Big Data Engineering and Architecture

Topic 6: NoSQL Data Modeling (Key/Value and Document)

What is Data Modeling?

- Process of creating a data model for data to be stored in a database
- Contains
  - Data objects
  - Associations between objects
  - Rules
- Types of data models in a design:
  - Conceptual
  - Logical
  - Physical (we will focus on this)

Physical Data Model

Describes HOW the system will be implemented in the database

Database, namespace, collections

- Type/Tech
- Name
- Storage details
- Configuration/properties

Tables*, indexes

- Name
- Schema
- Keys & key design
- Partitions
- Storage details
- Configuration/properties
- Indexes

Fields, columns

- Name
- Data types
- Constraints
- Nullable?
- Order

**NOTE for graphs: Tables would be nodes and arcs*

Data Model Design

- Design the model(s) and document them
- Describe important characteristics
- Describe design considerations (why certain decisions were made)
- Use graphics wherever possible
- Use tables to describe attributes of each level
- Include examples and hints for users to anchor their understanding in instances

Key/Value Stores

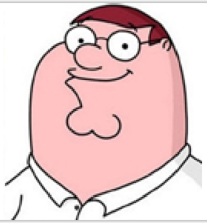
Overview

- Three essential features
  - Simplicity
  - Speed
  - Scalability
- Simplicity
  - Don’t need all features of RDBMS (joins, multiple entity queries, schema)
- Speed
  - Simple data structure/access → Fast reads/writes
- Scalability
  - Scale out with minimal disruption

Definitions

- Key
  - Reference to a value (like an address)
- Value
  - Data stored as referenced by a key (can be just about anything)
- Namespace
  - Collection of key-value pairs
  - Could be based on domain, database, or the entire key-value store
- Partition
  - Subset of a database
  - Store keys in different partitions based on their value

Data Samples

Key	Value
cust_1234_name	“Peter Griffin”
cust_1234_address	“31 Spooner St. Quahog, RI”
cust_1234_nameAddress	{“name”:”Peter Griffin”, “address”:”31 Spooner St. Quahog, RI”}
cust_1234_profileImage	
cust_2468_name	“Homer Simpson”
cust_2468_address	“742 Evergreen Terrace, Springfield, ???”
cust_2468_nameAddress	{“name”:”Homer Simpson”, “address”: “742 Evergreen Terrace, Springfield, ???”}
cust_2468_profileImage	

Data Structure Overview

- Data structure is an associative array in persistent storage
  - Dictionary, map, hash table
- Key-Value
  - Key is a unique identifier within the namespace
  - Value can be essentially anything (number, string, binary, etc.)
- Schemaless
  - Can hold multiple representations of the same data simultaneously
    - FirstName, LastName, FullName

Keys

- Unique identifier within a namespace
- Can use Id value (like SSN), but usually use compound keys
  - Id.Attribute: 1.Name, 1.HouseNumber, 1.Birthdate
  - Entity.Id.Attribute: cust_1234_firstName
  - Hash the combination?
    - Hash(“cust” + 1234 + “firstName”) → f7941de89a66fbaa082db5d9b255a57d
    - Might have collisions between hashed values
    - No range searching
    - Improved distribution

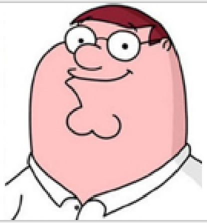
Key Design

- TIPS
  - Use meaningful and unambiguous naming components
    - ‘cust’ for customer, ‘srep’ for sales representative
  - Use range-based components if you need to retrieve ranges of values
    - Dates, Ids, etc.
  - Use a common delimiter when appending components
  - Keep keys as short as possible
- Well-designed keys minimize code required to access values
- Range key example for website visits
  - type_date_seqnum_attribute
  - visit_123116_1_sessionId, visit_123116_sessionId, etc.
- Make sure you account for the limitations of the KVDB you use
  - Byte limitations, data types
- Keep in mind your parititioning scheme when defining key structure

Values

- Strong typing NOT required
  - “31 Spooner Street, Quahog, RI”
  - (“31 Spooner Street”, “Quahog”, “RI”)
  - { “street”:”31 Spooner Street”, “City”:”Quahog”, “State”:”RI” }
- TIP: Make implementation choices that lead to *some* restrictions
- Can’t search values, but can create a search index based on values
  - ‘IL’: (‘cust.1234.state’, ‘cust.2468.state’, ‘store.1011.state’)
  - Index the values and keys that have that value
  - Not necessarily efficient, but could improve scans
- Structured data can help reduce latency
  - Store commonly associated values together, like {firstName:Don, lastName: Sawyer}

Data Samples (Again)

Key	Value
cust_1234_name	“Peter Griffin”
cust_1234_address	“31 Spooner St. Quahog, RI”
cust_1234_nameAddress	{“name”:”Peter Griffin”, “address”:”31 Spooner St. Quahog, RI”}
cust_1234_profileImage	
cust_2468_name	“Homer Simpson”
cust_2468_address	“742 Evergreen Terrace, Springfield, ???”
cust_2468_nameAddress	{“name”:”Homer Simpson”, “address”: “742 Evergreen Terrace, Springfield, ???”}
cust_2468_profileImage	

Other Definitions

- Namespace
  - Collection of key-value pairs (aka set, collection, bucket)
  - Separate data in namespaces like customer, orders, products
- Partition
  - Organize data based on keys
    - Customer 1-10000, 10001-20000, …
    - Different servers manage different partitions (multiple servers hold replicas of partitions for distributed reading)
    - You must understand your partitions and size in each
      - If customers are organized by name, there may be many more S’s than Z’s
    - Could use hash function to create hashed keys for better distribution
- Partition Key
  - The key used to determine which partition should hold a data value
- TTL (Time to Live)
  - Data goes away after a specific period of time

Improving Performance

- Well-designed keys
- Structured values for fewer lookups
- Copies of data (denormalization) to assist usage
  - firstName, lastName, address, firstLastNameAddress
  - Combining data puts data in the same storage block (I/O reduction)
  - No need for joins
- If structured values are large, consider a document database

Limitations

- Can only look up values by key
  - Some KVDBs have some version of search
  - Riak indexes values as well as keys
  - Use secondary indexes (if supported) or inverted indexes
- Range queries may not be supported
  - Ordered key-value db allows for this
- No comparable query language to SQL for RDBMS's
  - Cassandra has CQL
  - If storing JSON/XML, could integrate with Solr/Lucene for text search

Key/Value: Useful When

- Unstructured data is required
- High performance read/writes
- Value is fully identifiable via key alone
- Value is not dependent on other values
- Values simplistic in structure or binary
- Simple query patterns (insert, select, delete only)
  - Ease of storage & retrieval more important than complex data structures
  - Update might be available
- Values are manipulated at application layer

Key/Value: Not Useful When

- Need to search or filter data within the stored value
- Relationships exist between different key-value entries (joins)
- Multiple keys’ values need to be updated in single transaction
- Multiple keys need to be modified in a single operation
- Schema consistency across values is required
- Partial updates are required (single attribute in a value)

Document Databases

Data Samples

JSON Document


```json
{
    "addresses": [
        {
            "city": "Quahog",
            "is_primary": "true",
            "number": "31",
            "state": "RI",
            "street": "Spooner St.",
            "type": "home"
        }
    ],
    "age": 42,
    "children": [
        {
            "name": "Meg Griffin",
            "type": "daughter"
        },
        {
            "name": "Brian Griffin",
            "type": "son"
        },
        {
            "name": "Stewie Griffin",
            "type": "son"
        }
    ],
    "first_name": "Peter",
    "last_name": "Griffin",
    "spouses": [
        {
            "name": "Lois Griffin"
        }
    ]
}
```


XML Document


```xml
<Person>
    <Addresses>
        <Address IsPrimary="True" Type="home">
            <City>Quahog</City>
            <Number>31</Number>
            <State>RI</State>
            <Street>Spooner St.</Street>
        </Address>
    </Addresses>
    <Age>42</Age>
    <Children>
        <Child Type="daughter">
            <Name>Meg Griffin</Name>
        </Child>
        <Child Type="son">
            <Name>Brian Griffin</Name>
        </Child>
        <Child Type="son">
            <Name>Stewie Griffin</Name>
        </Child>
    </Children>
    <FirstName>Peter</FirstName>
    <LastName>Griffin</LastName>
    <Spouses>
        <Spouse>
            <Name>Lois Griffin</Name>
        </Spouse>
    </Spouses>
</Person>
```

Design/Features

- Designed for scalability
- Provides flexibility about the structure of documents
  - Not all documents need to have the same structure
  - Can be completely unrelated documents (not advised)
    - E.G.: customer, sales, clickstream, logs
- Designed to accommodate variations and schema evolution
  - Hence, try to avoid explicit schema definitions like a RDBMS
- Schemaless
  - Schema specification not *required* (not schema on write)
  - Allows adding k-v pairs to documents
  - Application code must enforce rules about data
- For performance, think about balancing normalization vs. denormalization

Document Collections

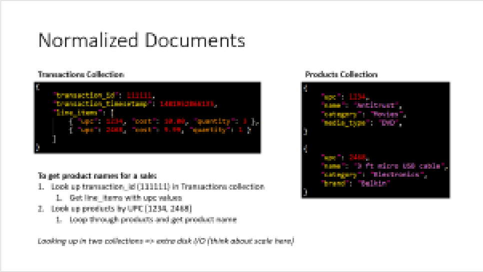
- Collection is a group/list of *related* documents
  - Documents don't have to have the same structure, but should share some common structure
- Tips
  - Avoid highly abstract entity types
    - Filtering collections is slower than working with multiple collections
    - Collections are stored near each other on disk, so you could end up reading a lot of data that needs to be filtered
  - You could index mixed entities, but won't necessarily be faster
    - If the entities are in separate collections, it might be faster to just scan the collection than read an index from disk
    - Indexing consumes resource to keep updated
  - Can use indexes on key terms instead scanning entire documents for an attribute value
  - Code for manipulating collections should apply to most/all documents
  - Use document subtypes when entities are aggregated/share substantial code
    - Products (appliances, clothing, music, toys)

Normalization vs. Denormalization

- REMEMBER: DocDB is being used for its scalability
- Goal: keep data frequently used together in the document
  - Larger documents can lead to fewer documents retrieved in a block
- Consider queries your application will issue to the DB

Joins Look Like (in application code):

```
for transactions in {Transaction collection query}
    for products in {Product collection query}:
        do something with transactions/products
```



Operations on Document Databases

- Insert, update, delete, retrieve
- No standard data manipulation language (MongoDB Examples Below)
  - `db.product.insert({"product_id": 1234, "cost": 10.00})`
  - `db.product.remove({"product_id": 1234})`
  - `db.product.remove({"cost": {"$gte": 10.00}})`
  - `db.product.update({"product_id": 1234}, {$set {"cost": 9.99}})`
  - `db.product.update({"product_id": 1234}, {$set {"weight": 1.0}})`
  - `db.product.find({"product_id": 1234})`
- Tips
  - Include unique identifier with each document
  - Usually more efficient to bulk insert instead of [individual inserts]()
  - Be careful when deleting documents with references to other documents

Individual Inserts vs. Bulk Insert

Individual Inserts

```
db.product.insert(
    {"product_id": 1234,
     "cost": 10.00} )

db.product.insert(
    {"product_id": 1235,
     "cost": 1.50} )

db.product.insert(
    {"product_id": 1236,
     "cost": 100.99} )
```

Bulk Insert

```
db.product.insert( [
    {"product_id": 1234,
     "cost": 10.00},

    {"product_id": 1235,
     "cost": 1.50},

    {"product_id": 1236,
     "cost": 100.99} ])
```

Return to Operations on Document Databases

Design: Partitioning

- Uses horizontal partitioning, not vertical
  - AKA Sharding
  - Divide database by documents (rows in RDBMS)
- Enables DB to scale horizontally
- Need to select a shard key + partitioning method
  - Key must exist in all documents
- Partitioning methods
  - Range (dates, numbers, alphabetic)
  - Hash (distribute keys evenly in partitions)
  - List (partition by lists of entities)
    - P1: Clothing | P2: Electronics, Toys, Office Supplies | P3: Grocery, Pharmacy

More Design Considerations

- Physical model: planning for mutability
  - On creation a document is allocated space + room for growth
  - If the document grows > block, might be moved to another block
    - Creating a document w/ sufficient space will help avoid this
- Indexing
  - Too few => poor read performance
  - Too many => poor write performance
  - Read-heavy => index most/all fields
  - Write-heavy => focus on essential indexes (keys + identifiers of relations)
  - TIP: EXPERIMENT and iterate

Modeling Relations

- Many-to-many
  - Use two collections
  - Each collection maintains a list of identifiers of related documents
  - Data integrity: be careful when updating many-to-many
- Hierarchies (parent-child, taxonomies)
  - Option: reference parent id of the parent
    - When you often need to traverse upwards
  - Option: reference child ids of children
    - When you often need to traverse downward
  - Option: list all ancestors (in path order)
    - When you need to know full path (single read to get full tree)

Reference Slides

Slides referenced from earlier slides

Normalized Documents

Transactions Collection

```
{
    "transaction_id": 111111,
    "transaction_timesetamp": 1481952866135,
    "line_items": [
        { "upc": 1234, "cost": 10.00, "quantity": 3 },
        { "upc": 2468, "cost": 9.99, "quantity": 1 }
    ]
}
```

Products Collection

```
{
    "upc": 1234,
    "name": "Antitrust",
    "category": "Movies",
    "media_type": "DVD",
}

{
    "upc": 2468,
    "name": "3 ft micro USB cable",
    "category": "Electronics",
    "brand": "Belkin"
}
```

To get product names for a sale:

1. Look up transaction_id (111111) in Transactions collection
    1. Get line_items with upc values
2. Look up products by UPC [1234, 2468]
    1. Loop through products and get product name

Looking up in two collections => extra disk I/O (think about scale here)

Denormalizing

Normalized

```
{
    "transaction_id": 111111,
    "transaction_timesetamp": 1481952866135,
    "line_items": [
        { "upc": 1234, "cost": 10.00, "quantity": 3 },
        { "upc": 2468, "cost": 9.99, "quantity": 1 }
    ]
}

{
    "upc": 1234,
    "name": "Antitrust",
    "category": "Movies",
    "media_type": "DVD",
}

{
    "upc": 2468,
    "name": "3 ft micro USB cable",
    "category": "Electronics",
    "brand": "Belkin"
}
```

Denormalized

```
{
    "transaction_id": 111111,
    "transaction_timesetamp": 1481952866135,
    "line_items": [
        {
            "cost": 10.00,
            "quantity": 3,
            "product":
            {
                "upc": 1234,
                "name": "Antitrust",
                "category": "Movies",
                "media_type": "DVD"
            }
        },
        {
            "cost": 9.99,
            "quantity": 1,
            "product":
            {
                "upc": 2468,
                "name": "3 ft micro USB cable",
                "category": "Electronics",
                "brand": "Belkin"
            }
        }
    ]
}
```

Use Cases

Some use cases where a document database could be used.

Give one reason why you would choose to move the following document data to a key-value database. Given an example of the schema design for the key-value record(s) for optimal performance.

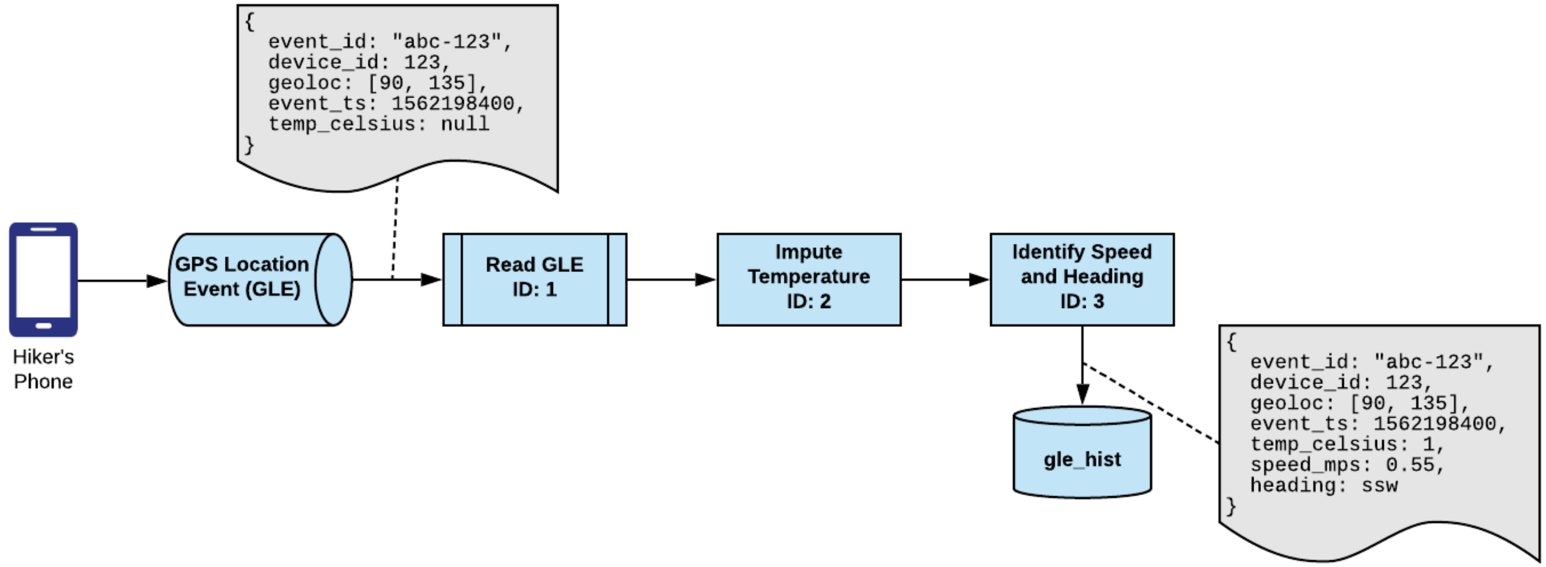
Document Collection: Hockey Scores

```
{
  “game_id” : 1,
  “home_score” : 4,
  “visitor_score” : 3
}
```

Use Case #1: Metric Tracking

Design a database that can track provenance about your data pipeline. Use the example below.

Hiking App Event Tracking Data Pipeline



Check for Duplicates in a Realtime Stream

Realtime streams are often built for extremely fast processing of data coming at a high velocity. Many of the technologies use don't guarantee "exactly once" processing, rather "at least once" processing. How can you solve this potential duplication problem with K-V stores?