of available data. The main design theme is **"What answers do I have?"**
    - 관계형 데이타 모델링은 보통 가능한 데이타 구조에서 출발한다. 따라서 디자인의 주제는 "무슨 대답을 내가 갖고 있나?" 이다.

    - 제안**>>** 관계형 데이터 모델링을 할 때에는 보통 실제의 데이터의 구조에서 출발한다. 따라서 데이터를 모델링할 때 던지는 중요한 질문은 "데이터베이스는 어떤 답을 갖고 있는가?"이다.

    - NoSQL data modeling is typically driven by application-specific access patterns, i.e. the types of queries to be supported. The main design theme is **"What questions do I have?"**
    - 반면에 NoSQL 데이타 모델링은 보통 응용프로그램에 특정한 패턴을 갖고 있다다. 따라서 디자인의 주제는 "무슨 질문을 내가 갖고 있나?" 인 셈이다.

    - 반면에 NoSQL 데이터 모델링을 할 때에는 특정 어플리케이션에서 접근하는 패턴에서 출발한다. 따라서 데이터를 모델링할 때 던지는 중요한 질문은 "어플리케이션에서 던지는 질문은 무엇인가?"이다.

- NoSQL data modeling often requires a deeper understanding of data structures and algorithms than relational database modeling does. In this article I describe several well-known data structures that are not specific for NoSQL, but are very useful in practical NoSQL modeling.
-
- NoSQL 데이타 모델링은 종종 관계형 데이타 모델링에서 요구하는 것보다 데이타 구조와 알고리즘에 대한 훨씬 더 깊은 이해를 필요로 한다. 이 글에서는 몇가지 잘 알려진, 딱히 NoSQL에 한정짓지 않지만 실제로 NoSQL 모델링에 굉장히 유용한, 데이타 구조들에 대해 설명할 것이다.

- 제안>> 관계형 데이터 모델링에 비해, NoSQL 데이터 모델링을 하려면 데이터 구조와 알고리즘에 대해 깊이 있게 이해해야 한다. 이 글에서는 NoSQL에만 해당하는 것은 아니지만, NoSQL 모델링을 할 때에 매우 유용하며, 잘 알려진 데이터 구조 몇몇에 대해 설명하겠다.

- Data duplication and denormalization are **first-class citizens.**
- 데이타 중복과 비정규화는 가장 중요한 내용이다.
-
- Relational databases are not very convenient for hierarchical or graph-like data modeling and processing. Graph databases are obviously a perfect solution for this area, but actually most of NoSQL solutions are surprisingly strong for such problems. That is why the current article devotes a separate section to hierarchical data modeling.
-
- 관계형 데이타베이스는 계층적이거나 그래프 형태의 데이타 모델링과 프로세싱에 그다지 편리하지 않다. 그래프형 데이타베이스는 이 분야에서 완벽한 솔루션같아 보이지만, 실제로 대부분의 NoSQL 솔루션이 놀랍게도 이러한 문제들에 대해 강점을 가진다. 그래서 이 글에서는 계층적인 데이타 모델링에 별도의 섹션을 할당하였다.

- 제안>> 계층적이거나 그래프와 비슷하게 데이터를 모델링하거나 처리하는 경우에는 관계형 데이터 베이스가 그다지 편리하지 않다. 이런 경우에는 물론 그래프 데이터베이스가 완벽한 솔루션이다. 하지만 대부분의 NoSQL 솔루션도 이런 문제에 놀라울 정도로 강력하다. 그래서 이 글에서 별도의 섹션을 두어, 위계적인 데이터 모델링에 대해 다루려고 한다.

Although data modeling techniques are basically implementation agnostic, this is a list of the particular systems that I had in mind while working on this article:

비록 데이타 모델링 기법이 기본적으로 개발을 하기 전까지는 아무도 모르는 것이라곤 해도,

아래의 것들은 이 글을 작업하는 동안 염두에 두었던 특정 시스템들의 리스트이다.

- Key-Value Stores: Oracle Coherence, Redis, Kyoto Cabinet
- BigTable-style Databases: Apache HBase, Apache Cassandra
- Document Databases: MongoDB, CouchDB
-
- Full Text Search Engines: Apache Lucene, Apache Solr
- Graph Databases: neo4j, FlockDB
-
- 키-밸류 스토어 : Oracle Coherence, Redis, Kyoto Cabinet
- 빅테이블형 데이터베이스: Apache HBase, Apache Cassandra
- 다큐먼트 데이타베이스: MongoDB, CouchDB
- 전문(Full Text) 검색엔진: Apache Lucene, Apache Solr
- 그래프 데이타베이스: neo4j, FlockDB

Conceptual Techniques

개념적 기술

This section is devoted to the basic principles of NoSQL data modeling.

이 섹션에서는 NoSQL 데이터 모델링의 기본 원리에 대해 다룬다.

(1) Denormalization

(1) 비정규화

Denormalization can be defined as the copying of the same data into multiple documents or tables in order to simplify/optimize query processing or to fit the user's data into a particular data model. Most techniques described in this article leverage denormalization in one or another form.

비정규화는 다수의 문서나 테이블에 같은 데이터의 복사본을 정의할 수 있다. 쿼리 처리를 단순화하거나 최적화하기 위해서, 또는 사용자의 데이터를 특정한 데이터 모델에 맞추기 위해서 말이다. 대부분의 기술들은 비정규화에 대한 이 문서에서 설명하였다.

제안>> 비정규화란 쿼리 처리를 단순화하거나 최적화하기 위해, 또는 유저의 데이터를 특정 데이터 모델에 맞추기 위해, 동일한 데이터를 여러개의 다큐먼트나 테이블에 복사하는 것을 의미한다. 이 문서에서 다룬 기법들은 대부분 다양한 방식으로 비정규화를 사용한다.

In general, denormalization is helpful for the following trade-offs:

일반적으로, 비정규화는 다음과 같은 트레이드오프에 도움이 된다.

- *Query data volume* or *IO per query* VS *total data volume*.
-

(제안) 질의할 때 입출력 부하를 줄일지 아니면 데이터 저장공간 부담을 줄일지 선택

- Using denormalization **one can group all data** that is needed to process a query **in one place.**
-
- 비정규화를 사용하면, 쿼리를 한 군데에서 모두 처리하기 위해 필요한 데이터를 모두 그루핑할 수 있다.
-
- 비정규화를 사용하면, 어떤 질의를 처리하는데 필요한 모든 데이터를 (여러 곳에 나눌 필요 없이) 한 곳에 모아둘 수 있고, 따라서 입출력부담을 줄이는데 도움이 된다.

- 쿼리당 쿼리데이타(?) 쿼리당 IO VS 전체 데이타 사이즈. 비정규화를 사용함으로써 하나의 장소에 하나의 쿼리만 사용해야했던 모든 데이타들을 모을 수 있게 되었다?????
-
- This often means that
  for different query flows
  the same data will be accessed in different combinations.

  이것이 의미하는 것은, 여러가지 쿼리 플로우에 대해서, 같은 데이터가 다양하게 조합된 방식으로 접근된다는 것이다.

- 그런데 종종 질의마다 다른 데이터들을 조합하므로, 동일한 데이터가 다른 조합으로 접근된다.

- 같은 데이타에 대해 다른 방법(조합)으로 조회할 수 있다는것을 의미한다.
-
- Hence we need to duplicate data, which increases total data volume.
- 결국 데이터를 복사해야 하게 되고, 그 결과 전체 데이터의 양이 증가한다.
-
- 따라서 비정규화를 통해 각 질의에 맞는 조합으로 데이터들을 모은다면 결국 같은 데이터를 여러 번 복사해야 하고 그 결과 전체 데이터 저장공간을 더 많이 차지하게 된다.

- 그러므로 (비정규화를 위해) 데이타를 중복시켜야한다는것은 전체 데이타 사이즈를 증가시킨다.
-

Processing complexity VS *total data volume*.
복잡성을 처리하는 것 *Vs* 전체 데이터의 양

Modeling-time normalization and consequent query-time joins obviously increase complexity of the query processor, especially in distributed systems. Denormalization allow one to store data in a query-friendly structure to simplify query processing.

처리할 때 복잡도를 줄일지 아니면 데이터 저장공간 부담을 줄일지 선택

모델링할 때는 동일 데이터는 한번만 저장하는 정규화를 적용하고 질의할 때 조합(join)하는 방법을 선택하면 질의 처리 엔진이 복잡해질 수밖에 없다. 특히 분산시스템에서 더더욱 복잡해진다. 대신 비정규화를 적용하면 질의 처리를 단순화할 수 있는 구조로 데이터를 저장할 수 있다.

Applicability: Key-Value Stores, Document Databases, BigTable-style Databases

적용 가능한 데이터 모델 : 키-밸류 스토어, 다큐먼트 데이터베이스, 빅테이블형 데이터베이스

(2) **Aggregates** (일단 원강민 찜^^ 이렇게 하는거 맞나요? ㅎㅎ)
(2) 집계

All major genres of NoSQL provide soft schema capabilities in one way or another:
NoSQL의 주요 장르는 어느 방식으로든 소프트 스키마 기능을 제공합니다.

- Key-Value Stores and Graph Databases typically do not place constraints on values, so values can be comprised of arbitrary format. It is also possible to vary a number of records for one business entity by using composite keys. For example, a user account can be modeled as a set of entries with composite keys like *UserID_name, UserID_email, UserID_messages* and so on. If a user has no email or messages then a corresponding entry is not recorded.

키-밸류 저장소와 그래프 데이터데이스에서는 일반적으로 값에 제약 조건을 걸지 않는다. 그래서 임의의 형식으로 값을 구성할 수 있다. 또한 하나의 비즈니스 엔티티에 대해, 복합키를 사용하는 방법으로, 레코드의 수를 바꿀 수도 있다. 예를 들어, UserID_name, UserID_email, UserID_messages 등과 같은 복합키를 사용하여 사용자 계정을 항목의 집합(set of entry)으로 모델링 할 수 있다. 사용자의 이메일주소가 없거나, 메세지가 없는 경우에는 해당 항목(entry)은

기록되지 않는다.

>> 0425 일단 요기까지 제안..

- BigTable models supp**ort soft schema via a variable set of columns within a** *column family* and a **variable number of *versions*** for one *cell*.

- Document databases are inherently schema-less, although some of them allow one to validate incoming data using a user-defined schema.

다큐먼트 데이터베이스는 본질적으로 스키마가 없다. 하지만 다큐먼트 중 일부에서는 유저가 정의한 스키마를 사용해서 입력하는 데이터가 유효한지를 검증할 수도 있다.

비록 데이터베이스의 일부는 사용자 정의된 스키마를 사용하여 들어오는 데이터를 확인할 수 있지만, 도큐먼트 데이터베이스는 본질적으로 스키마가 적다.

Soft schema allows one to form classes of entities with complex internal structures (nested entities) and to vary the structure of particular entities.This feature provides two major facilities:

소프트 스키마를 사용하면 복잡한 내부 구조를 가진 엔티티(중첩된 엔티티)의 클래스를 형성할 수 있고 특정 엔티티의 구조를 변경할 수 있습니다. 이 기능으로 인해 두 가지 주요

- Minimization of one-to-many relationships by means of nested entities and, consequently, reduction of joins.

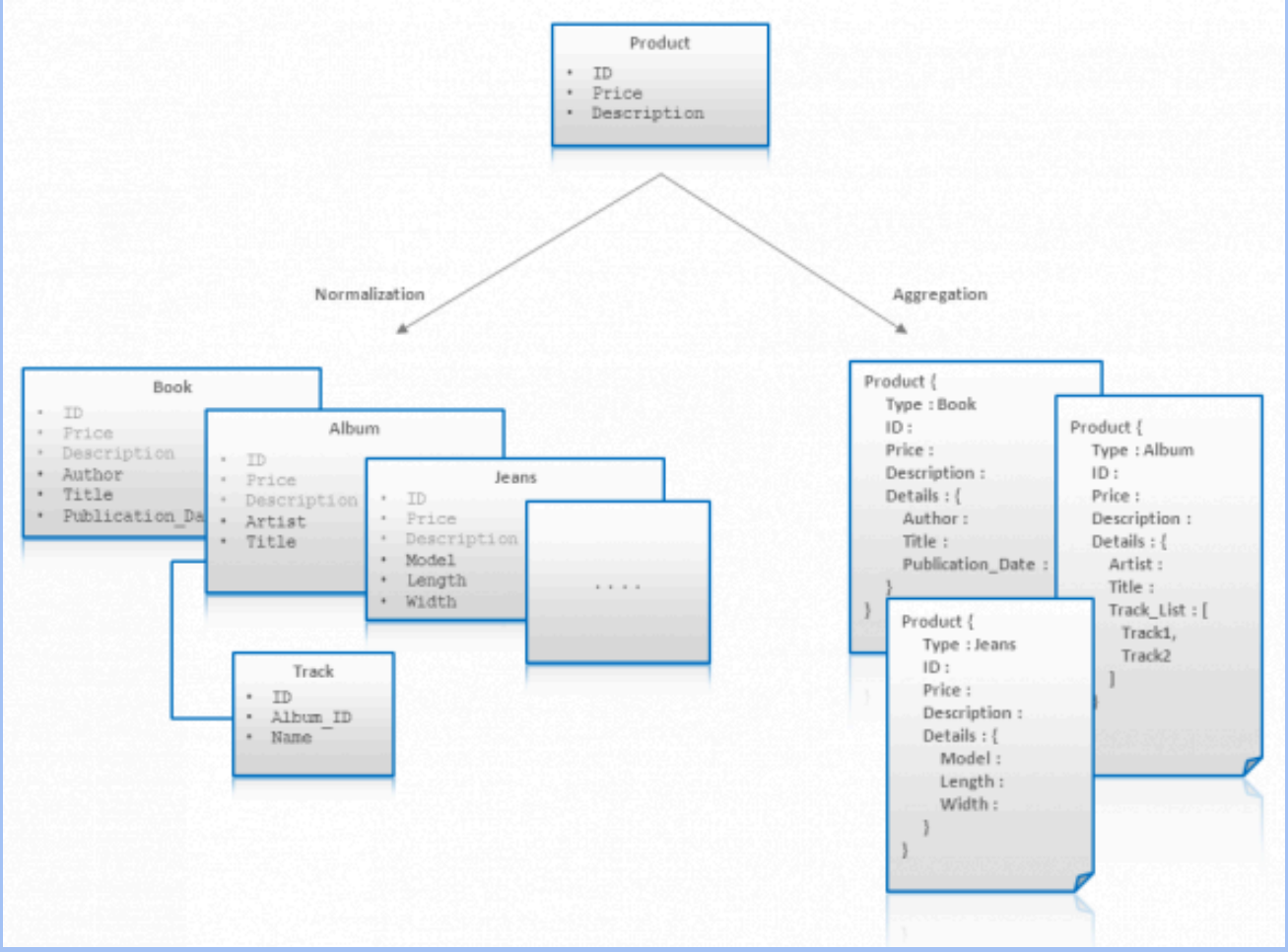
중첩된 엔티티에 의해 일대다 관계의 최소화, 그리고 결과적으로 조인의 감소.

- Masking of “technical” differences between business entities and modeling of heterogeneous business entities using one collection of documents or one table.

These facilities are illustrated in the figure below.
This figure depicts modeling of a product entity for an eCommerce business domain. **Initially,** we can say that **all products have an ID, Price, and Description. Next,** we discover that d**ifferent types of product**s have different **attributes** like **Author for Book or Length for Jean**s.
Some of these attributes have a **one-to-many or many-to-many nature** like Tracks in Music Albums.
Next, it is possible that s**ome entities can not be modeled** using **fixed type**s at all. For example, **Jeans attributes** are not consistent across brands and specific for each manufacturer.
It is possible to overcome all these issues in a **relational normalized data mode**l,

but **solutions are far from elegant(**가능은 하지만 우아하지가 않다**)**. **Soft schema** allows one to use a **single Aggregate (product)** that can model all types of products and their attributes:

- 이러한 기능들은 아래의 그림에 나타나있다. 이 그림은 전자 상거래 비즈니스 도메인 제품의 실체의 모델링을 나타냈다. 우선, 모든 제품들이 ID, 가격, 그리고 설명 정보를 갖고 있다고 해보자. (죄송합니다. 집중력이 떨어지네요 ㅜㅜ; 여기까지 하겠습니다.) 그리고 제품의 유형마다 다양한 속성을 갖고 있다. 예를 들어 청바지의 경우에는 길이 속성이 있고, 책의 경우에는 저자 속성이 있다. 어떤 속성들은 1대다, 다대다의 성격을 가진다. 이를테면, 음악 앨범의 트랙 같은 경우가 그렇다. 그리고 몇몇 entity는 고정된 유형으로는 모델링을 할 수가 없다. 예를 들어, 청바지의 속성은 브랜드별로, 제조사별로 다 다르다. relational normalized data model을 사용해서 이런 문제를 해결하는 것이 가능하기는 하지만, 이런 방법은 우아하지가 않다. 소프트한 스키마를 사용하면 (유연한 스키마??) single Aggregate(product)를 사용할 수 있습니다. single Aggregate를 사용하면 모든 종류의 제품과 이들의 속성을 모델링할 수 있습니다.



Entity Aggregation

Embedding with denormalization can greatly impact updates both in performance and consistency, so special attention should be paid to update flows.

비정규화는 업데이트의 성능과 지속성에 큰 영향을 미칩니다. 그래서 특별한 업데이트 흐름에 특별한 주의가 필요합니다.

Applicability: Key-Value Stores, Document Databases, BigTable-style Databases

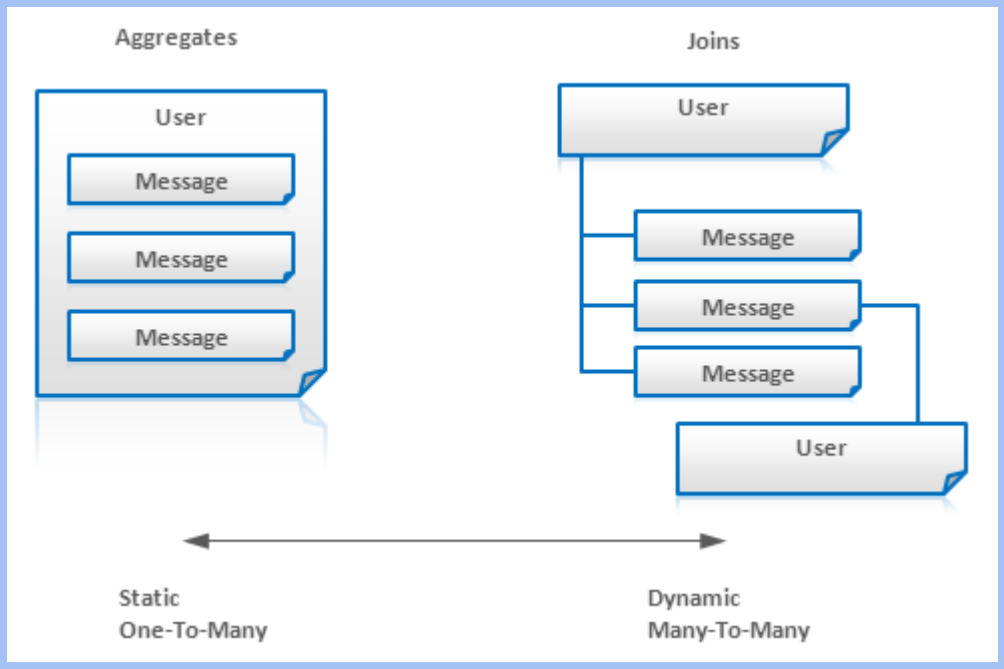
(3) Application Side Joins

Joins are rarely supported in NoSQL solutions. 조인은 NoSQL제품군에서는 거의 지원되지 않는다. As a consequence of the "question-oriented" NoSQL nature, joins are often handled at design time as opposed to relational models where joins are handled at query execution time.
NoSQL 기본인 질의지향으로 인해 조인은 쿼리 실행시 다루어지는 관계형 모델과 달리 디자인시에 다루어지게 됩니다.
Query time joins almost always mean a performance penalty,
쿼리 실행시의 조인은 언제나 성능상에 부하를 가져옵니다.
but in many cases one can avoid joins using Denormalization and Aggregates, i.e. embedding nested entities. Of course, in many cases joins are inevitable and should be handled by an application. The major use cases are:
하지만 많은 경우 집계와 비정규화를 통해서 조인을 사용하지 않을 수 있습니다, 예를들면 연관 종속 속성의 많은 경우 조인을 사용해야만하고 어플리케이션 레벨에서 다루어져야 합니다. 주로 사용되어지는 경우는 아래와 같습니다.

- Many to many relationships are often modeled by links and require joins.
- 다-대-다 관계는 종종 중간 링크로 모델링 되고 조인이 필요합니다.

- Aggregates are often inapplicable when entity internals are the subject of frequent modifications. It is usually better to keep a record that something happened and join the records at query time as opposed to changing a value . For example, a messaging system can be modeled as a User entity that contains nested Message entities. But if messages are often appended, it may be better to extract Messages as independent entities and join them to the User at query time:

- 집계의 경우 내부 속성이 자주 수정되어질 경우 종종 적용이 불가능해집니다. 이것은 보통 레코드로 유지되는것이 바람직하고 값을 바꾸는 것보다는 쿼리시에 조인이 되는것이 좋습니다. 예를 들면, 메세징 시스템은 종속 메세지 속성을 갖는 속성으로 정의될수 있을것입니다. 하지만 만약 메세지가 자주 추가 된다면, 메세지를 추출해서 독립적인 속성으로 하고 쿼리시에 조인을 하는것이 바람직할 것입니다.
-

- 

Applicability: Key-Value Stores, Document Databases, BigTable-style Databases, Graph Databases

General Modeling Techniques

일반적인 모델링 기술

In this section we discuss general modeling techniques that applicable to a variety of NoSQL implementations.

이 섹션에서 우리는 다양한 NoSQL구현에 적용가능한 일반적인 모델링 기술에 대해서 다룰것입니다.

(4) Atomic Aggregates - 으으 적절한 단어가 생각이 안나네여;

Atomic aggregate: indicates that a router has aggregated several routes into a larger block (애초에 이거 전문용어네요)

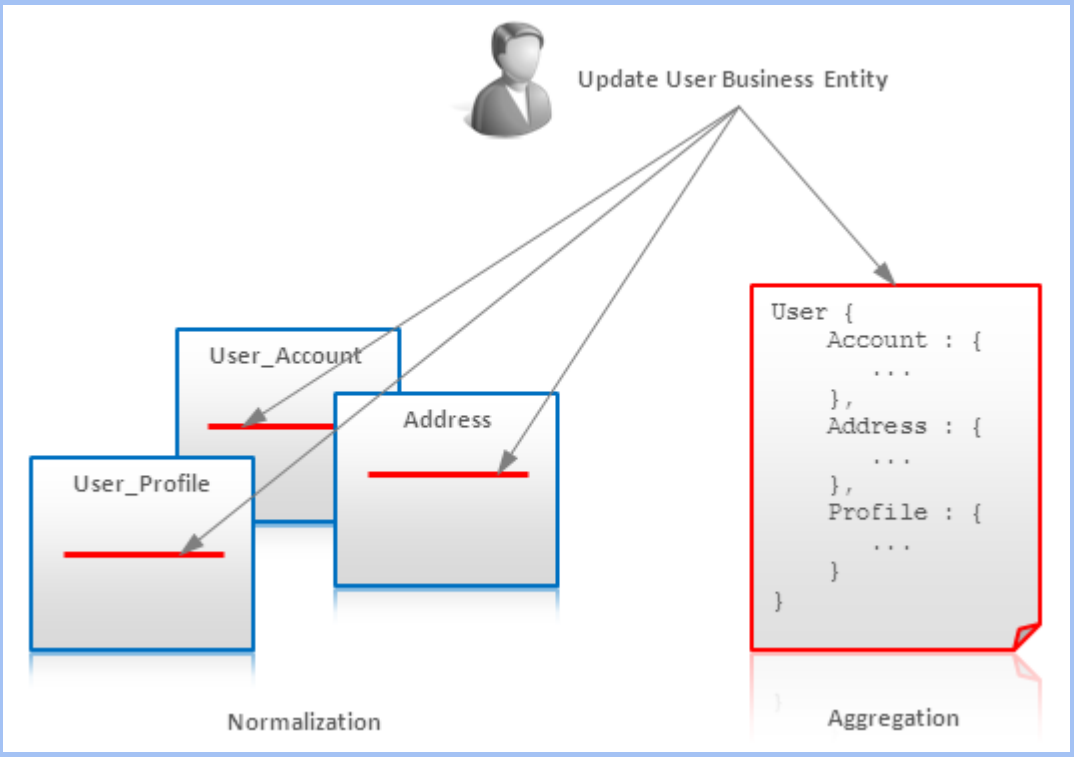
Many, although not all, NoSQL solutions have limited transaction support. In some cases one can achieve transactional behavior using distributed locks or application-managed MVCC, but it is common to model data using an Aggregates technique to guarantee some of the ACID properties.

전부는 아니지만, 많은 NoSQL 솔루션은 한정적인 트랜잭션을 지원하고 있다. 분산 락 (Lock) 이나, application-managed MVCC 를 사용해서 트랜잭셔널한 행동을 달성하고 있는 것도 있지만, ACID 속성의 일부를 보증하기 위한 집계기법을 사용해서 데이터를 모델링하는 것이 일반적이다.

One of the reasons why powerful transactional machinery is an inevitable part of the relational databases is that normalized data typically require multi-place updates. On

the other hand, Aggregates allow one to store a single business entity as one document, row or key-value pair and update it atomically:

파워풀한 트랜잭션구성이 관계형 데이터베이스의 불가결한 부분인 이유중 하나는, 정규화된 데이터는 전형적으로 복수의 업데이트가 필요해지기 때문이다. 한편, 집계(Aggreagte)에서는 하나의 비지니스 엔티티를 하나의 문서, 행 또는 key-value 짝으로 저장하고, 자동으로(-> 개별로) 업데이트하는 게 가능해진다.



Atomic Aggregates

Of course, Atomic Aggregates as a data modeling technique is not a complete transactional solution, but if the store provides certain guaranties of atomicity, locks, or test-and-set instructions then Atomic Aggregates can be applicable.

물론, 데이터 모델링 기법으로써의 Atomic Aggregates 는 완벽한 트랜잭션 솔루션은 아니지만, 만일 스토어(store)가 어느 정도 일관성보증, 락(locks), test-and-set 구성을 제공하는 경우, Atomic Aggregates는 적용가능하다.

Applicability: Key-Value Stores, Document Databases, BigTable-style Databases

적용범위 : 키-값 (key-value) 스토어, 문서 데이터베이스, 빅테이블 스타일 데이터베이스

(5) Enumerable Keys

열거가능한 키

Perhaps the greatest benefit of an unordered Key-Value data model is that entires can be partitioned across multiple servers by just hashing the key. Sorting makes things more complex, but sometimes an application is able to take some advantages of ordered keys even if storage doesn't offer such a feature. Let's consider the modeling of email messages as an example:

아마도 비정렬된 키-값 데이터모델의 최대의 이점은 키를 해싱하는 것만으로도 엔트리를 복수의 서버로 파티션할 수 있다는 점이다.

정렬은 복잡하게 만들 수 있으나, 가끔 어플리케이션은 비록 스토리지가 그런 특징을 제공하지 않는다고 해도 정렬된 키를 갖는 것이 더 이점이 있다. 자 이메일 메시지 모델링의 예를 들어 생각해보자.

1. Some NoSQL stores provide atomic counters that allow one to generate sequential IDs. In this case one can store messages using *userID_messageID* as a composite key. If the latest message ID is known, it is possible to traverse previous messages. It is also possible to traverse preceding and succeeding messages for any given message ID.

몇몇 NoSQL 저장소는 일련증가 ID를 생성하는 atomic counter를 제공해줍니다. 이 경우에 메세지를 userID_MessageID라는 복합키로 저장할수 있습니다. 만약 가장 최근의 메세지 ID가 주어진다면, 그 전의 메세지를 찾을 수도 있습니다. 이것은 또한 주어진 어떠한 메세지 ID에 관해서 그 바로 전이나 그 바로 후의 메세지를 찾을 수 잇습니다.

2. Messages can be grouped into buckets, for example, daily buckets. This allows one to traverse a mail box backward or forward starting from any specified date or the current date.

메세지는 daily bucket등으로 그룹화될수 있습니다. 이것은 어플리케이션으로 하여금 메일박스를 특정한 날짜는 현재 날짜로 시작하는 메세지들을 검색할수 있게합니다.

Applicability: Key-Value Stores

(6) Dimensionality Reduction

Dimensionality Reduction is a technique that allows one to map **multidimensional data to a Key-Value model** or to other **non-multidimensional models.**
이 기법을 사용하면 다차원적인 데이터를 키-밸류 모델이나 다차원적이지 않은 모델에 매핑시킬 수 있다.

Traditional geographic information systems
use some variation of a Quadtree or R-Tree **for indexes.**
These structures need to be updated in-place
and are expensive to manipulate
when data volumes are large.
An alternative approach is to traverse the 2D structure and flatten it into a plain list of entries.
전통적인 지리정보시스템에서는 **index**용으로 Quadtree나 R-Tree를 변형한 것을 사용한다.
이런 구조는 최신으로 업데이트를 해줘야 하고, 데이터 규모가 클 때에 다루는 데에 비용이 많이 든다.
대안적인 접근법으로는 **2D** 구조를 가로질러서**(??)** 정보가 나열된 형태로 평평하게 **;;** 만드는 방법이 있다**.**

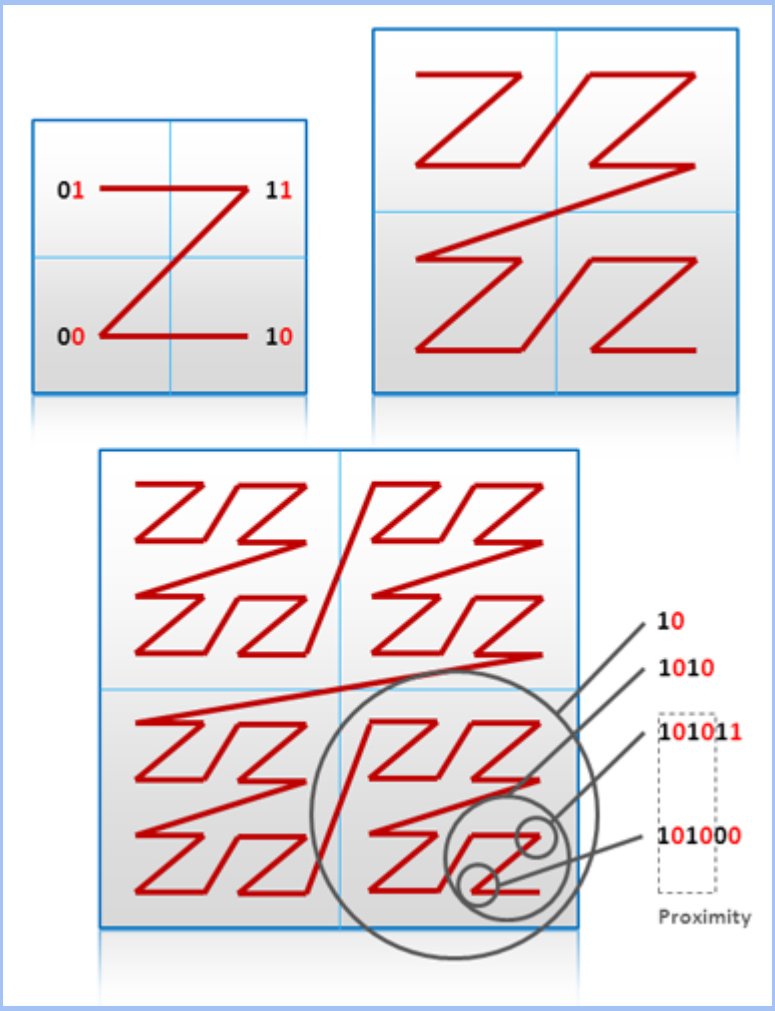
One well known example of this technique is a Geohash.
A Geohash uses a **Z-like scan** to **fill 2D space** and each move is encoded as 0 or 1 depending on direction.

이 기술의 가장 유명한 예제로는 Geohash가 있다.
Geohash는 2D 공간을 이동하며 Z축 검색을 사용하며, 각 이동시 방향에 따라 0 또는 1 으로 인코딩한다. (의역했습니다.)

Bits for longitude(경도) and latitude(위도) moves are interleaved as well as moves.

The encoding process is illustrated in the figure below, where black and red bits stand for longitude and latitude, respectively:

인코딩하는 과정은 아래의 그림에 나타나 있다. 여기서 검은 부분과 빨간 부분은 각각 경도와 위도를 의미한다 :



Geohash Index

An important feature of a Geohash is its **ability to estimate distance** between regions using **bit-wise code proximity,** as is shown in the figure.

Geohash의 중요한 특징은 bit-wise code proximity를 사용해서 영역 간의 거리를 측정하는 능력이 있다는 점이다. 그림에서 proximity 부분을 참조하라.

Geohash encoding **allows one to store geographical information** using **plain data models,** like **sorted key values** preserving **spatial relationships**. The Dimensionality Reduction technique for BigTable was described in [6.1]. M**ore information** about Geohashes and other related techniques can be found in [6.2] and [6.3].

Geohash 인코딩을 사용하면 정렬된 키 밸류 같은 평평한 데이터 모델(plain data model) 을 사용해서 지리 정보를 저장할 수 있다. 이 평평한 데이터 모델에는 장소에 관한 관계정보도 보존된다. BigTable용으로 쓰는 Dimensionality Reduction 기법은 [6.1]에서 설명하였다. Geohash에 대한 정보와 이와 관련된 기법을 보려면 [6.2]와 [6.3]을 참고하라.

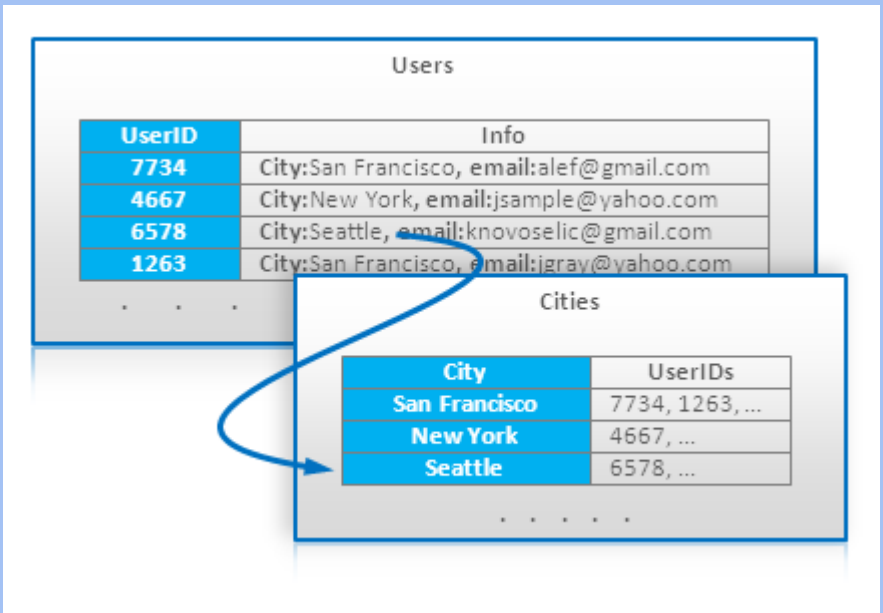
Applicability: Key-Value Stores, Document Databases, BigTable-style Databases

(7) Index Table

(7) 인덱스 테이블

Index Table is a very **straightforward technique** that allows one to take **advantage of indexes in stores** that do not **support indexes internally**. The most important **class of such stores** is the **BigTable-style database.** The idea is to create and maintain a **special table with keys** that follow the **access pattern.** For example, there is a **master table** that stores user accounts that can be accessed by user ID. A query that **retrieves all users** by a **specified city** can be supported by means of an additional table where **city is a key:**

인덱스 테이블은 매우 직선적인 기법이며, 이 기법을 사용하면 내부적으로 인덱스를 지원하지 않는 indexes in stores의 장점을 사용할 수 있다.그런 보존의 가장 중요한 클래스는 빅테이블형 데이터베이스이다. 원리는 이렇다. keys가 들어있는 특별한 테이블을 만들고 유지하는 것이다. 여기서 키(keys)는 엑세스 패턴(??)을 따른다. 예를 들어 유저 ID를 가지고 유저 계정에 접근할 수 있고, 유저 계정을 저장하는 마스터 테이블이 있다고 하자. 특정 도시 정보로 모든 유저를 찾아낼 때, 도시를 키로 하는 추가 테이블을 가지고 이 쿼리를 서포트할 수 있다.



Index Table Example

An Index table can be updated for each update of the master table or in batch mode. Either way, it results in an additional performance penalty and become a consistency issue.
Index Table can be considered as an analog of materialized views in relational

databases.

Applicability: BigTable-style Databases
적용 : 빅데이터-스타일 데이터베이스

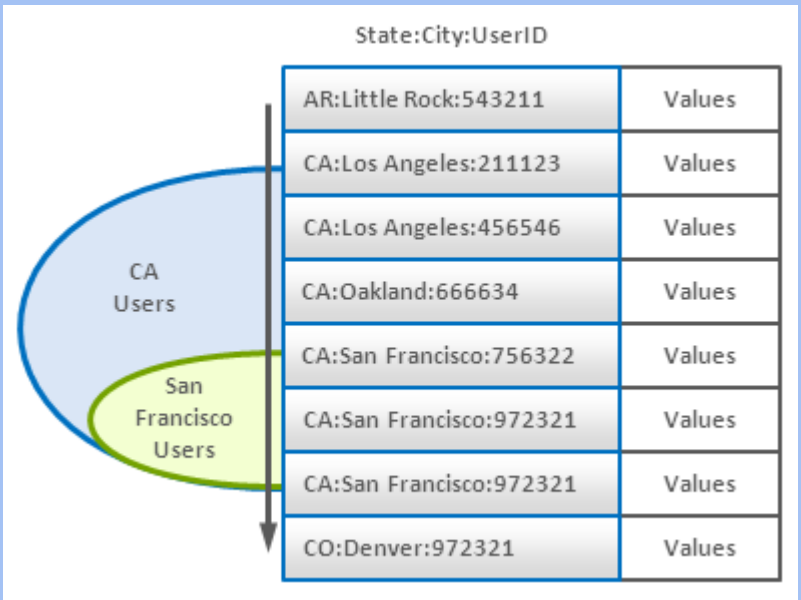
(8) Composite Key Index
복합 키 인덱스

Composite key is a very generic technique, but it is extremely beneficial when a store with ordered keys is used. Composite keys in conjunction with secondary sorting allows one to build a kind of multidimensional index which is fundamentally similar to the previously described Dimensionality Reduction technique. For example, let's take a set of records where each record is a user statistic. If we are going to aggregate these statistics by a region the user came from, we can use keys in a format *(State:City:UserID)* that allow us to iterate over records for a particular state or city if that store supports the selection of key ranges by a partial key match (as BigTable-style systems do):

복합 키는 매우 일반적인 기법이지만, 주문 키가 있는 저장소일 때 매우 유용다다. 보조 정렬과 결합된 복합 키는 앞에선 설명한 차원 감소 기법과 근본적으로 유사한 다차원 색인을 만들 수 있다. 예를 들면, 각 레코드가 사용자 통계인 레코드의 집합이 있다고 하자. 사용자들로부터 온 지역별 통계를 집계한다면, (State:City:UserID) 형식의 키를 사용할 수 있다. 저장소가 일부 키 매치(빅데이터-스타일 시스템과 같은)에 의한 키 범위의 선택을 지원한다면, 우리가 특정 주 또는 도시에 대한 기록을 통해 반복할 수 있도록 허용한다.

1	SELECT Values WHERE state="CA:*"

2	SELECT Values WHERE city="CA:San Francisco*"



Composite Key Index

Applicability: BigTable-style Databases

(9) Aggregation with Composite Keys

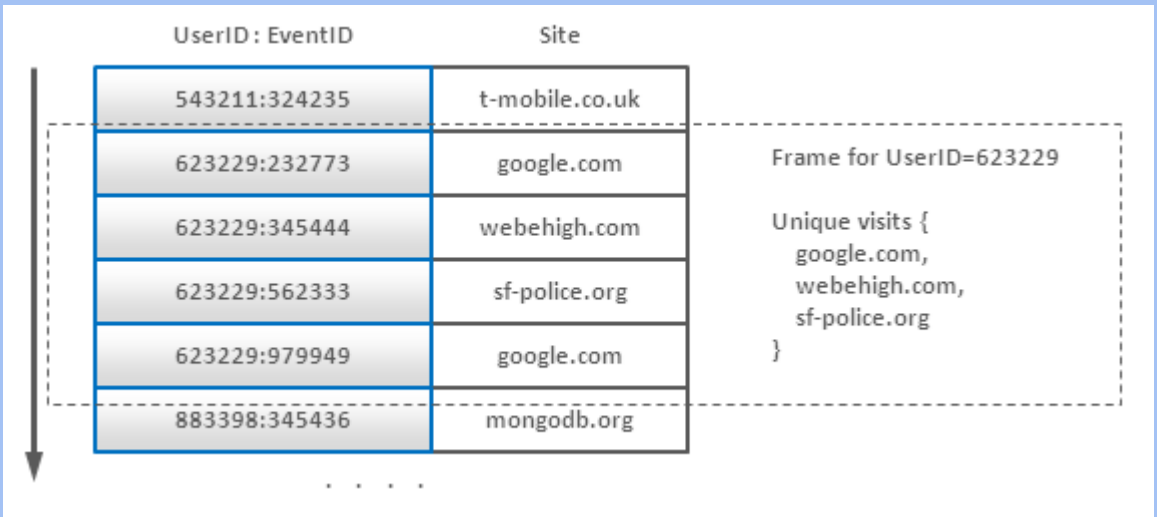
복합키를 사용한 집계

Composite keys may be used not only for indexing, but for different types of grouping. Let's consider an example. There is a huge array of log records with information about internet users and their visits from different sites (*click stream*). The goal is to count the number of unique users for each site. This is similar to the following SQL query:

복합키는 인덱싱뿐만 아니라 다른 종류의 그룹화를 하는데도 사용할수 있다. 예를 들어 보자. 인터넷 사용자와 그들의 다른 사이트로부터의 방문통계가 기록되어있는 거대한 배열의 로그가 있다고 하자. 목표는 각 사이트마다의 유니크 유저의 수를 찾아내는 것이다. 이 경우는 다음 쿼리와 비슷할 것이다.

```
1  SELECT count(distinct(user_id)) FROM
   clicks GROUP BY site
```

We can model this situation using composite keys with a UserID prefix:
이러한 경우를 UserID 접두사를 사용해서 복합키로 만들수 있다.



Counting Unique Users using Composite Keys

The idea is to keep all records for one user collocated, so it is possible to fetch such a frame into memory (one user can not produce too many events) and to eliminate site duplicates using hash table or whatever. An alternative technique is to have one entry for one user and append sites to this entry as events arrive. Nevertheless, entry modification is generally less efficient than entry insertion in the majority of implementations.
이것은 모든 레코드가 유저정보와 함께 저장될 수 있도록 합니다. 그리고 해쉬테이블이나 다른 방법을 사용해서 중복된 사이트 정보를 삭제한후 메모리에 적재하는것이 가능합니다. 한 유저가 한가지의 레코드를 가지게 하거나 사이트를 추가하는 또 다른 방법으로 이벤트가 완료때 하는 방법이 있습니다. 하지만 속성 수정은 일반적으로 대부분의 구현에서 속성삽입 보다는 비효율적입니다.

Applicability: Ordered Key-Value Stores, BigTable-style Databases

(10) Inverted Search – Direct Aggregation

역검색 - 직접 집계

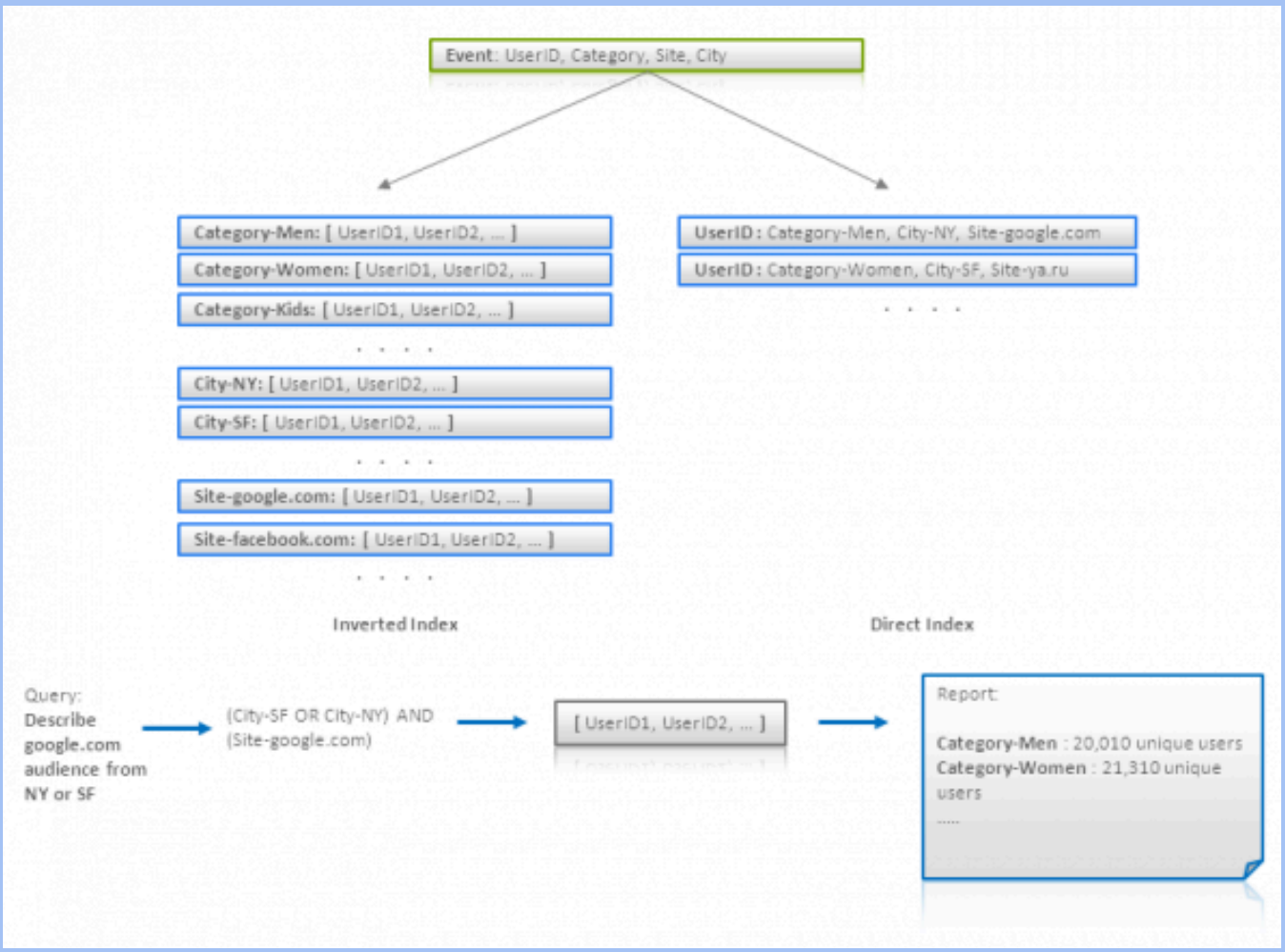
This technique is more a data processing pattern, rather than data modeling. Nevertheless, data models are also impacted by usage of this pattern. The main idea of this technique is to use an index to find data that meets a criteria, but aggregate data using original representation or full scans. Let's consider an example. There are a number of log records with information about internet users and their visits from different sites (*click stream*). Let assume that each record contains user ID, categories this user belongs to (Men, Women, Bloggers, etc), city this user came from, and visited site. The goal is to describe the audience that meet some criteria (site, city, etc) in terms of unique users for each category that occurs in this audience (i.e. in the set of users that meet the criteria).
It is quite clear that a search of users that meet the criteria can be efficiently done

using inverted indexes like *{Category -> [user IDs]}* or *{Site -> [user IDs]}*. Using such indexes, one can intersect or unify corresponding user IDs (this can be done very efficiently if user IDs are stored as sorted lists or bit sets) and obtain an audience. But describing an audience which is similar to an aggregation query like

이 기술은 데이터 모델링보다는 데이터 프로세싱 패턴이다. 하지만 데이터모델 또한 이 패턴의 영향을 받는다. 이 기술의 핵심은 검색조건에 맞는 데이터를 찾는데 인덱스를 이용하는 것이다.

```
1  SELECT count(distinct(user_id)) ...
   GROUP BY category
```

cannot be handled efficiently using an inverted index if the number of categories is big. To cope with this, one can build a direct index of the form *{UserID -> [Categories]}* and iterate over it in order to build a final report. This schema is depicted below:



Counting Unique Users using Inverse and Direct Indexes

And as a final note, we should take into account that random retrieval of records for each user ID in the audience can be inefficient. One can grapple with this problem by leveraging batch query processing. This means that some number of user sets can be precomputed (for different criteria) and then all reports for this batch of audiences can be computed in one full scan of direct or inverse index.

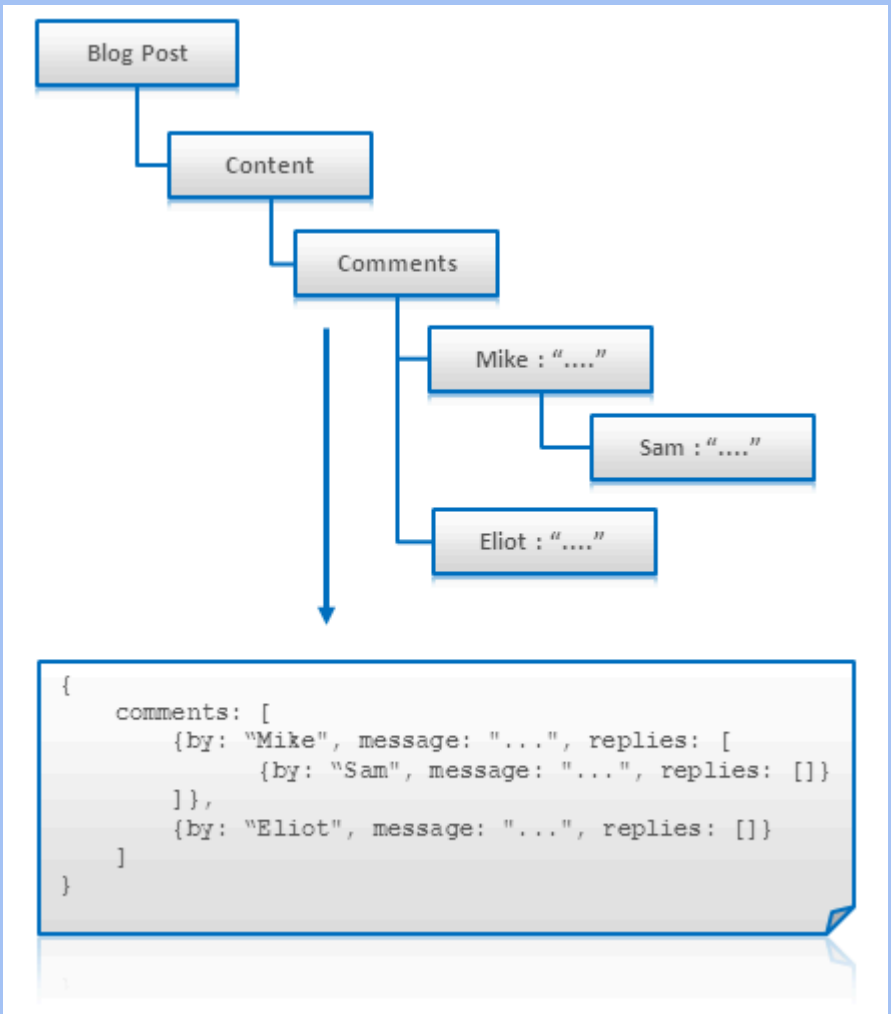
Applicability: Key-Value Stores, BigTable-style Databases, Document Databases

Hierarchy Modeling Techniques

(11) Tree Aggregation

Trees or even arbitrary graphs (with the aid of denormalization) can be modeled as a single record or document.

- This techniques is efficient when the tree is accessed at once (for example, an entire tree of blog comments is fetched to show a page with a post).
- Search and arbitrary access to the entries may be problematic.
- Updates are inefficient in most NoSQL implementations (as compared to independent nodes).



Tree Aggregation

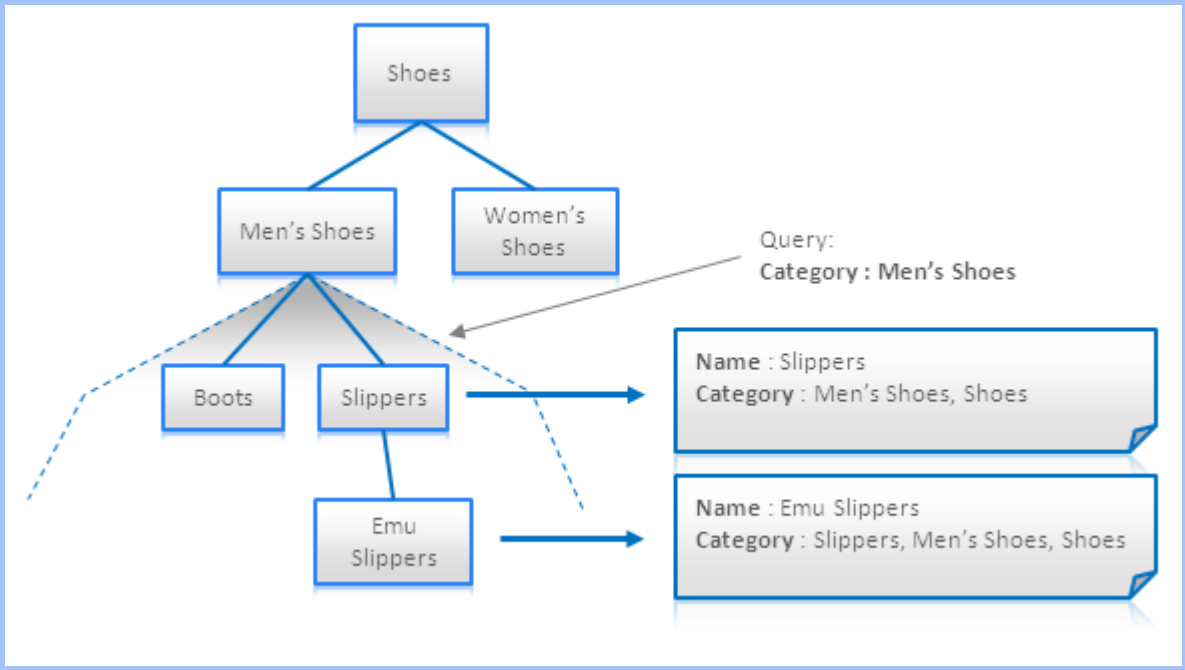
Applicability: Key-Value Stores, Document Databases

(12) Adjacency Lists

Adjacency Lists are a straightforward way of graph modeling – each node is modeled as an independent record that contains arrays of direct ancestors or descendants. It allows one to search for nodes by identifiers of their parents or children and, of course, to traverse a graph by doing one hop per query. This approach is usually inefficient for getting an entire subtree for a given node, for deep or wide traversals.
Applicability: Key-Value Stores, Document Databases

(13) Materialized Paths

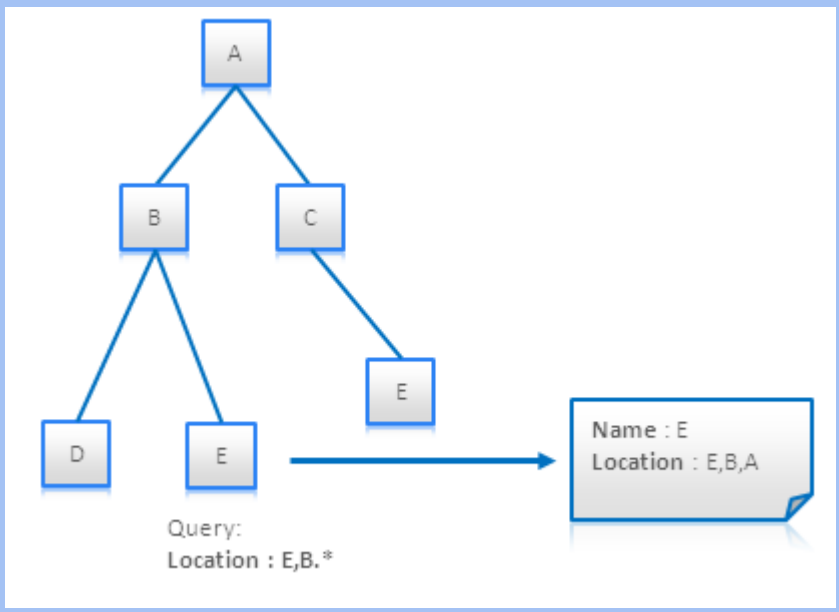
Materialized Paths is a technique that helps to avoid recursive traversals of tree-like structures. This technique can be considered as a kind of denormalization. The idea is to attribute each node by identifiers of all its parents or children, so that it is possible to determine all descendants or predecessors of the node without traversal:



Materialized Paths for eShop Category Hierarchy

This technique is especially helpful for Full Text Search Engines because it allows one to convert hierarchical structures into flat documents. One can see in the figure above that all products or subcategories within the *Men's Shoes* category can be retrieved using a short query which is simply a category name.

Materialized Paths can be stored as a set of IDs or as a single string of concatenated IDs. The latter option allows one to search for nodes that meet a certain partial path criteria using regular expressions. This option is illustrated in the figure below (path includes node itself):

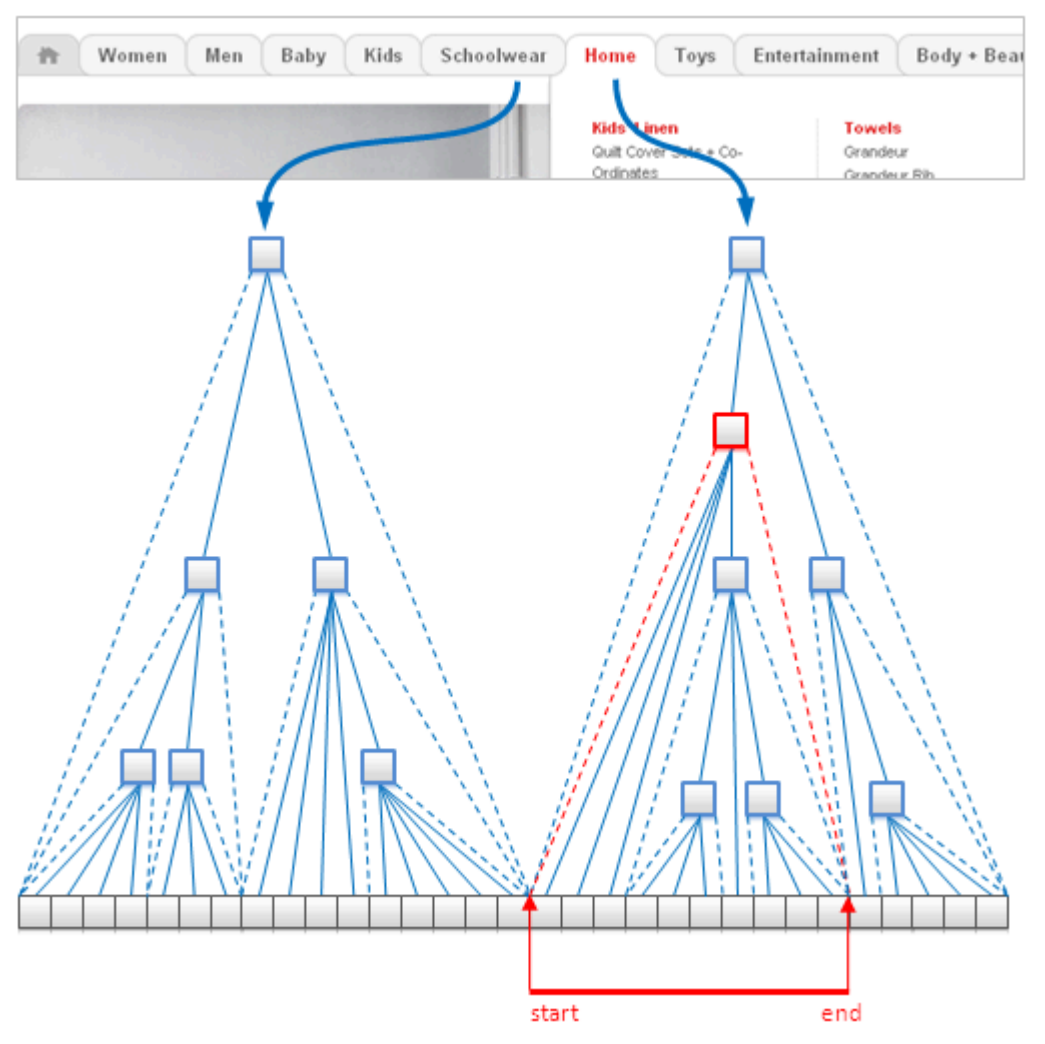


Query Materialized Paths using RegExp

Applicability: Key-Value Stores, Document Databases, Search Engines

(14) Nested Sets

Nested sets is a standard technique for modeling tree-like structures. It is widely used in relational databases, but it is perfectly applicable to Key-Value Stores and Document Databases. The idea is to store the leafs of the tree in an array and to map each non-leaf node to a range of leafs using start and end indexes, as is shown in the figure below:



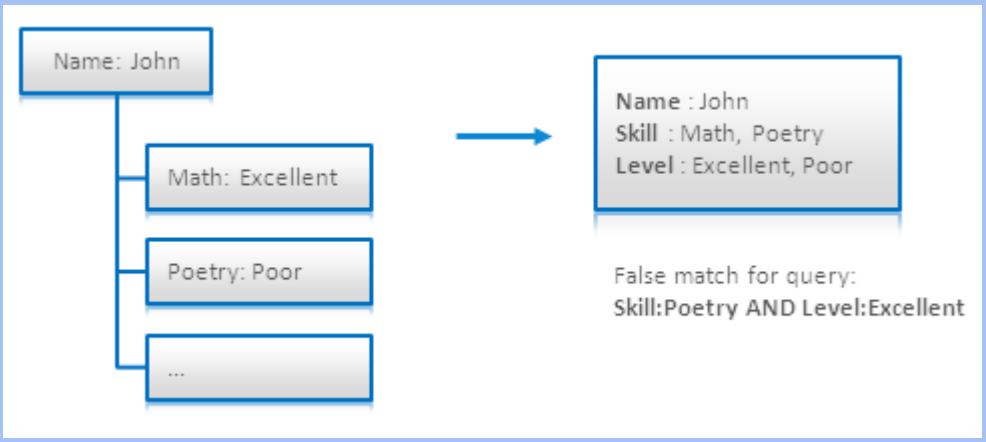
Modeling of eCommerce Catalog using Nested Sets

This structure is pretty efficient for immutable data because it has a small memory footprint and allows one to fetch all leafs for a given node without traversals. Nevertheless, inserts and updates are quite costly because the addition of one leaf causes an extensive update of indexes.

Applicability: Key-Value Stores, Document Databases

(15) Nested Documents Flattening: Numbered Field Names

Search Engines typically work with flat documents, i.e. each document is a flat list of fields and values. The goal of data modeling is to map business entities to plain documents and this can be challenging if the entities have a complex internal structure. One typical challenge mapping documents with a hierarchical structure, i.e. documents with nested documents inside. Let's consider the following example:

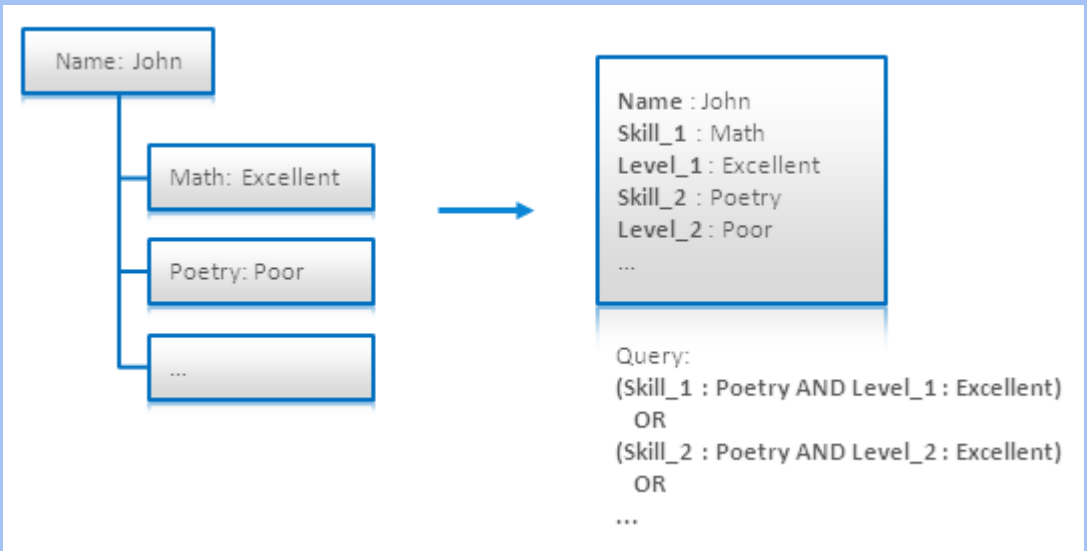


Nested Documents Problem

Each business entity is some kind of resume. It contains a person's name and a list of his or her skills with a **skill level.** An obvious way to model such an **entity** is to create a **plain document** with*Skill* and *Level* **fields.** This model allows one to **search for a person** by skill or by level, but queries that combine both fields are liable to r**esult in false matches**, as depicted in the figure above.

비즈니스 엔티티 각각은 일종의 이력서라고 할 수 있다. 각 비즈니스 엔티티에는 한 사람의 이름과, 그 사람이 가진 기술 목록, 각 기술의 수준이 나와 있다. 이런 엔티티를 모델링하는 명확한 방법은 Skill과 Level 필드가 들어있는 plain document를 만드는 것이다. 이렇게 모델링하면 개별 Skill이나 Level로는 해당하는 사람을 찾을 수 있지만, SKill과 Level 필드를 둘다 포함하는 쿼리를 날리면, 위 그림에서 보여주듯 잘못된 결과가 나온다.

One way to overcome this issue was suggested in [4.6]. The **main idea of this technique** is t**o index each skill and corresponding level** as a **dedicated pair** of fields *Skill_i* and *Level_i,* and **to search for all these pairs** simultaneously (where the number of **OR-ed terms** in a query is as high as the **maximum number of skills** for one person):



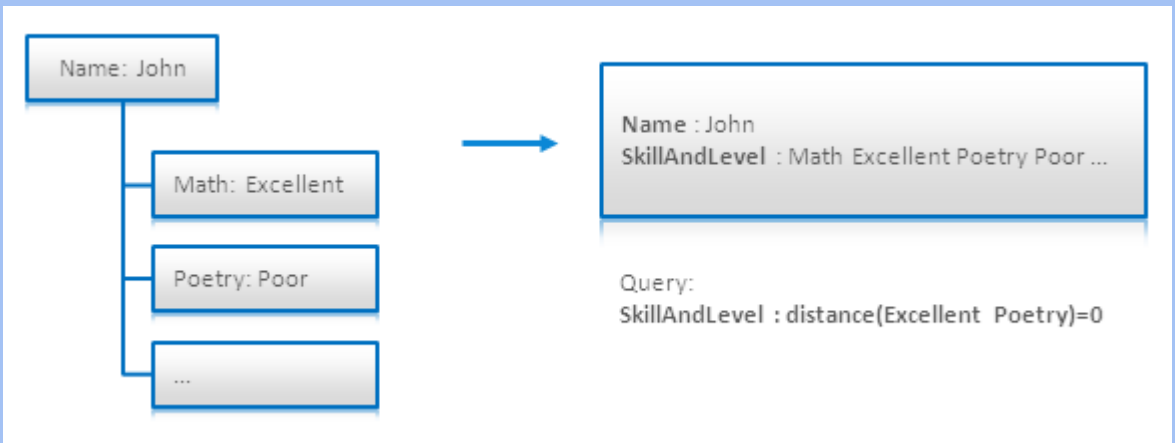
Nested Document Modeling using Numbered Field Names

This approach is not really scalable because query complexity grows rapidly as a function of the number of nested structures.

Applicability: Search Engines

(16) Nested Documents Flattening: Proximity Queries

The problem with nested documents can be solved using another technique that were also described in [4.6]. The idea is to use proximity queries that limit the acceptable distance between words in the document. In the figure below, all skills and levels are indexed in one field, namely, SkillAndLevel, and the query indicates that the words "Excellent" and "Poetry" should follow one another:



Nested Document Modeling using Proximity Queries

[4.3] describes a success story for this technique used on top of Solr.

Applicability: Search Engines

(17) Batch Graph Processing

Graph databases like neo4j are exceptionally good for exploring the neighborhood of a given node or exploring relationships between two or a few nodes. Nevertheless, global processing of large graphs is not very efficient because general purpose graph databases do not scale well. Distributed graph processing can be done using MapReduce and the Message Passing pattern that was described, for example, in one of my previous articles. This approach makes Key-Value stores, Document databases, and BigTable-style databases suitable for processing large graphs.

Applicability: Key-Value Stores, Document Databases, BigTable-style Databases

References

Finally, I provide a list of useful links related to NoSQL data modeling:
마지막으로 NoSql 데이터 모델링에 관련된 쓸만한 링크를 공유한다.

-