Large-Scale and Multi-Structured Databases

Key-value Databases Design Tips and Case Study

Prof Pietro Ducange

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

IN SUPREMÆ DIGNITATIS 1343
UNIVERSITÀ DI PISA

CROSSLAB Innovation for industry 4.0

Coding Tips

- ***Well-designed*** key pattern helps minimize the amount of code a developer needs to write to create functions that access and set values.

  ```
  cust:1234123:firstName: “Pietro”
  custr:1234123:lastName: “Ducange”
  ```

- Using generalized ***set*** and ***get*** functions helps improve the readability of code and reduces the repeated use of low-level operations, such as concatenating strings and looking up values.

- Consider to have a ***naming*** convention for ***namespaces***.

- In production applications, we should include appropriate ***error checking and handling***

Naming Convention for Keys

- Use ***meaningful*** and ***unambiguous*** naming components, such as 'cust' for customer or 'inv' for inventory.

- Use ***range-based*** components when you would like to retrieve ranges of values. Ranges include dates or integer counters.

- Use a ***common delimiter*** when appending components to make a key (e.g. the ‘:’ delimiter)

- Keep keys as ***short*** as possible without sacrificing the other characteristics mentioned in this list.

Get and Set Functions

```
define getCustAttr(p_id, p_attrName)
    v_key = 'cust' + ':' + p_id + ':' + p_attrName;
    return(AppNameSpace[v_key]);
```

```
define setCustAttr(p_id, p_attrName, p_value)
    v_key = 'cust' + ':' + p_id + ':' + p_attrName
    AppNameSpace[v_key] = p_value
```

AppNameSpace is the name of the ***namespace*** holding keys and values for this application.

Dealing with Ranges of Values (I)

Query: retrieve all customers who made a purchase on a particular date.

We can define keys associated with the customers who purchased products on a specific date as in the following example:

```
cust:061514:1:custId
cust:061514:2:custId
cust:061514:3:custId
cust:061514:4:custId
```

This type of key is useful for querying ranges of keys because you can easily write a function to retrieve a range of values.

Dealing with Ranges of Values (II)

The following function retrieves a list of customerIDs who made purchases on a particular date:

```
define getCustPurchByDate(p_date)
    v_custList = makeEmptyList();
    v_rangeCnt = 1;

    v_key = 'cust:' + p_date + ':' + v_rangeCnt +
      ':custId';
    while exists(v_key)
        v_custList.append(myAppNS[v_key]);
        v_rangeCnt = v_rangeCnt + 1;
        v_key = 'cust:' + p_date + ':' + v_rangeCnt +
          ':custId';

    return(v_custList);
```

Simple or Complex Values? (I)

Consider the following function which retrieves both the name and the address:

```
define getCustNameAddr(p_id)
    v_fname = getCustAttr(p_id,'fname');
    v_lname = getCustAttr(p_id,'lname');
    v_addr  = getCustAttr(p_id,'addr');
    v_city = getCustAttr(p_id,'city');
    v_state = getCustAttr(p_id,'state');
    v_zip = getCustAttr(p_id,'zip');
    v_fullName = v_fname + ' ' + v_lname;
    v_fullAddr = v_city + ' ' + v_state + ' ' + v_zip;
    return(makeList(v_fullName, v_fullAddr);
```

The function makes 6 access to the database (on the disk) using the `getCustAttr` function.

To speedup the function execution, we should reduce the number of times the developer has to call `getCustAttr` or caching data in memory.

Simple or Complex Values? (II)

The `getCustNameAddr` function ***makes six access*** to the database (on the disk) using the `getCustAttr` function.

To speedup the function execution, we should ***reduce*** the number of times the developer has to ***call*** `getCustAttr` or caching data in memory.

We may store ***commonly*** used attribute values ***together*** as follows:

```
cstMgtNS[cust: 198277:nameAddr] = '{ 'Jane Anderson' ,
  '39 NE River St. Portland, OR 97222'}
```

Key-value databases usually store the entire list together in a ***data block***, thus just one block in the disk will be accessed, rather than six.

Simple or Complex Values? (III)

Pay attention with too complex data structure for values.


```
{
  'custFname': 'Liona',
  'custLname':  'Williams',
  'custAddr' :  '987 Highland Rd',
  'custCity' :  'Springfield',
  'custState': 'NJ',
  'custZip'  :  21111,
  'ordItems' [
        {
            'itemID' : '85838A',
             'itemQty' : 2 ,
            'descr' : 'Intel Core i7-4790K Processor
              (8M Cache,
   4.40 GHz)',
        'price:' : $325.00
        } ,
        {
         'itemID' : '38371R',
         'itemQty' : 1 ,
        'descr' : 'Intel BOXDP67BGB3 Socket 1155, Intel
          P67',
            CrossFireX & SLI SATA3&USB3.0, A&GbE, ATX
            Motherboard',
        'price' : $140.00
         } ,
         {
        'itemID' : '10484K',
        'itemQty' : 1,
        'descr' : 'EVGA GeForce GT 740 Superclocked Single
          Slot 4GB
            DDR3 Graphics Card'
             'price': '$201.00'
          } ,
```


This ***entire structure*** can be stored under an order key, such as `'ordID:781379'`.

The advantage of using a structure such as this is that much of the information about orders is available with a ***single key lookup***.

As the structure ***grows in size***, the time required to read and write the data can increase, because the value, will be store in ***more than one memory block***.

In general, if we need to use complex structure for the DB of our application, it is better to move towards different architectures, such as ***Document Databases***.

Limitation of Key Value DB

The only way to look up values is by key
Some DBMSs for key-value DB offer APIs that support common search features, such as wildcard searches, proximity searches, range searches, and Boolean operators. Search functions return a set of keys that have associated values that satisfy the search criteria.

Some key-value databases do not support range queries.
Some DBMSs (ordered KV databases) keeps a sorted structure that allows for range queries and/or support secondary indexes and some text search.

There is no standard query language comparable to SQL for relational databases

Case Study: K-V DBs for Mobile App Configuration (I)

We consider a Mobile Application used for ***tracking*** customer shipments

The following ***configuration information*** about each customer are stored in a centralized database:

- Customer name and account number
- Default currency for pricing information
- Shipment attributes to appear in the summary dashboard
- Alerts and notification preferences
- User interface options, such as preferred color scheme and font

Most of the operations on the DB will be ***read operations***

Case Study: Naming Convention

Naming Convention for keys -> entity type:account number

Identified entities:

- Customer information, abbreviated ‘cust’
- Dashboard configuration options, abbreviated ‘dshb’
- Alerts and notification specifications, abbreviated ‘alrt’
- User interface configurations, abbreviated ‘ui’

Case Study: Entities attributes and values (I)

Customer Information:

```
TrackerNS['cust:4719364'] = {'name':'Prime Machine, Inc.',
  'currency':'USD'}
```

Dashboard Configuration options:

```
TrackerNS['dash:4719364'] =
  {'shpComp','shpState','shpDate','shpDelivDate'}
```

Dashboard Configuration Options

- Ship to company (shpComp)
- Ship to city (shpCity)
- Ship to state (shpState)
- Ship to country (shpCountry)
- Date shipped (shpDate)
- Expected date of delivery (shpDelivDate)
- Number of packages/containers shipped (shpCnt)
- Type of packages/containers shipped (shpType)
- Total weight of shipment (shpWght)
- Note on shipment (shpNotes)

Case Study: *Entities attributes and values (I)*

Alert and notification specifications:

```
TrackerNS[alrt:4719364] =
     { altList :
     {'jane.washingon@primemachineinc.com','pickup'},
     {'(202)555-9812','delay'}
}
```

User interface configuration options:

```
TrackerNS[alrt:4719364] = { 'fontName': 'Cambria',
            'fontSize': 9,
            'colorScheme' : 'default'
     }
```

Suggested Readings

Chapter 5 of the book *"Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015"*

Images

Al the images shown in this lecture have been extracted from:

“Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015”