# Modelling data in a schema-free world?

FrOSCon
2012-08-26

Jan Steemann (triAGENS)

# Schemas in relational databases

- Relational databases are around for a long time already
- In most relational databases we can deal with these schema elements:
  - tables
  - columns / fields
  - data types
  - indexes
  - relationships
  - views
  - triggers
  - procedures / functions
- Schema elements are auxiliary means to organise and validate the actual data!

# Normalisation in relational databases

- Relational databases like your data to be in a highly normalized and de-duplicated form
- Put simple, normalisation means
  - dividing big entities (tables, fields) into smaller ones
  - defining relationships between them
- Example data, not normalised:

| user_id | user_name | group_name | status |
|---|---|---|---|
| 1 | foo | Power user | Active |
| 2 | bar | Rookie | Inactive |
| 3 | baz | Rookie | Active |

# Normalisation in relational databases

- Example data, normalised (relationships not depicted):

**Table „users“**

| id | name | group | status |
|---|---|---|---|
| 1 | foo | 1 | 1 |
| 2 | bar | 2 | 2 |
| 3 | baz | 2 | 1 |

**Table „groups“**

| id | name |
|---|---|
| 1 | Power user |
| 2 | Rookie |

**Table „status“**

| id | name |
|---|---|
| 1 | Active |
| 2 | Inactive |

- Normalisation helps to
  - reduce redundancies and storage requirements
  - increase consistency and integrity
- Normalised data is re-arranged on retrieval using joins, projections etc.
- Due to normalisation, data may be fetched from several tables

# Issues with relational databases

- Issues of relational databases:
  - scaling is hard:
    This is due to the ACID (atomicity, consistency, isolation, durability) guarantees they usually provide.
  - rigid database schemas can obstruct rapid application development:
    Initial and ongoing database schema development and evolution takes time, but you must do it in order to have your database accept any data at all, designing good schemas is hard
  - fixed database schemas block deployment of changes:
    When deploying application changes, the database schema needs to stay in sync with the application. Deploying application changes may even cause downtime due to long-running database migrations (e.g. ALTER TABLE statements)

# Issues with relational databases

- Some more issues:
  - static schemas do not play well with dynamically typed languages:
    many languages support dynamic keys for their container types, dynamic variables, run-time typing etc. To use this with a relational database, one needs workarounds (e.g. entity-attribute value modelling, auxilliary tables) that increase complexity and make it slow
  - saving multi-valued data is „forbidden“:
    Mutli-value containers (e.g. hashes, dictionaries, lists, arrays) are used frequently in programming but must not be saved in the database as such
    Complex workarounds are necessary to handle that
  - saving objects can be expensive:
    saving one object in a relational way may create a lot of noise in relational databases (though this may be obscured by frameworks)

# NoSQL databases – why?

- In the past 5 years, many new database products have emerged
- The common term for these new products is „NoSQL“ databases, mainly interpreted as „not only SQL“
- „non-relational“ is also a good name for them because almost no NoSQL database follows the relational model
- Instead, most NoSQL databases overcome specific issues that relational databases have, especially
  - scaling
  - schema rigidity

# NoSQL databases – scaling

- Scaling problems of relational databases are addressed in most NoSQL databases by reducing ACID guarantees and removing relational features:
  - Atomicity is guaranteed only for some operations, not all
  - Isolation is only guaranteed for some operations
  - Strict consistency is relaxed to eventual consistency
  - Strict durability is normally turned off (but is configurable by admin)
  - Don't offer referential integrity or do not enforce it
  - Complex data operations as joins etc. are highly restricted or completely disallowed
- These measures help to reduce locking and synchronisation overhead and make scaling possible at all

# NoSQL databases – schema rigidity

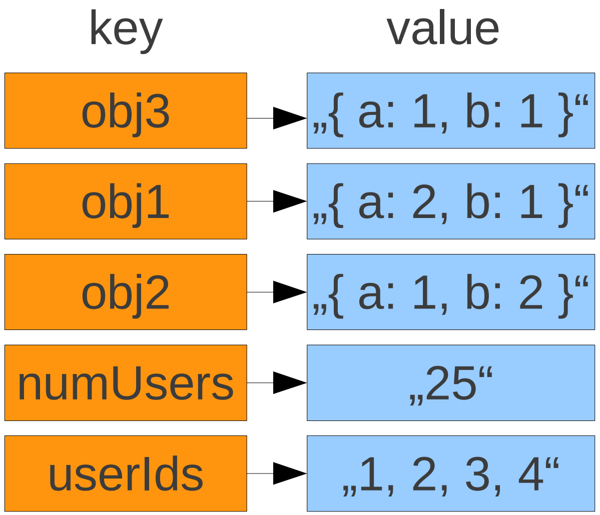
- Most NoSQL databases lift the requirement of having to define explicit database schemas:
  - Some NoSQL databases don't use schemas at all, they are fully schema-free
  - Some require the user to only predefine highest level schema elements (e.g. databases, collections, column families) upfront (and some allow it as you go)
  - Some use schemas implicitly by analysing incoming data, using dynamic schemas for each data element inserted instead of predefined schemas
- Benefits:
  - Less time spent on upfront schema development and ongoing schema evolution, more time can be used for actual application development
  - Can store and retrieve programming language objects more easily
  - No need to sync application changes deployment with database schema changes, no database downtime required for schema modifications

# NoSQL databases – overview

- There is a multitude of different NoSQL databases around
- http://nosql-databases.org/ currently lists 122 NoSQL databases
- Some of the most popular ones:
  - MongoDB
  - CouchDB
  - RavenDB
  - Hadoop
  - Hypertable
  - Cassandra
  - SimpleDB
  - Riak
  - Redis
  - Tokyo Cabinet
  - Voldemort
  - Neo4j
  - OrientDB
  - InfiniteGraph
- They are all different, but a basic categorisation follows

# NoSQL databases – categorisation

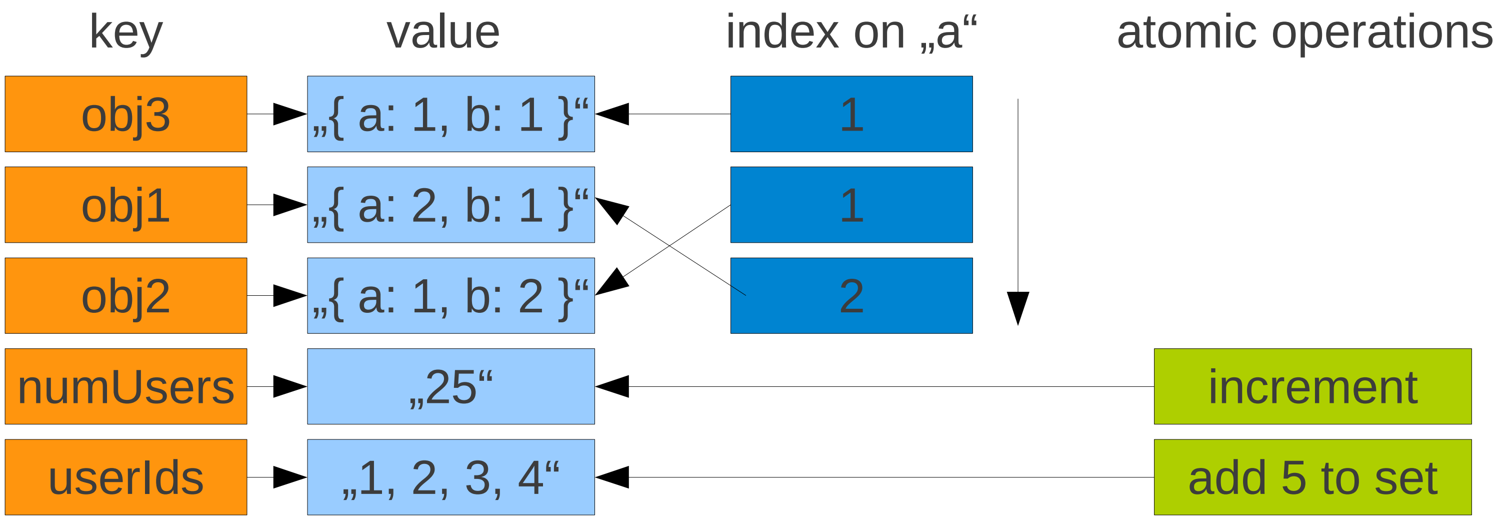
- Key-value stores:
  - Map value data to unique string keys (identifiers)
  - Allow access to data by key only
  - Treat data as opaque (data has no schema)
  - Do not save keys in predicatable order
  - Sometimes disallow key enumeration because it is expensive
  - May have one or multiple key spaces to logically similar group keys
  - Can implement scaling and partitioning easily due to simplistic data model



| key | value |
|---|---|
| obj3 | „{ a: 1, b: 1 }“ |
| obj1 | „{ a: 2, b: 1 }“ |
| obj2 | „{ a: 1, b: 2 }“ |
| numUsers | „25“ |
| userIds | „1, 2, 3, 4“ |

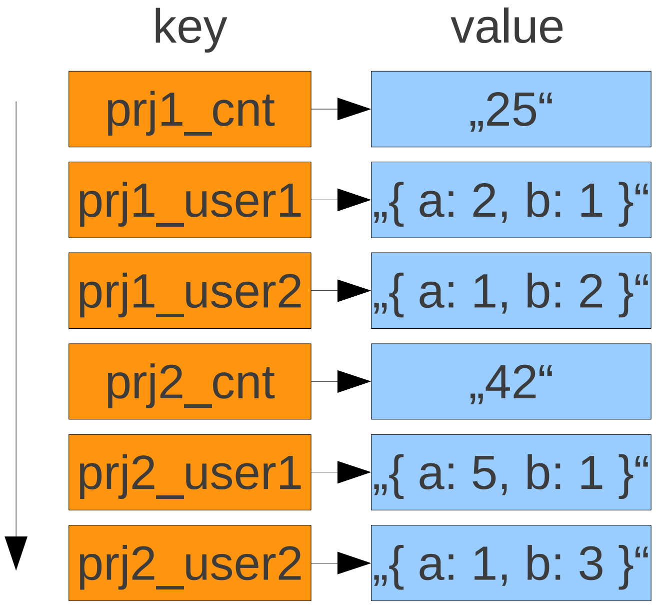
# NoSQL databases – categorisation

- Key-value store extensions (containers, counters, expiration):
  - may allow querying on value data, too, by offering secondary indexes. secondary index values must be provided by application because store still treats data as opaque (data has no schema)
  - may offer atomic container operations, counters (e.g. inc / dec) or value expiration features that allow broader queries and use cases



# NoSQL databases – categorisation

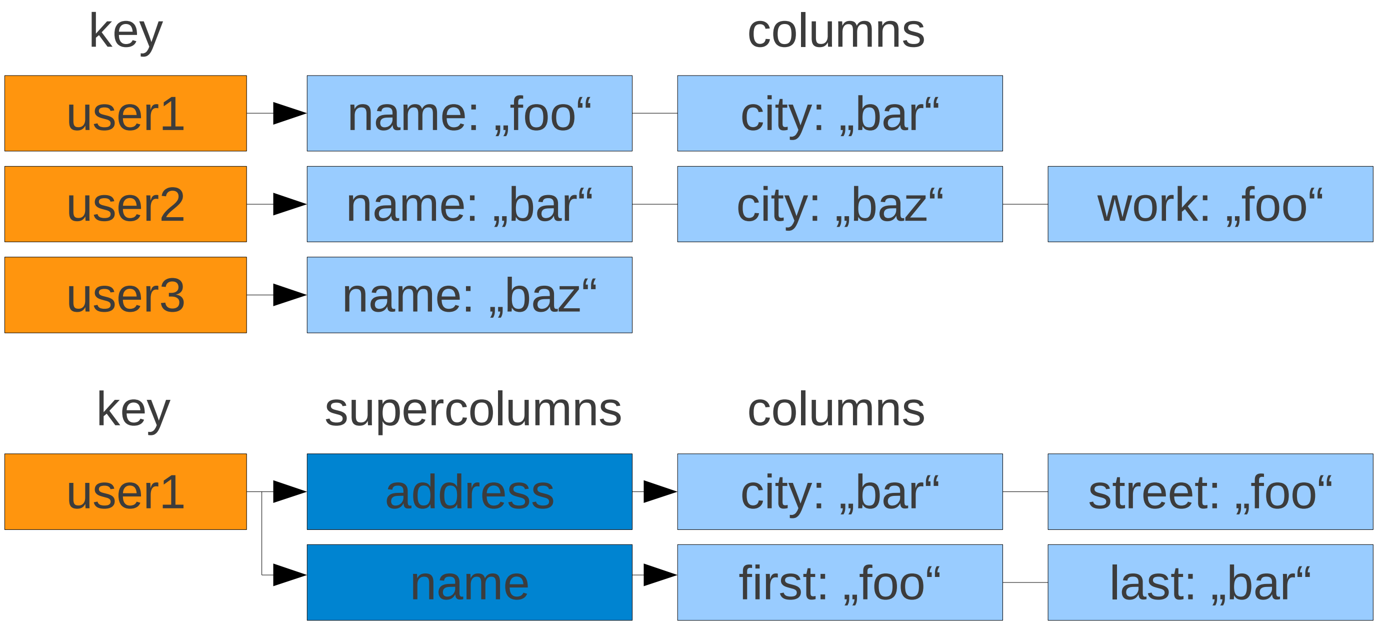
- Ordered key-value stores:
  - Co-locate values with adjacent keys so keys are stored sorted
  - Also allow range queries on keys
  - Still treat value data as opaque (data has no schema)



| key | value |
|---|---|
| prj1_cnt | „25“ |
| prj1_user1 | „{ a: 2, b: 1 }“ |
| prj1_user2 | „{ a: 1, b: 2 }“ |
| prj2_cnt | „42“ |
| prj2_user1 | „{ a: 5, b: 1 }“ |
| prj2_user2 | „{ a: 1, b: 3 }“ |

# NoSQL databases – categorisation

- Wide column stores:
  - Save values in an n-level map-of-maps substructure,
    e.g. column families, supercolumns and columns with value & timestamp
  - Allow querying specific attributes for keys (at least at the top levels)



key columns

user1 → name: „foo“ — city: „bar“

user2 → name: „bar“ — city: „baz“ — work: „foo“

user3 → name: „baz“

key supercolumns columns

user1 → address → city: „bar“ — street: „foo“

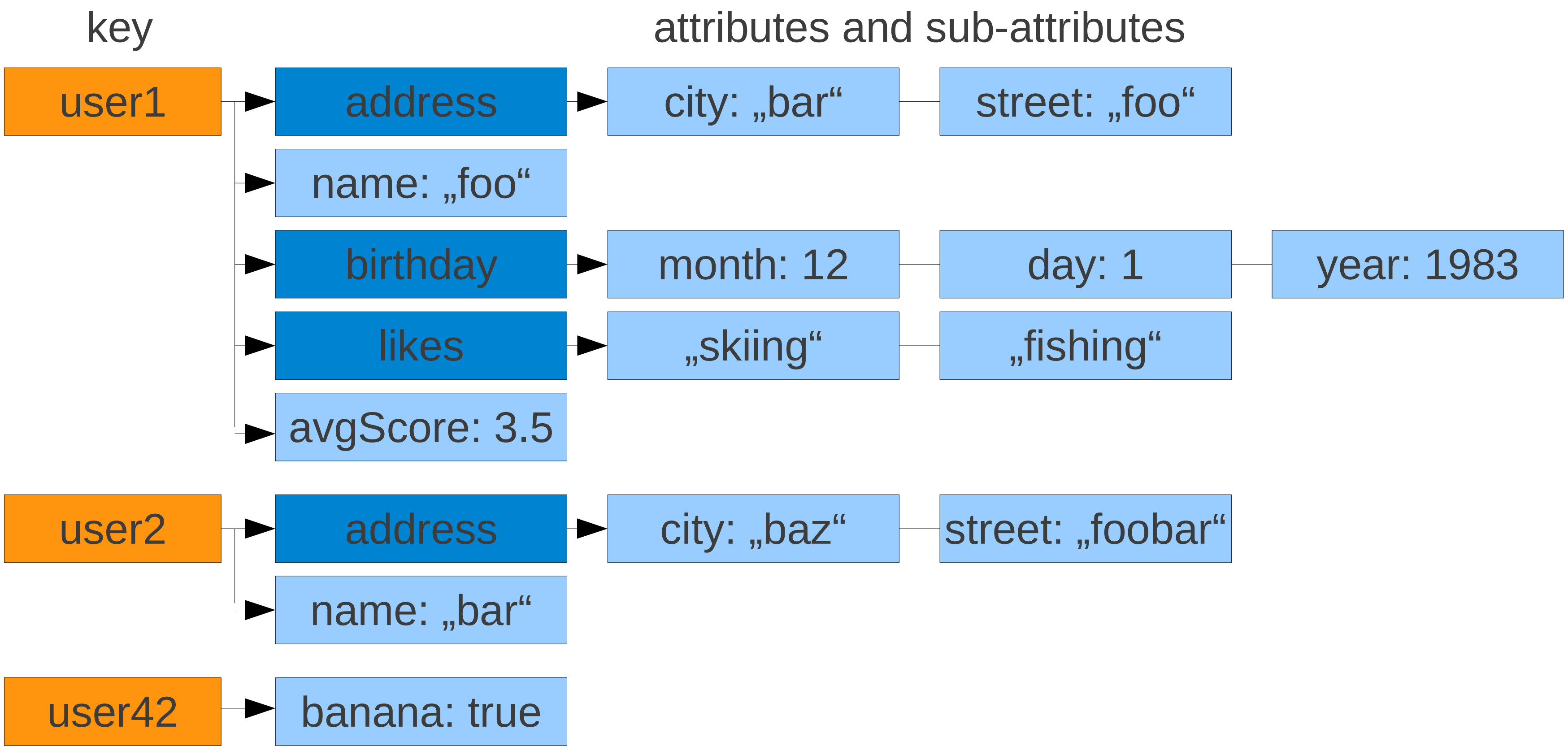
user1 → name → first: „foo“ — last: „bar“

# NoSQL databases – categorisation

- Document stores:
  - Normally based on key-value stores (each document still has a unique key)
  - Allow to save documents with logical similarity in „databases“ or „collections“
  - Treat data records as attribute-structured documents (data is no more opaque)
  - Use inherent schemas of documents to categorise attributes and sub-attributes
  - Introduce data types (primitive types, compound types)
  - Allow arbitrary nesting and document complexity using lists, arrays etc.
  - Allow individual documents to have different attributes, even if in same collection
  - Often allow querying and indexing document attributes

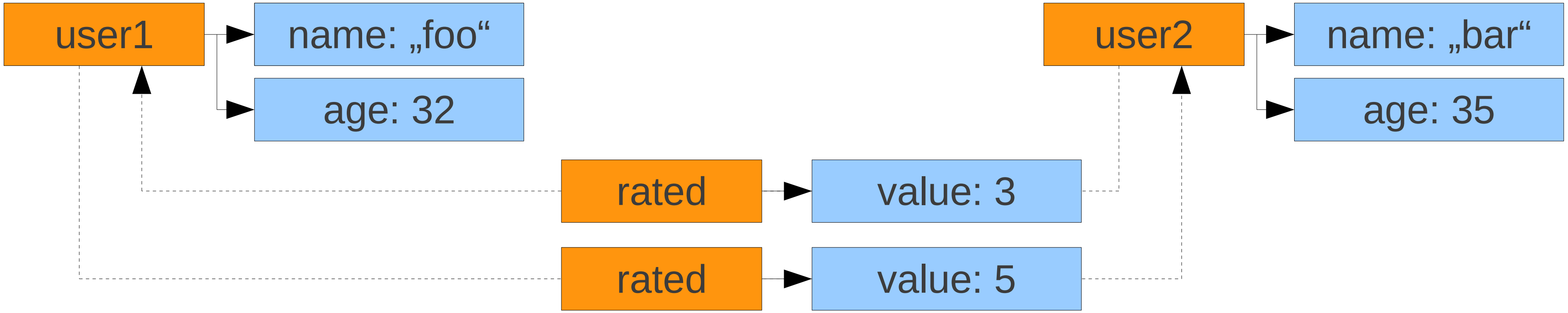
# NoSQL databases – categorisation

- Example documents in a document store:



# NoSQL databases – categorisation

- Graph databases:
  - Often based on key-value or document stores
  - Additionally save relationships (edges) between documents (vertices)
  - Allow querying relationships between documents
  - Relationships can have attributes as well



# Caveat emptor

- Things you should be aware of when using a NoSQL database:
  - multi-query, multi-document or multi-key „transactions“ may not be atomic and not be isolated
  - the application may need to handle inconsistent / stale reads
  - the application may need to resolve version conflicts or handle multiple schema versions of documents
  - referential integrity needs to be enforced by the application
  - if the datastore treats data as opaque, it will not at all validate it
  - by design, the database may not be able to answer some classes of queries, e.g. joins need to be performed by the application, not by the database
- Put simple: database will do less, application must do more!

# Which datastore to use?

- Relational databases are still a good fit for many problems, but not for all
- The same is true for NoSQL databases
- Each category of NoSQL databases and each individual product has its own sweet spots (and downsides)
- Check what you want to achieve first, then pick the right tool(s) for the job
- It may make sense to use different databases in parallel

# Why care about data modelling in NoSQL?

- Even in NoSQL databases, your data will have some structure
- This effective data model determines
  - the consistency level you're able to achieve
  - the retrieval and update performance
  - the storage space required for saving the data
  - the types of queries you will be able to run on the data
- So even if you don't define explicit schemas, you still need to care about structuring your data properly

# Some NoSQL data modelling considerations

- Data modelling for schema-less or dynamic schema datastores is different than modelling for relational databases
- Different NoSQL databases require different modelling
- The techniques to apply are not revolutionary, but in the relational world they are unnecessary or even anti-patterns
- A few things to look at:
  - Proper key and attribute naming
  - Building „indexes“ by hand for aggregation
  - Data denormalisation and duplication

# Picking „good“ keys – key length

- In plain key-value stores, keys are the only thing that you can query
- Keys are also present in other categories of NoSQL databases. Data elements are still identified using unique keys
- Keys should be picked so that they are sensible and legible
- Keys are normally stored just once, but...
- ...in some NoSQL databases, key values may be duplicated a lot of times in secondary indexes, append-only log structures etc.
- In these situations, keys should also be as short as can be tolerated to not waste disk space / RAM

# Key length – CouchDB example

- Case: insert 1000 documents with unique keys into vanilla CouchDB
- Key prefix is „very-long-id“
- For each document, we also store an attribute named „very-long-value“
- Setup:

```
for i in `seq 1000 1999`
 do
  curl -X POST                                    \
       --header "Content-Type: application/json"  \
       --data "{ \"_id\": \"very-long-id-$i\",    \
                 \"very-long-value\": \"$i\" }"   \
       http://127.0.0.1:5984/keytest
 done
```

# Key length – CouchDB example

- Now check how often the key prefix occurs in the datafile:

```
strings /db/keytest.couch | grep -c "very-long-id"
strings /db/keytest.couch | grep -c "very-long-value"
```

- Results:
  - the non-key attribute name „very-long-value“ is saved 1,000 times
  - the key prefix is stored 42,492 times instead of the expected 1,000 times !!
- This is due to MVCC that organises data in copy-on-write B+ trees

# Picking „good“ keys – hierarchical keys

- If in a key-value store only one global keyspace is available, use „hierarchical keys“ to distinguish keys from multiple applications or multiple users. This helps preventing unintended key collisions.

| Avoid these keys: | Instead, use such keys: |
|---|---|
| user1 | app1_user1 |
| profile42 | client6_profile42 |
| item23 | app1_client6_item23 |

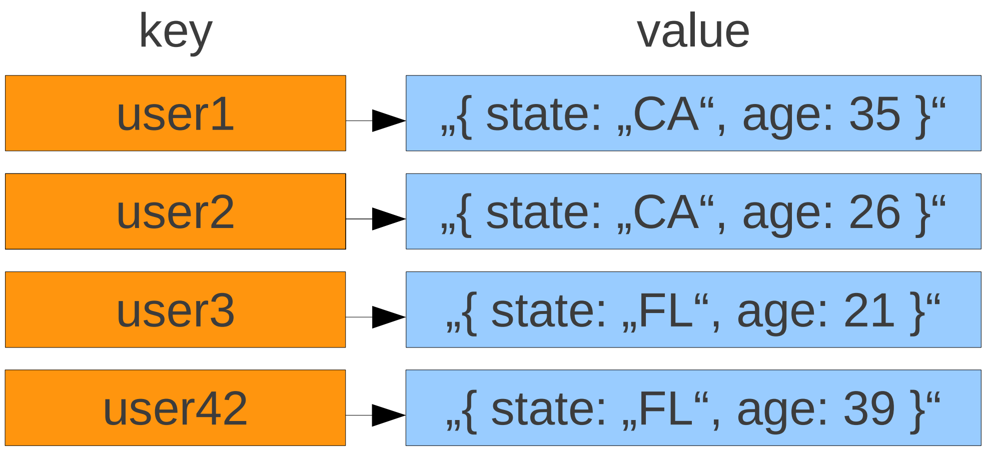
- If your datastore allows efficient key enumeration with prefixes, this also allows hierarchical operations by prefix, e.g. DELETE client6_*, GET app1_client6_item*
- Don't need this in document stores due to collections

# Attribute names in value data

- The value data you save is often a combination of attribute names and values, e.g. { „age“: 35, „gender“: „m“, „name“: „foo“ }
- In a fully schema-free datastore (e.g. plain key-value store), the store treats data as opaque and cannot tell attribute names and values in the value data apart
- In document stores, data is no more opaque. However, there are no fixed schemas. Each document, even if in the same collection, could have different attributes
- This means non-key attribute names are saved redundantly for EACH record/document
- Result: using attribute name „f“ will normally result in much lower disk footprint than using attribute name „firstName“.
- Check these MongoDB-specific links:
  - http://christophermaier.name/blog/2011/05/22/MongoDB-key-names
  - https://jira.mongodb.org/browse/SERVER-863

# Aggregation – home-made indexes

- NoSQL stores normally don't support aggregation queries, but you can workaround that by creating „index“ keys on your own.
  Initial data:



key → value

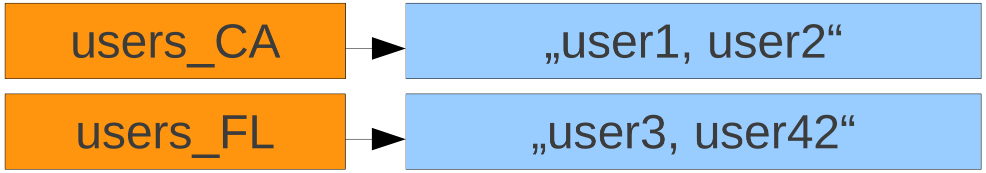
user1 → „{ state: „CA“, age: 35 }“

user2 → „{ state: „CA“, age: 26 }“

user3 → „{ state: „FL“, age: 21 }“

user42 → „{ state: „FL“, age: 39 }“

- From that you can derive „index“ keys that provide users by state.
  You can now run count queries easily:

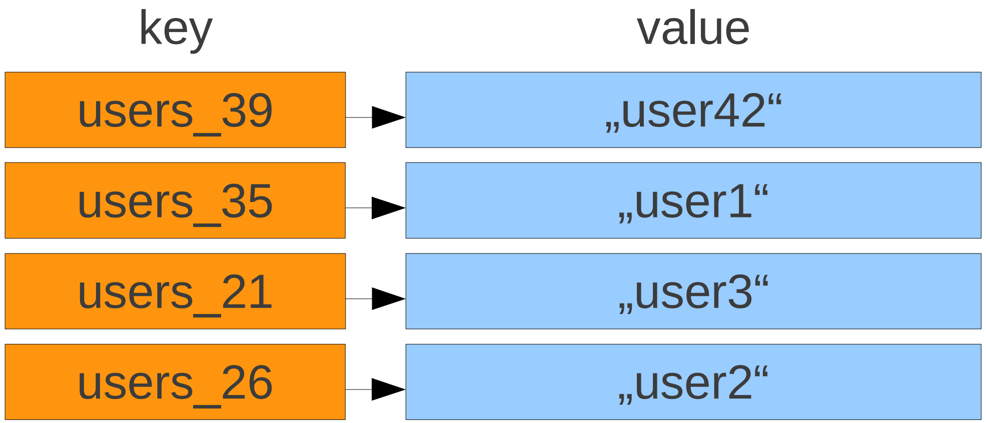


users_CA → „user1, user2“

users_FL → „user3, user42“

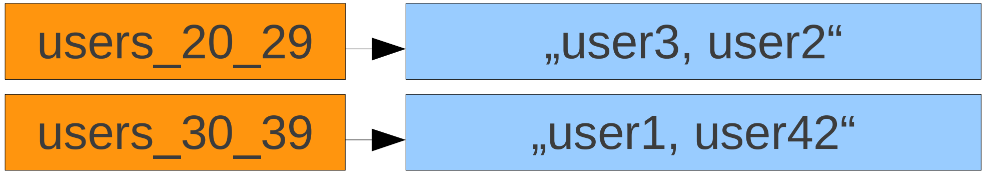
# Aggregation – home-made indexes

- „Index“ keys that provides users by age:



| key | value |
|---|---|
| users_39 | „user42“ |
| users_35 | „user1“ |
| users_21 | „user3“ |
| users_26 | „user2“ |

- Can also directly aggregate categories in „index“ keys if there are (too) many distinct category values:



| | |
|---|---|
| users_20_29 | „user3, user2“ |
| users_30_39 | „user1, user42“ |

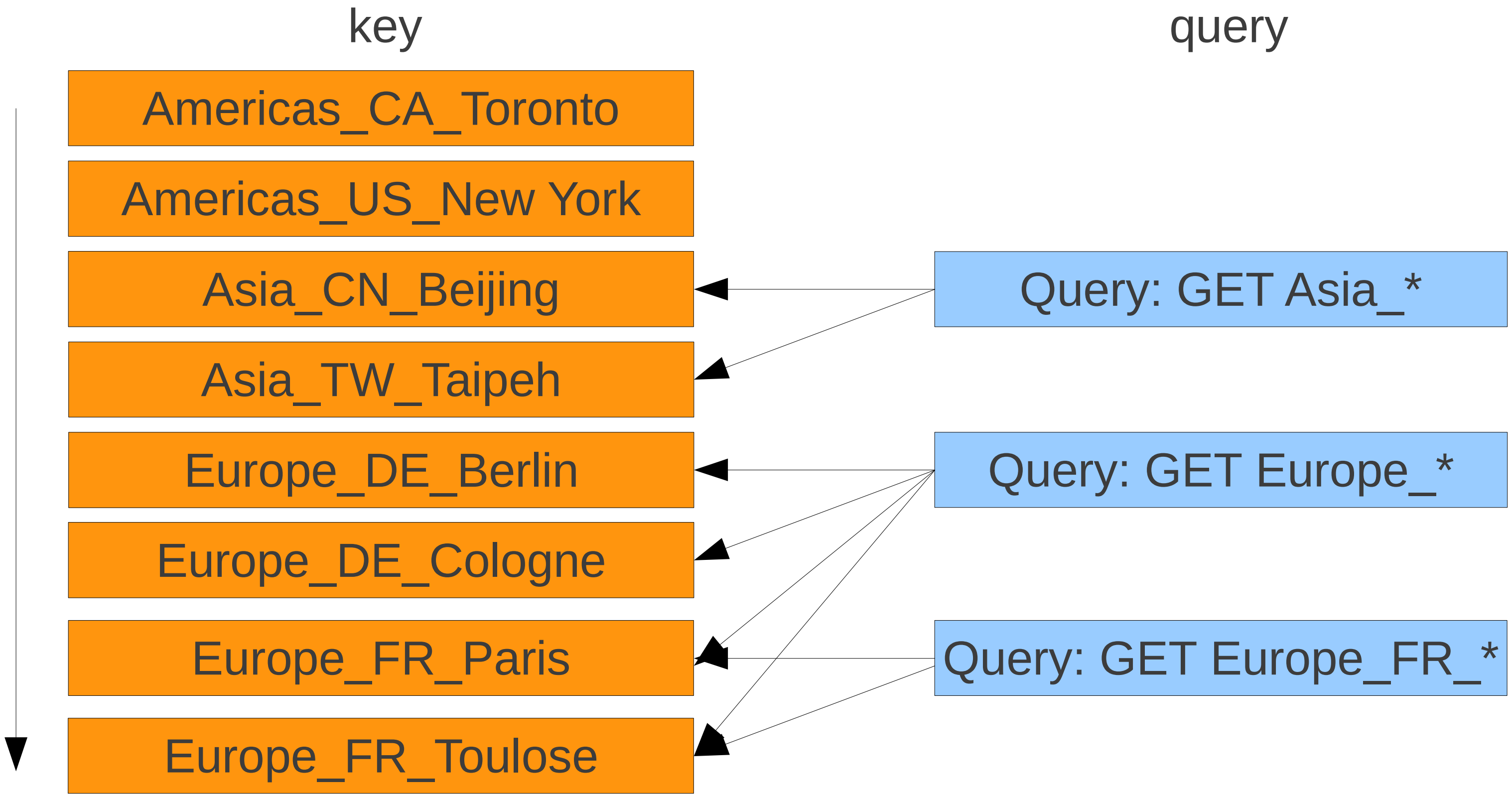
*(I know, I should have used better key names)*

# Aggregation – home-made indexes

- Home-made „indexes“ provide about the same functionality as materialised views do in relational databases
- And as in materialised views, this means data duplication!
- The database won't maintain the „indexes“ for you, so you have to maintain them yourself (from out of the application)
- Keeping the „indexes“ up-to-date may require data updates in several places
- Can easily get ouf of sync or provide an inconsistent view of the data
- Accuracy may still be good enough, but this depends on the use case

# Aggregation – hierarchical keys

- If your datastore allows efficient key range enumeration using leftmost prefix queries, using hierarchical „index“ keys can also be used for multi-dimensional aggregation

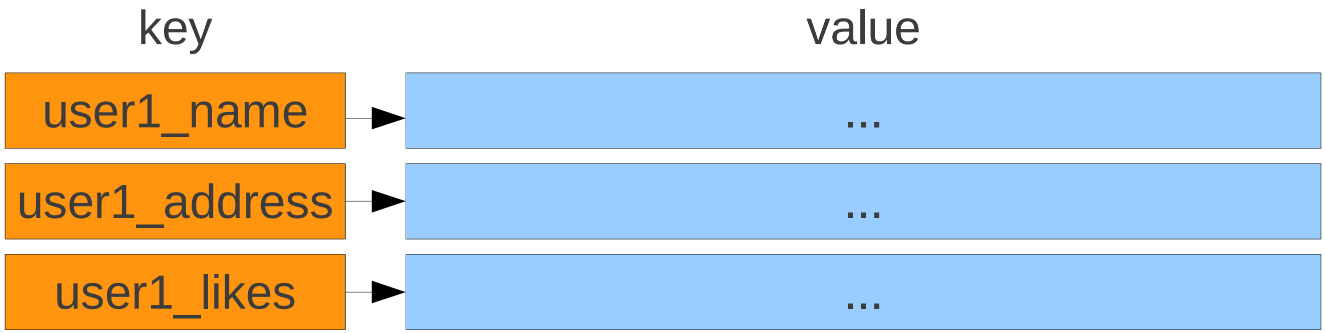


# Aggregation – key buckets

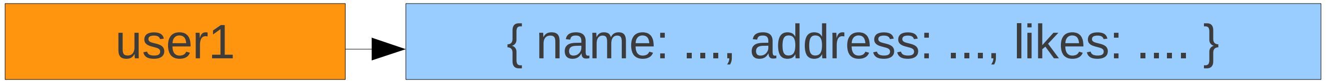
- If the datastore does not allow querying attribute values, buckets with predictable key names can be used to support aggegation
- Example case: users online on a website during the last x minutes
- Key name convention: „usersOnline_${slot}“, with ${slot} being
  slot = (time() / 60).floor() * 60
  (the current Unix timestamp rounded to a 60 second slot)
- This will produce predictable key names such as
  - usersOnline_1345732920
  - usersOnline_1345732980
  - UsersOnline_1345733040
- You can now easily save and query which users were online in which time slot and create reports

# Denormalisation – nesting sub elements

- It may be beneficial to all nest sub elements of a logical unit in the unit itself instead of saving the sub elements separately
- Document stores are built for exactly this case (nesting attributes), but this pattern can be used in key-value stores as well
- Instead of saving all sub elements individually



Can store all sub elements of a user in one key / document only:

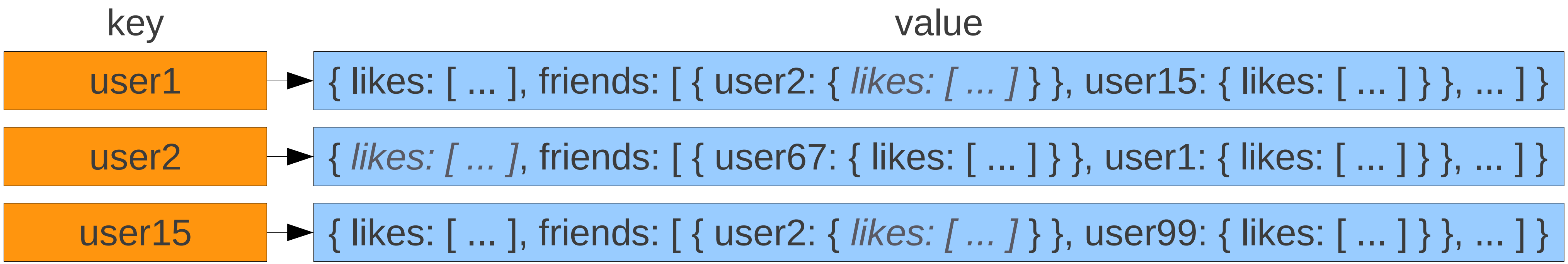


# Denormalisation – nesting sub elements

- Benefit: nesting sub elements reduces the number of elements that keep data for a logical unit (here: „user“)
- This reduces the number of read / write operations for each unit
- Accessing data of a single key is normally an atomic operation, so nesting prevents sub elements from getting out sync, avoiding inconsistent reads and writes of logical units
- In NoSQL databases, nesting is also a natural way to avoid joins, which are not offered by most NoSQL products

# Denormalisation – nesting sub elements

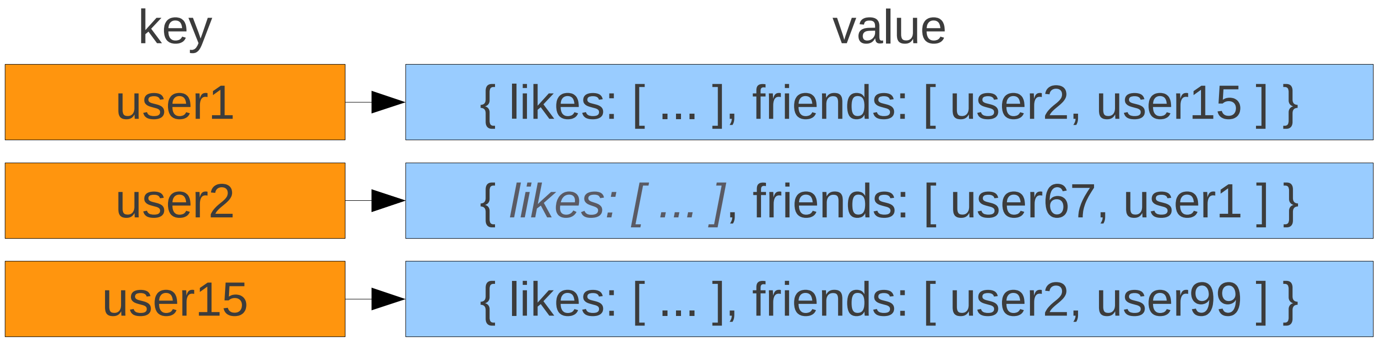
- Downside: nesting will increase the data volume stored for each key, increasing the overhead for each read and write in both the datastore and the application
- If you overdo nesting, you might end up having the same data stored in multiple places, and no clear responsibility for it
- This will waste space and cause consistency issues
- For example, a change to user2's „likes“ attribute requires 3 writes:



| key | value |
|---|---|
| user1 | { likes: [ ... ], friends: [ { user2: { *likes: [ ... ]* } }, user15: { likes: [ ... ] } }, ... ] } |
| user2 | { *likes: [ ... ]*, friends: [ { user67: { likes: [ ... ] } }, user1: { likes: [ ... ] } }, ... ] } |
| user15 | { likes: [ ... ], friends: [ { user2: { *likes: [ ... ]* } }, user99: { likes: [ ... ] } }, ... ] } |

# Denormalisation – nesting sub elements

- Instead of nesting such elements fully, prefer referencing them by key / id:



| key | value |
|---|---|
| user1 | { likes: [ ... ], friends: [ user2, user15 ] } |
| user2 | { *likes: [ ... ]*, friends: [ user67, user1 ] } |
| user15 | { likes: [ ... ], friends: [ user2, user99 ] } |

- This will allow you to update data in only one place, saving storage space and improving consistency

- The price you pay: to retrieve all details for the referenced items, you now need additional read operations

# Summary

- Data modelling for NoSQL databases is different than for relational databases:
  - Instead of normalising your schema, you'd rather denormalise your data
  - Data duplication is common to support analysis and aggregation queries, and for fast retrieval
- Different NoSQL databases have different features and to be used effectively, may require different data models
- To make best use of a specific store, you should have some knowledge about how it organises data internally and what guarantees it provides:
  - Atomic operations should be exploited where possible to improve consistency
  - Cross-record/document consistency is mostly not guaranteed.
    If your store at all offers it, it will be slowed down by it
- In general, logic is moved from the database to the application
  - Application needs to handle inconsistencies, version conflicts etc.
  - Your application has more control over the data, but it has to use this power

# Misc

- NoSQL user group Cologne:
  every 1st Wednesday / month
  http://www.nosql-cologne.org/
- NoSQL matters conference Barcelona:
  October 6th, 2012
  http://www.nosql-matters.org/
- ArangoDB 1.0 release:
  September 2012
  http://www.arangodb.org/