NoSQL Data Modeling Techniques

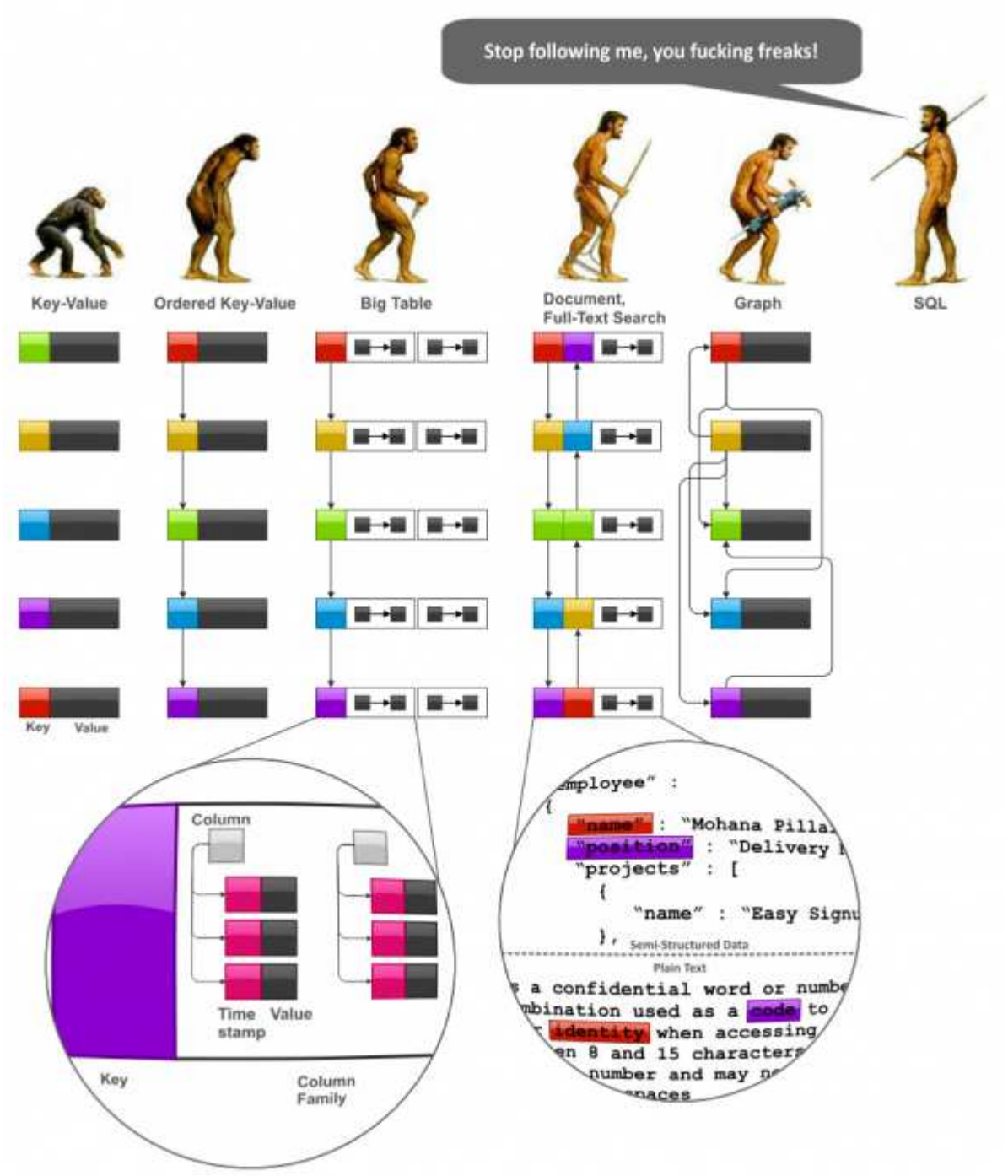
Posted on March 1, 2012

35

NoSQL databases are often compared by various non-functional criteria, such as scalability, performance, and consistency. 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. At the same time, NoSQL data modeling is not so well studied and lacks the systematic theory found in relational databases. 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.

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

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. 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:



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:

- 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.

- No one can expect human users to explicitly control concurrency, integrity, consistency, or data type validity. That's why SQL pays a lot of attention to transactional guaranties, schemas, and referential integrity.

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. Besides this, elimination of these features had an extremely important influence on the performance and scalability of the stores. And this was where a new evolution of data models began:

- Key-Value storage is a very simplistic, but very powerful model. Many techniques that are described below are perfectly applicable to this model.
- 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.
- Ordered Key-Value model is very powerful, but it does not provide 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.
- 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 Text 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 field names, as opposed to Search Engines that group indexes by field values. 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.
- 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.

General Notes on NoSQL Data Modeling

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:

- NoSQL data modeling often starts from the application-specific queries as opposed to relational modeling:
    - Relational modeling is typically driven by the structure of available data. The main design theme is "**What answers do I have?"**

    - 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?”**
- 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.
- Data duplication and denormalization are first-class citizens.
- 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.

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:

- 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

Conceptual Techniques

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

(1) Denormalization

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.

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

- *Query data volume* or *IO per query* VS *total data volume*. Using denormalization one can group all data that is needed to process a query in one place. This often means that for different query flows the same data will be accessed in different combinations. Hence we need to duplicate data, which increases total data volume.
- *Processing complexity* VS *total data volume*. 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.

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

(2) Aggregates

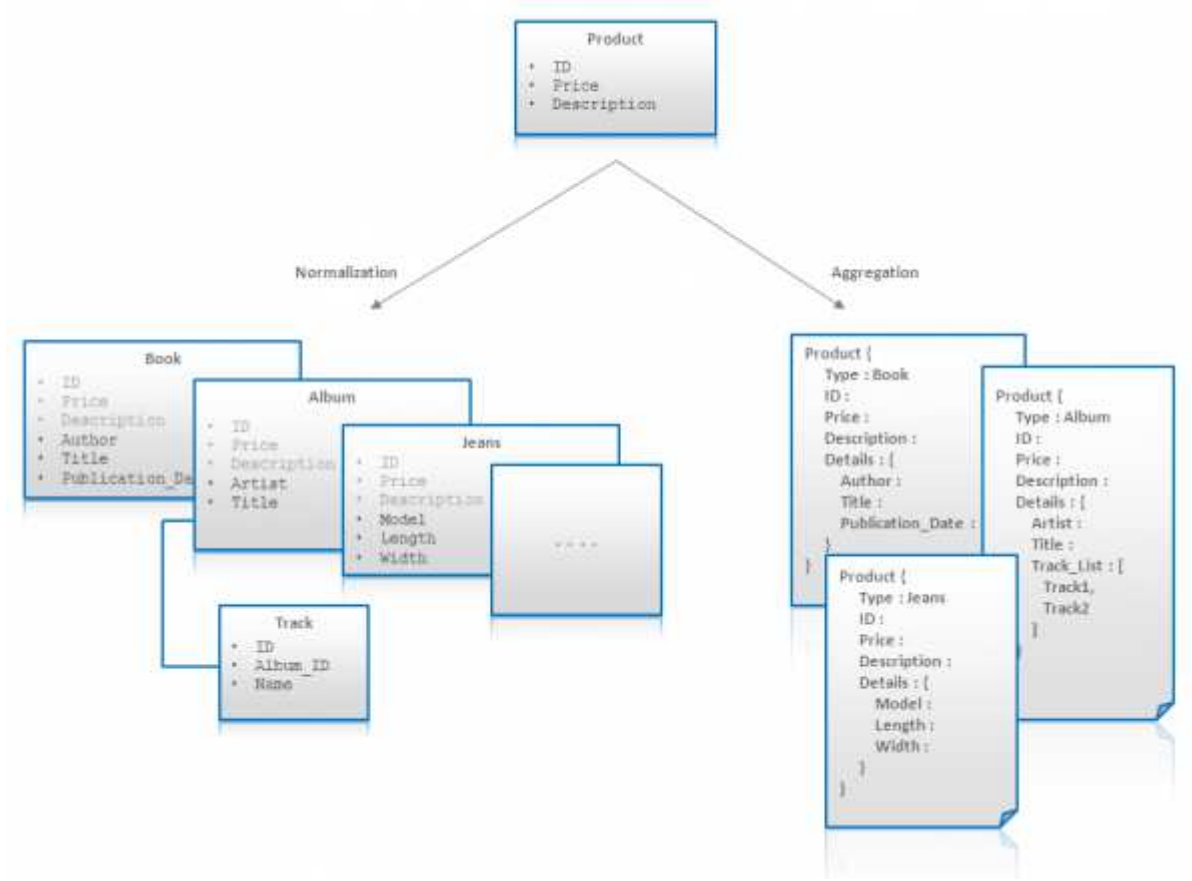
All major genres of NoSQL provide soft schema capabilities in one way or another:

- 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.
- BigTable models support 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.

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:

- Minimization of one-to-many relationships by means of nested entities and, consequently, reduction of joins.
- 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 different types of products have different attributes like Author for Book or Length for Jeans. Some of these attributes have a one-to-many or many-to-many nature like Tracks in Music Albums. Next, it is possible that some entities can not be modeled using fixed types 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 model, but solutions are far from elegant. Soft schema allows one to use a single Aggregate (product) that can model all types of products and their attributes:



Entity Aggregation

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

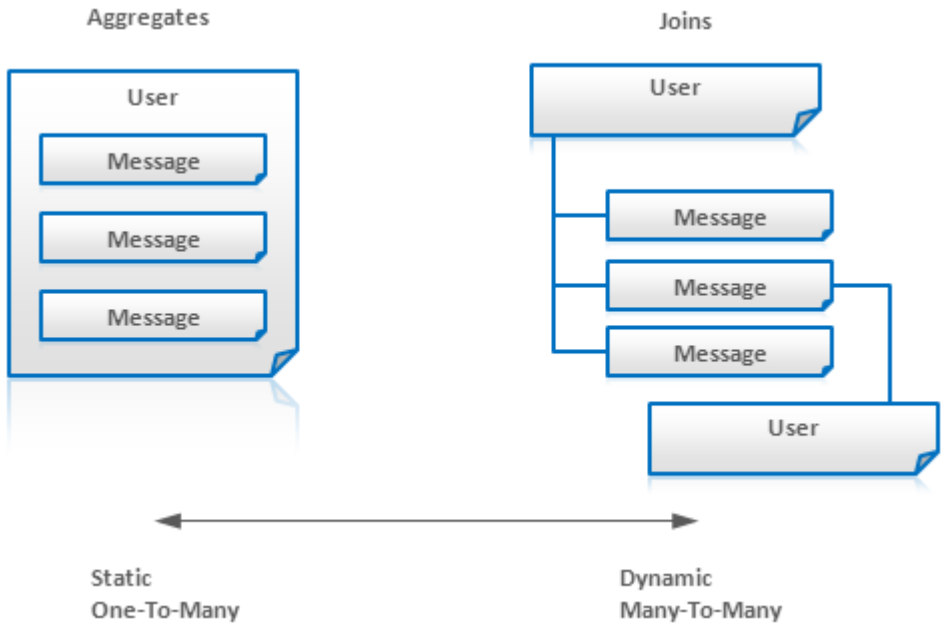
Applicability: Key-Value Stores, Document Databases, BigTable-style Databases

(3) Application Side Joins

Joins are rarely supported in NoSQL solutions. 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. Query time joins almost always mean a performance penalty, 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:

- Many to many relationships are often modeled by links and require joins.
- 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:



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

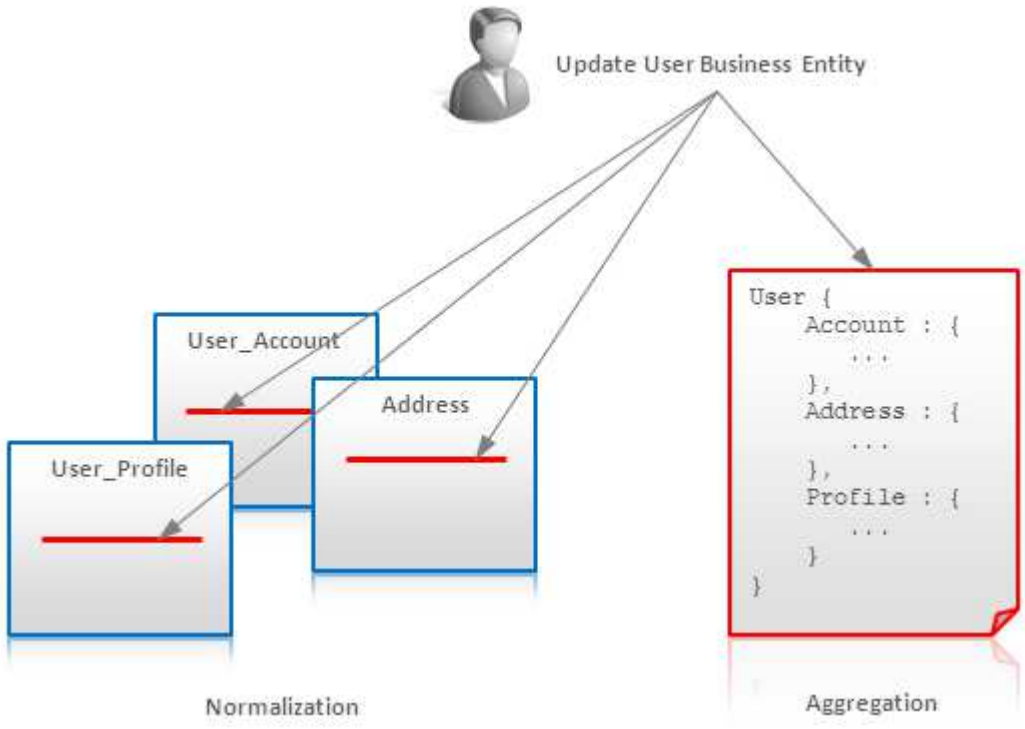
General Modeling Techniques

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

(4) Atomic Aggregates

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.

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:



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.

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

(5) Enumerable Keys

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:

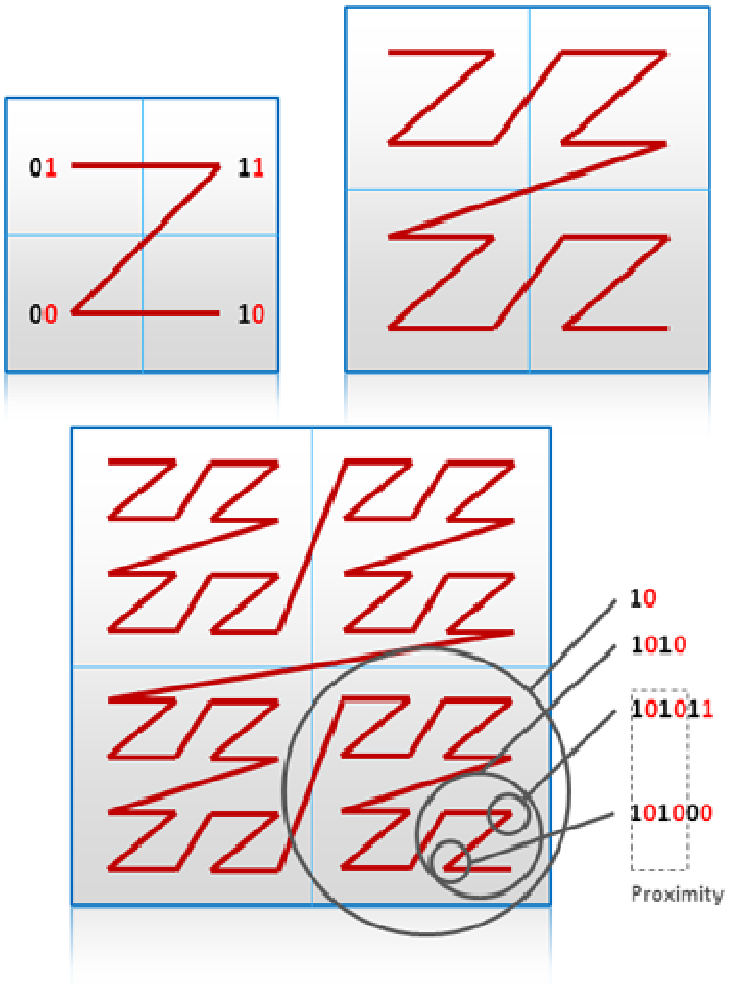
1. Some NoSQL stores provide atomic counters that allow one to generate sequential IDs. In this case one can store messages using *userID_messageID* as a composite key. If the latest message ID is known, it is possible to traverse previous messages. It is also possible to traverse preceding and succeeding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

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.

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. 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. Bits for longitude and latitude 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:



Geohash Index

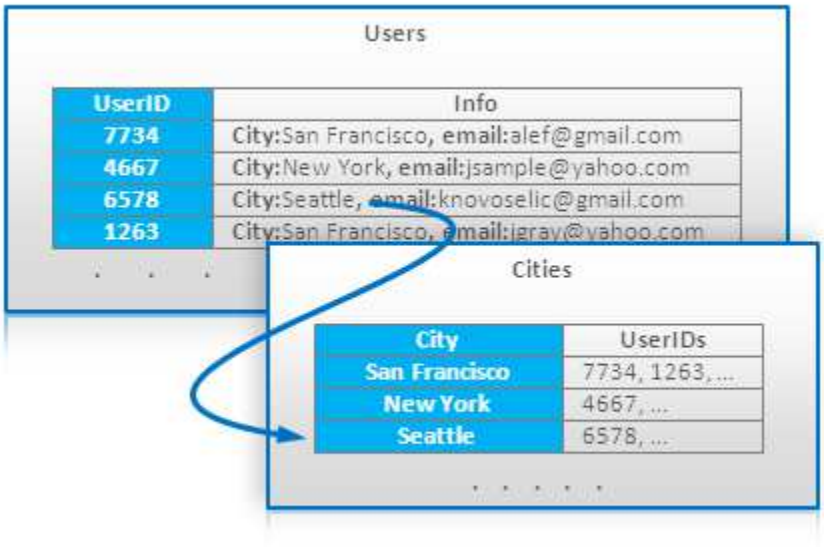
An important feature of a Geohash is its ability to estimate distance between regions using bit-wise code proximity, as is shown in the figure. Geohash encoding allows one to store

geographical information using plain data models, like sorted key values preserving spatial relationships. The Dimensionality Reduction technique for BigTable was described in [6.1]. More information about Geohashes and other related techniques can be found in [6.2] and [6.3].

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

(7) Index Table

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:



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

(8) Composite Key Index

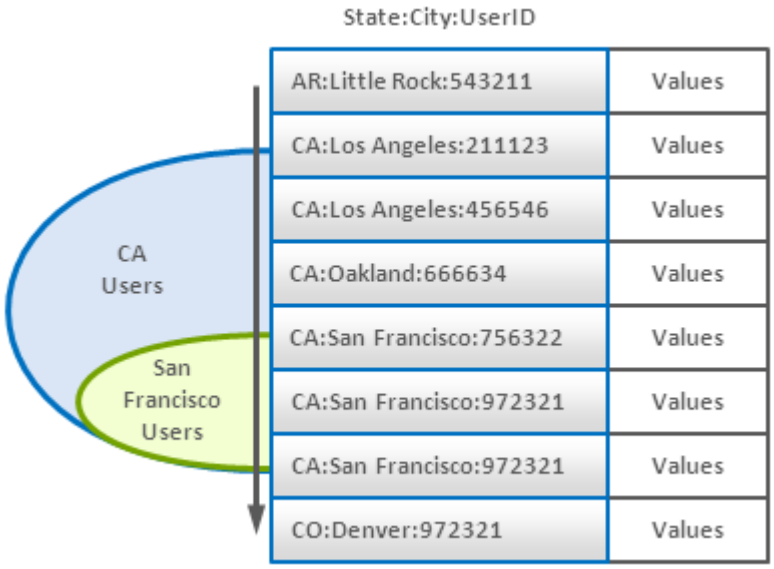
Composite key is a very generic technique, but it is extremely beneficial when a store with ordered keys is used. Composite keys in conjunction with secondary sorting allows one to build a kind of multidimensional index which is fundamentally similar to the previously described

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):

view source

print?

```
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

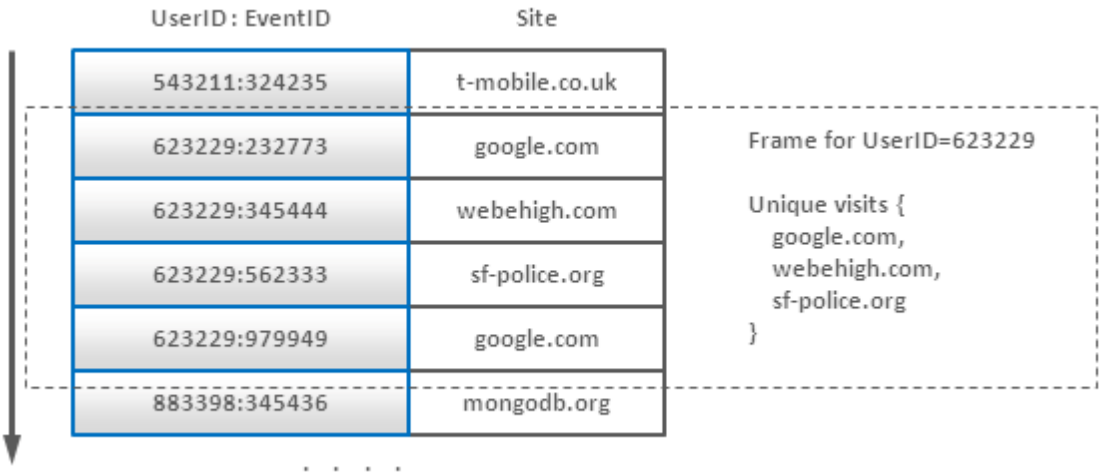
Composite keys may be used not only for indexing, but for different types of grouping. Let's consider an example. There is a huge array of log records with information about internet users and their visits from different sites (*click stream*). The goal is to count the number of unique users for each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

view source

print?

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

We can model this situation using composite keys with a UserID prefix:



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.

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

(10) Inverted Search – Direct Aggregation

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).

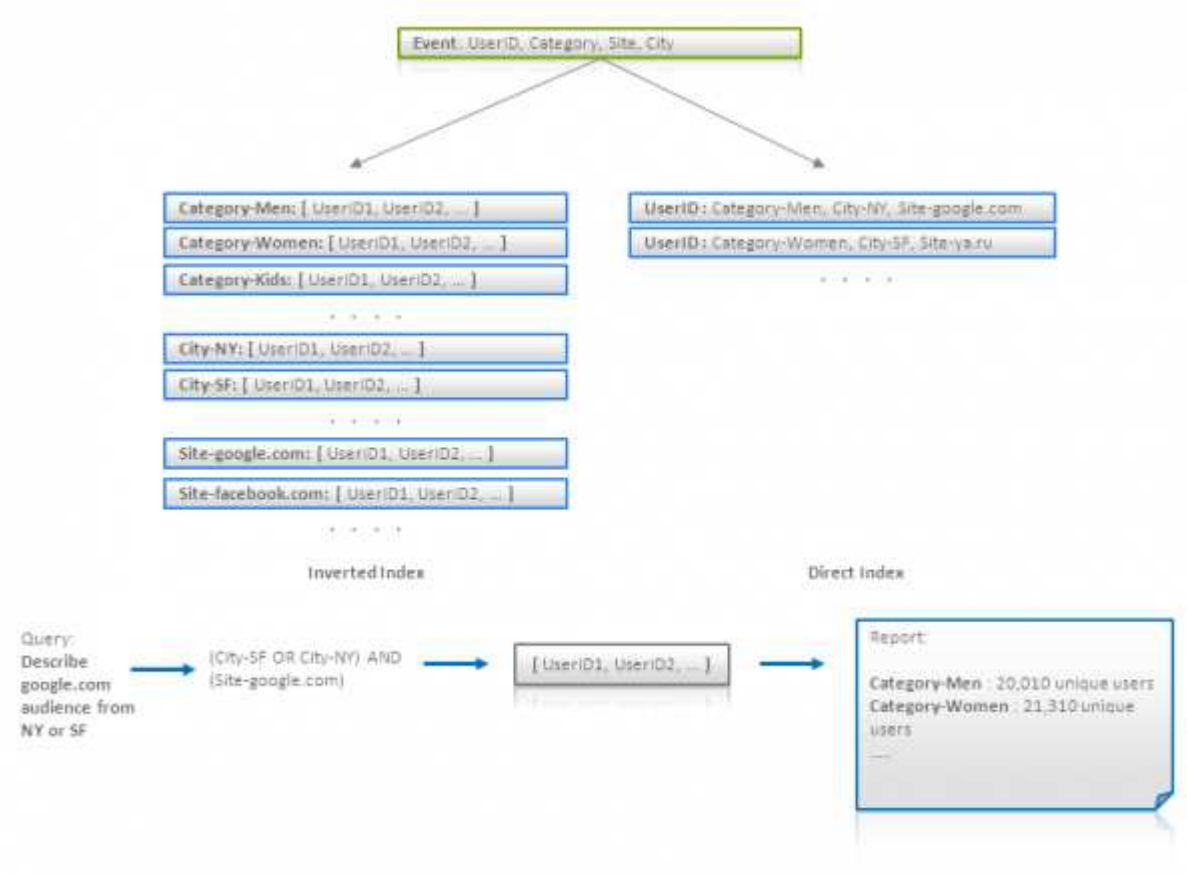
It is quite clear that a search of users that meet the criteria can be efficiently done using inverted indexes like *{Category -> [user IDs]}* or *{Site -> [user IDs]}*. Using such indexes, one can intersect or unify corresponding user IDs (this can be done very efficiently if user IDs are stored as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

view source

print?

```
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.

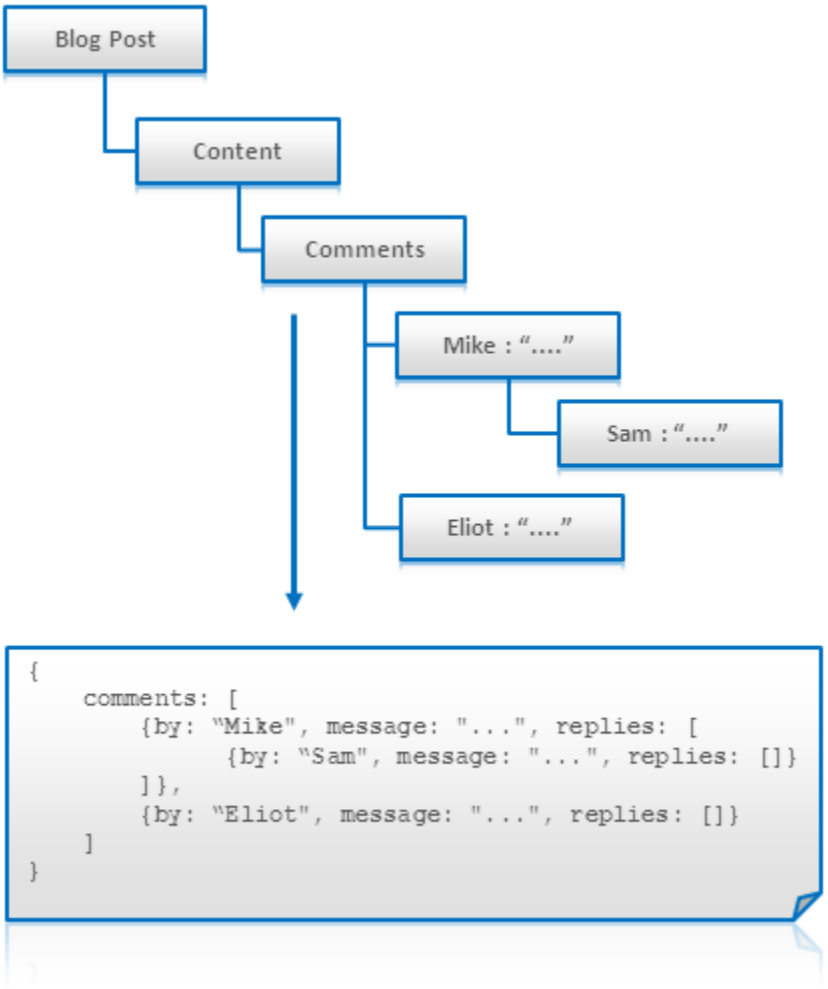
Applicability: Key-Value Stores, BigTable-style Databases, Document Databases

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

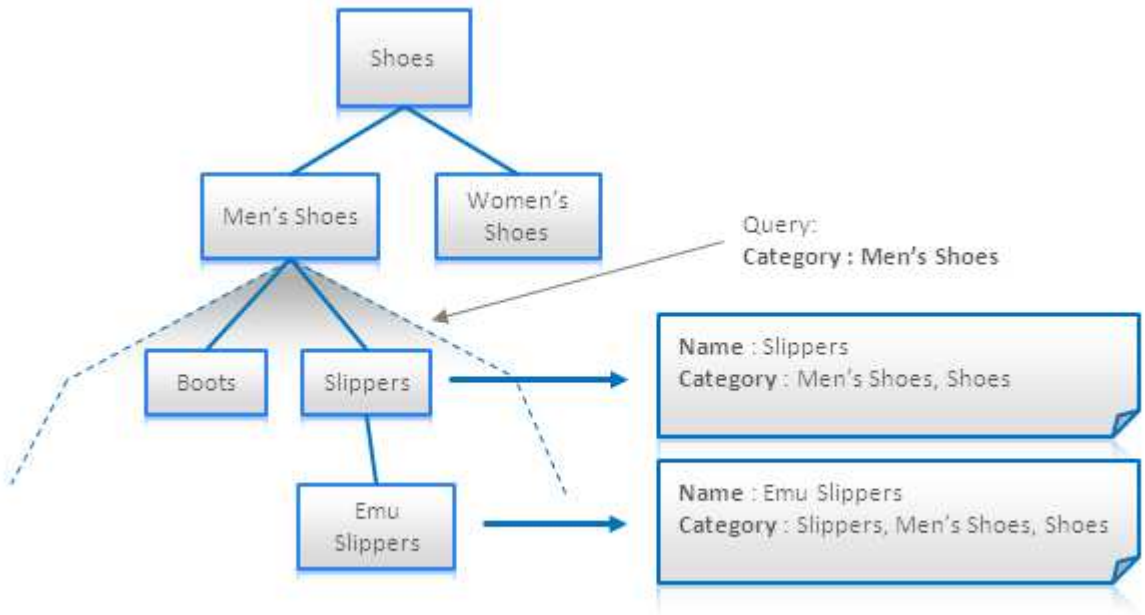
(12) Adjacency Lists

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

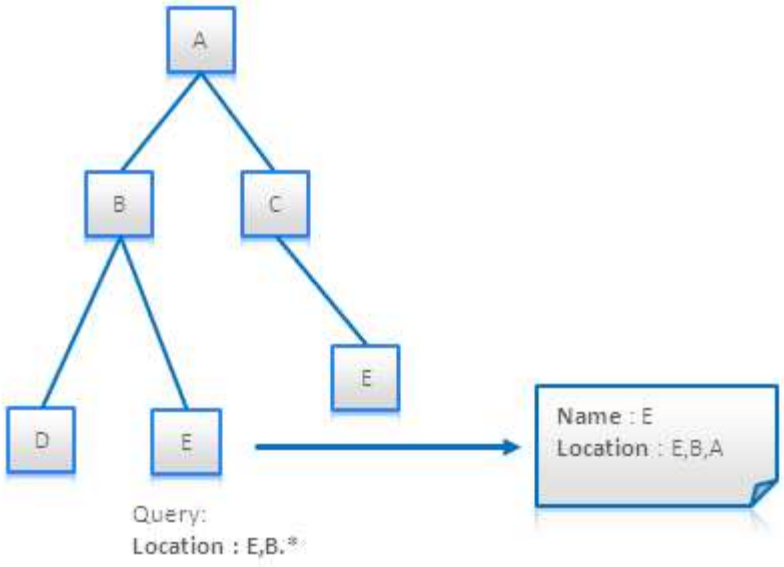
Materialized Paths is a technique that helps to avoid recursive traversals of tree-like structures. This technique can be considered as a kind of denormalization. The idea is to attribute each node by identifiers of all its parents or children, so that it is possible to determine all descendants or predecessors 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):

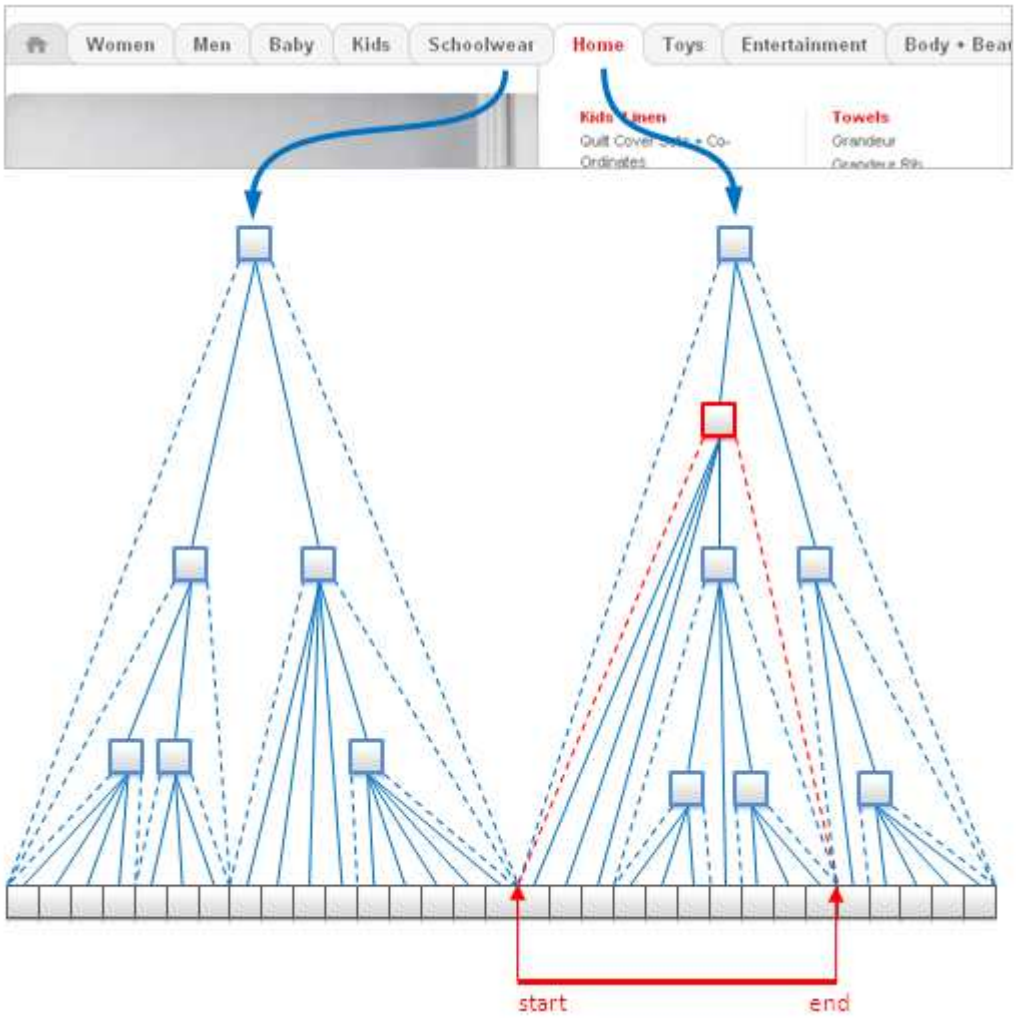


Query Materialized Paths using RegExp

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:



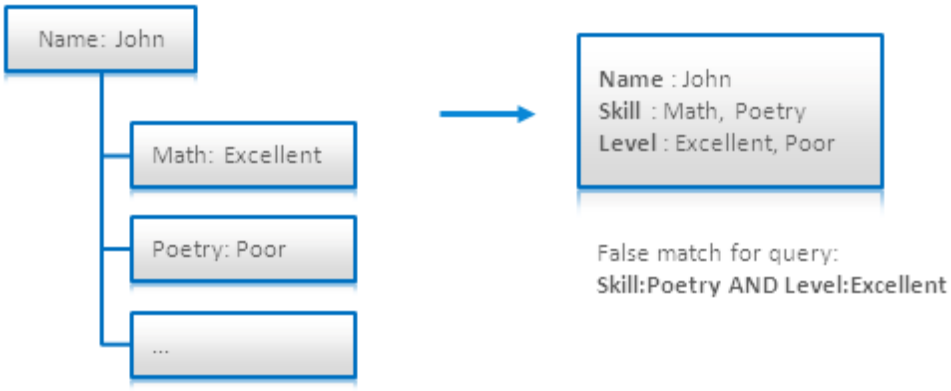
Modeling of eCommerce Catalog using Nested Sets

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

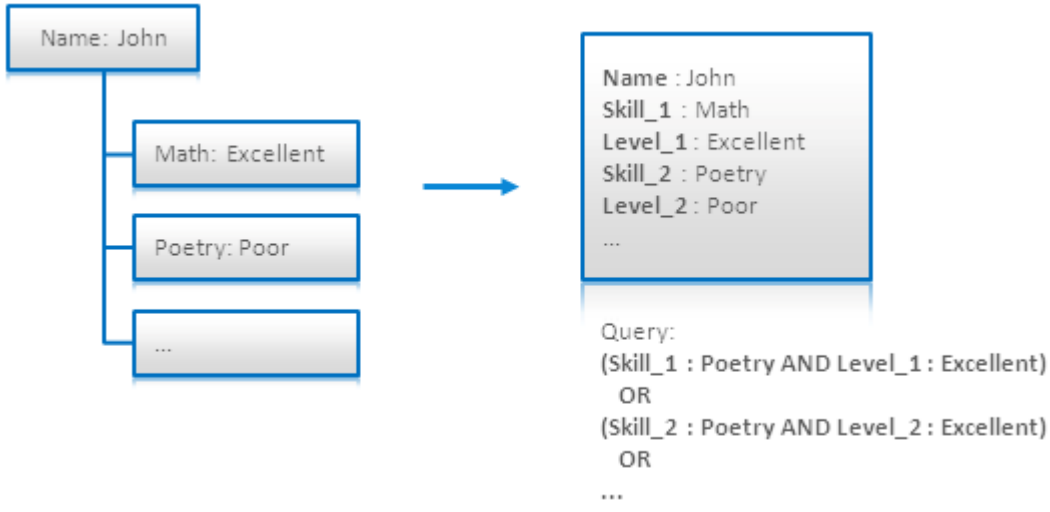
Search Engines typically work with flat documents, i.e. each document is a flat list of fields and values. The goal of data modeling is to map business entities to plain documents and this can be challenging if the entities have a complex internal structure. One typical challenge mapping documents with 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 result in false matches, as depicted in the figure above.

One way to overcome this issue was suggested in [4.6]. The main idea of this technique is to 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):



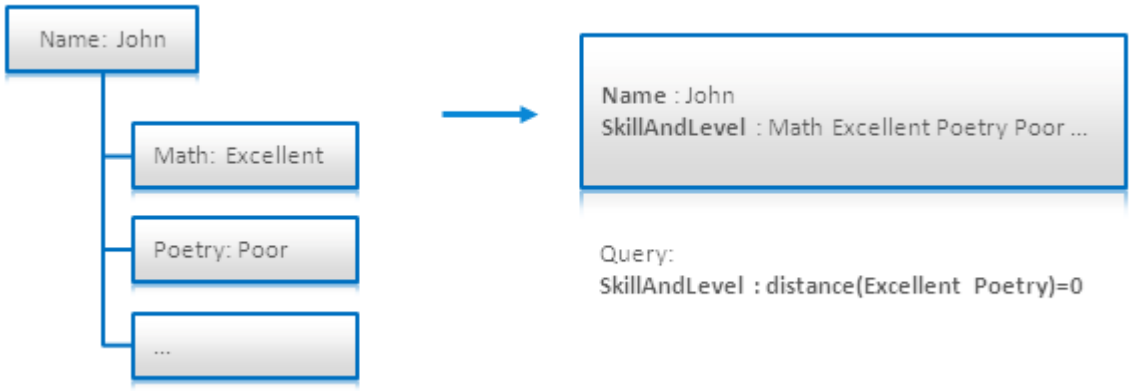
Nested Document Modeling using Numbered Field Names

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