Data Modeling

FoundationDB’s core provides a simple data model coupled with powerful transactions. This combination allows building richer data models and libraries that inherit the scalability, performance, and integrity of the database. The goal of data modeling is to design a mapping of data to keys and values that enables effective storage and retrieval. Good decisions will yield an extensible, efficient abstraction. This document covers the fundamentals of data modeling with FoundationDB.

- For general guidance on application development using FoundationDB, see Developer Guide (developer-guide.html).
- For detailed API documentation specific to each supported language, see API Reference (api-reference.html).

Modeling with layers

Before we dive into details, it's worth noting that not all data modeling in FoundationDB needs to be done from scratch. FoundationDB makes available a number of layers that implement common data models, exposing them as APIs. One of the most commonly used layers, the tuple layer, is discussed below. For a description of our other layers, see our layers page (/layers).

The core data model

FoundationDB’s core data model is an ordered key-value store. Also known as an ordered associative array, map, or dictionary, this is a common data structure composed of a collection of key-value pairs in which all keys are unique. Starting with this simple model, an application can create higher-level data models by mapping their elements to individual keys and values.

In FoundationDB, both keys and values are simple byte strings. Apart from storage and retrieval, the database does not interpret or depend on the content of values. In contrast, keys are treated as members of a total order, the lexicographic order over the underlying bytes, in which keys are sorted by each byte in order. For example:

- '0' is sorted before '1'
- 'apple' is sorted before 'banana'
- 'apple' is sorted before 'apple123'
- keys starting with 'mytable\' are sorted together (e.g. 'mytable\row1', 'mytable\row2', ...)

The ordering of keys is especially relevant for range operations. An application should structure keys to produce an ordering that allows efficient data retrieval with range reads.

Encoding data types

Because keys and values in FoundationDB are always byte strings, an application developer must serialize other data types (e.g., integers, floats, arrays) before storing them in the database. For values, the main concerns for serialization are simply CPU and space efficiency. For keys, there’s an additional consideration: it’s often important for keys to preserve the order of the data types (whether primitive or composite) they encode. For example:

Integers

- The standard tuple layer provides an order-preserving, signed, variable length encoding.
- For positive integers, a big-endian fixed length encoding is order-preserving.
- For signed integers, a big-endian fixed length two’s-complement encoding with the most significant (sign) bit inverted is order-preserving.

Unicode strings

- For Unicode strings ordered lexicographically by Unicode code point, use UTF-8 encoding. (This approach is used by the tuple layer.)
- For Unicode strings ordered by a particular collation (for example, a case insensitive ordering for a particular language), use an appropriate string collation transformation and then apply UTF-8 encoding. Internationalization or "locale" libraries in most environments and programming languages provide a string collation transformation, for example C (http://pubs.opengroup.org/onlinepubs/7908799/xsh/wcsxfm.html), C++
Floating point numbers

IEEE floating point numbers in a big-endian encoding with the most significant bit inverted have a total ordering which is more or less compatible with the mathematical one, except that -0 and +0 are not equal, and NaN values will be treated differently than by the IEEE standard comparisons (which are not a total ordering).

Composite types

An application's data is often represented using composite types, such as structures or records with multiple fields. It's very useful for the application to use composite keys to store such data. In FoundationDB, composite keys can be conveniently represented as tuples that are mapped to individual keys for storage.

For the purpose of illustration, we'll use the FoundationDB's Python language binding, including the `@fdb.transactional` decorator described in Python API documentation. The design patterns illustrated are applicable to all of the languages supported by FoundationDB.

Tuples

FoundationDB's keys are ordered, making tuples a particularly useful tool for data modeling. FoundationDB supports tuples by providing a built-in layer (available in each language binding) to encode tuples into keys. This layer lets you store data using a tuple like `(state, county)` as a key. Later, you can perform reads using a prefix like `(state,)`. The layer works by preserving the natural ordering of the tuples.

You could implement a naive encoding of tuples of strings into keys by using a tab character as a simple delimiter. You could do this with the following Python code:

```python
def tupleToKeyWithTab(tup):
    return '\t'.join(str(i) for i in tup)
```

# Example: Order first by state, then by county

```python
@fdb.transactional
def setCountyPopulation(tr, state, county, population):
    tr[tupleToKeyWithTab((state, county))] = str(population)
```

In this example, population figures for the United States are stored using keys formed from the tuple of state and county.

Of course, this encoding would only work if all the bytes in the individual keys in the tuple were greater than the delimiter byte. Therefore, FoundationDB's built-in tuple layer implements a more robust encoding supporting elements of various data types: byte strings, unicode strings, 64-bit signed integers, and null values.

The tuple layer's encoding is compatible between languages, although some languages are limited in what data types they support. For language-specific documentation of the tuple layer, see the corresponding API Reference documentation.

Because of its ordering of keys, FoundationDB supports efficient range reads on any set of keys that share a prefix. The tuple layer preserves the ordering of tuples sorted by element from left to right; as a result, the leftmost elements of a tuple will always represent a prefix in key space and can be used for range reads. A basic principle of data modeling with the tuple layer is to order tuple elements to facilitate such range reads. The examples below illustrate this principle.

Sometimes data attributes will have a natural order of containment imposed by your domain. A common example is geographic attributes, such as state and county in the United States. By constructing keys from tuples of the form `(state, county)`, where state is the first tuple element, all data for states will be stored in an adjacent range of keys. This ordering allows you to retrieve the populations for all counties in a given state with a single range read. You could use the tuple layer with the following functions:
A few simple models

Let's begin with a few examples of simple data models built on tuples.

Arrays

You can easily map arrays to the key-value store using tuples. To model a one-dimensional array, you can construct a key for each array element from a tuple containing the array name and index.

For example, suppose you have a year's worth of average temperature data indexed by an integer ranging from 1 to 365 representing the day. You could then construct keys from tuples of the form \( (\text{"temps2012"}, \text{day}) \).

To set and get array elements with this technique, you can use Python functions such as:

```python
@fdb.transactional
def arraySet(tr, array, index, value):
    tr[fdb.tuple.pack((array, index))] = str(value)

@fdb.transactional
def arrayGet(tr, array, index):
    return tr[fdb.tuple.pack((array, index))]

@fdb.transactional
def addTemp(tr, day, temp):
    arraySet(tr, "temps2012", day, temp)

@fdb.transactional
def getTemp(tr, day):
    return int(arrayGet(tr, "temps2012", day))
```

This approach has a few nice properties:

- It can be extended to multidimensional arrays simply by adding additional array indexes to the tuples.
- Unassigned elements consume no storage, so sparse arrays are stored efficiently.

The tuple layer makes these properties easy to achieve, and most well-designed data models using tuples will share them.

An array can only have a single value for each index. Likewise, the key-value store can only have a single value for each key. The simple mapping above takes advantage of this correspondence to store the array value as a physical value. In contrast, some data structures are designed to store multiple values. In these cases, data models can store the logical values within the key itself, as illustrated next.

Multimaps

A multimap is a generalization of an associative array in which each key may be associated with multiple values. Multimaps are often implemented as associative arrays in which the values are sets rather than primitive data types.

Suppose you have a multimap that records student enrollment in classes, with students as keys and classes as values. Each student can be enrolled in more than one class, so you need to map the key-value pairs of the multimap (with their multiple values) to the database. A simple approach is to construct a key from a tuple of the form \( (\text{"enrollment"}, \text{student}, \text{className}) \) for each class in which a student is enrolled. Each class will generate a unique key, allowing as many classes as needed. Moreover, all the data in the multimap will be captured in the key, so you can just use an empty string for its value. Using this approach, you can add a class for a student or get all the student's classes with the Python functions:

```python
@fdb.transactional
def addStudent(tr, student, className):
    tr[fdb.tuple.pack(("enrollment", student, className))] = ""

@fdb.transactional
def getStudents(tr, className):
    return tr.keys(fdb.tuple.pack(("enrollment", className)))
```
You can easily use tuples to store data in tabular form with rows and columns. The most common data model for a table is to make each cell in the table a key-value pair. To do this, a composite key is constructed from a tuple containing the row and column identifiers. As with the above array model, unassigned cells in tables constructed using this technique will consume no storage, so sparse tables can be stored efficiently. As a result, a table can safely have a very large number of columns.

You can make your model row-oriented or column-oriented by placing either the row or column first in the tuple, respectively. Because the lexicographic order sorts tuple elements from left to right, access is optimized for the element placed first. Placing the row first makes it efficient to read all the cells in a particular row; reversing the order makes reading a column more efficient.

For example, we could implement the common row-oriented version in Python as follows:

```python
@fdb.transactional
def tableGetRow( tr, table, row ):
    cols = {}
    for k,v in tr[ fdb.tuple.range( (table, row, ) ) ]:
        t, r, c = fdb.tuple.unpack(k)
        cols[c] = v
    return cols
```

## Sub-keyspaces

One of the simplest ways to exploit the ordering of tuple elements is to define sub-keyspaces. As a best practice, you should always use at least one sub-keyspace to as a namespace (http://en.wikipedia.org/wiki/Namespace_(computer_science)) for your application data.

Moreover, your application will probably have multiple kinds of data to store, and it’s a good idea to separate them into different sub-keyspaces. The use of distinct sub-keyspaces will allow you to avoid conflicts among keys as your application grows. Establishing a new sub-keyspace can easily be accomplished using tuples: Just make the first element of the tuple an identifier for the sub-keyspace.

For example, if your application tracks profile data for your users, you could store the data in a sub-keyspace where “user” is the first element of each tuple. Likewise, system data could be stored in the “system” sub-keyspace. Within each sub-keyspace, the other data modeling techniques can be applied using the remainder of the tuple. In Python, the backend functions might look something like:

```python
@fdb.transactional
def multiGet( tr, multimap, index, value ):
    return [ fdb.tuple.unpack(k)[-1] for k,v in pairs ]
```

The `fdb.tuple.range()` function returns all keys that encode tuples with the specified tuple as a prefix, in this case, `(multimap, index)`. The `[-1]` extracts the last element of the tuple unpacked from the key, which in this case will encode a class.

As this model for multimaps illustrates, data that is treated as a value at one level may be mapped to a key in the database. (The reverse may also occur, as shown in the discussion of indirection below.) Data modeling in FoundationDB is not dictated by how your data is represented in your programming language.

### Tables

You can easily use tuples to store data in tabular form with rows and columns. The most common data model for a table is to make each cell in the table a key-value pair. To do this, a composite key is constructed from a tuple containing the row and column identifiers. As with the above array model, unassigned cells in tables constructed using this technique will consume no storage, so sparse tables can be stored efficiently. As a result, a table can safely have a very large number of columns.

You can make your model row-oriented or column-oriented by placing either the row or column first in the tuple, respectively. Because the lexicographic order sorts tuple elements from left to right, access is optimized for the element placed first. Placing the row first makes it efficient to read all the cells in a particular row; reversing the order makes reading a column more efficient.

For example, we could implement the common row-oriented version in Python as follows:

```python
@fdb.transactional
def tableSetCell( tr, table, row, column, value ):
    tr[ fdb.tuple.pack( (table, row, column) ) ] = str(value)
```

```python
@fdb.transactional
def tableGetCell( tr, table, row, column ):
    return tr[ fdb.tuple.pack( (table, row, column) ) ]
```

```python
@fdb.transactional
def tableSetRow( tr, table, row, cols ):
    del tr[ fdb.tuple.range( (table, row, ) ) ]
    for c,v in cols.iteritems():
        tableSetCell( tr, table, row, c, v )
```

```python
@fdb.transactional
def tableGetRow( tr, table, row ):
    cols = {}
    for k,v in tr[ fdb.tuple.range( (table, row, ) ) ]:
        t, r, c = fdb.tuple.unpack(k)
        cols[c] = v
    return cols
```
Entity-relationship models

Entity-relationship models are often used to describe a database at various levels of abstraction. In this methodology, a logical data model consisting of entities, attributes, and relationships is defined before mapping it to a physical data models specifying keys and other implementation features. Entity-relationship models can be easily modeled in FoundationDB using tuples.

Attributes

Suppose you’re storing entity-relationship data for users in an "ER" sub-keyspace. You might identify each entity with a unique identifier and define a key for each attribute with the tuple ("ER", entityID, attribute). You could then store the user's region using the Python functions:

```python
@fdb.transactional
def addUserRegion(tr, userID, region):
    addAttributeValue(tr, userID, "region", region)
```

Relationships

Using the pattern we saw above with multimaps, you can store relationships and related entities as an element of the key and use an empty string as the physical value. Suppose your users can belong to one or more groups. To add a user to a group or retrieve all groups to which a user belongs, you can use the Python functions:

```python
@fdb.transactional
def getUserGroups(tr, userID):
    return getRelatedEntities(tr, userID, "belongsTo")
```

Indexes

A common technique is to store the same data in different ways to allow efficient retrieval for multiple use cases, effectively creating indexes. This technique is especially useful when there are many more reads than writes. For example, you may find it most convenient to store user data based on userID but sometimes need to retrieve users based on their region. An index allows this retrieval to be performed efficiently.

Indexes can have a very simple tuple structure consisting of an identifying sub-keyspace, the relationship being indexed, and a value: (subKeyspaceForIndex, relationship, value). Placing the relationship before the value is crucial: it allows efficient retrieval of all the associated values with a single range read.

With FoundationDB’s transactions, you can easily build an index and guarantee that it stays in sync with the data: just update the index in the same transaction that updates the data. For example, suppose you’d like to add an index to efficiently look up users by region. You can augment the Python function addUser with the index and add a new function for retrieval:
To retrieve first-degree connections ("friends") and second-degree connections ("friends of friends"), you can use:

```python
@fdb.transactional
def getFriends( tr, userID ):
    friendKVs = tr[ fdb.tuple.range( ("users", userID, "friendsWith") ) ]
    return [ fdb.tuple.unpack(k)[1] for k,v in friendKVs ]

@fdb.transactional
def getFriendsOfFriends( tr, userID ):
    secondDegree = []
    for f in getFriends(tr, userID):
        secondDegree = secondDegree + getFriends(tr,f)
    secondDegree = list(set(secondDegree))
    secondDegree.remove(userID)
    return secondDegree
```

Alternately, your application might allow a user to "follow" other users without their necessarily reciprocating, resulting in a directed graph. To model this graph while allowing efficient retrieval in either link direction, you can store distinct relationships for "hasFollowed" and "followedBy":

```python
@fdb.transactional
def addUser( tr, userID, name, region ):
    tr[ fdb.tuple.pack( ("user", userID) ) ] = str(name)
    tr[ fdb.tuple.pack( ("regionIdx", region, userID) ) ] = ""

@fdb.transactional
def getUsersInRegion( tr, region ):
    items = tr[ fdb.tuple.range( ("regionIdx", region) ) ]
    return [ fdb.tuple.unpack(k)[1] for k,v in items ]
```
Hierarchies

Many applications work with hierarchical data represented by nested dictionaries or similar composite data types. Such data is often serialized to or deserialized from a format such as JSON or XML. Looking at a hierarchical object as a tree, you can use a tuple to represent the full path to each leaf (sometimes called a “materialized path”). By storing each full path as a key, you get an index for each leaf. FoundationDB can then efficiently retrieve any individual piece of data or entire sub-tree.

For example, suppose you have hierarchical data such as the following nested dictionaries and lists:

```
{ 'user': { 'jones': { 'friendOf': 'smith',
                     'group': [ 'sales', 'service' ]},
               'smith': { 'friendOf': 'jones',
                          'group': [ 'dev', 'research' ]}}}
```

To distinguish the list elements from dictionary elements and preserve the order of the lists, you can just include the index of each list element before it in the tuple. Using this technique, the data above would be converted to the following tuples:

```
[ ( 'user', 'jones', 'friendOf', 'smith'),
  ( 'user', 'jones', 'group', 0, 'sales'),
  ( 'user', 'jones', 'group', 1, 'service'),
  ( 'user', 'smith', 'friendOf', 'jones'),
  ( 'user', 'smith', 'group', 0, 'dev'),
  ( 'user', 'smith', 'group', 1, 'research')]
```

Suppose you’d like to use this representation to implement a nested keyspace, i.e., a key-value store in which values can themselves be nested dictionaries or lists. Your application receives a stream of serialized JSON objects in which different objects may contain data about the same entities, so you’d like to store the data in a common nested keyspace.

You can deserialize the data using Python’s standard `json` module, generate the corresponding set of paths as tuples, and store each tuple in a “hier” sub-keyspace:

```
import json, itertools

EMPTY_OBJECT = -2
EMPTY_ARRAY = -1

def to_tuples(item):
  if item == {}:
    return [(EMPTY_OBJECT, None)]
  elif item == []:
    return [(EMPTY_ARRAY, None)]
  elif type(item) == dict:
    return [(k,) + sub for k, v in item.items() for sub in to_tuples(v)]
  elif type(item) == list:
    return [(k,) + sub for k, v in enumerate(item) for sub in to_tuples(v)]
  else:
    return [(item,)]

@fdb.transactional
def insertHier(tr, hier):
  if type(hier) == str:
    hier = json.loads(hier)
  for tup in to_tuples(hier):
    tr[fdb.tuple.pack( ("hier",) + tup )] = ""
```

You can then retrieve any sub-tree from the nested keyspace by giving the partial path to its root. The partial path will just be a tuple that your query function uses as a key prefix for a range read. For example, to retrieve the data for ‘smith’ from the hierarchy above, you would use ("user", "smith").

The retrieved data will be a list of tuples. The final step before returning the data is to convert it back to a
Suppose you’d like to use the above representation to implement a simple document-oriented data model. As before, your application receives serialized data in JSON, only now you’d like to store each JSON object as an independent document. To do so, you just need to ensure that each tuple created for that object is stored with a unique identifier for the document. If a `doc_id` has not already been supplied, you can randomly generate one.

To store a path, you can construct a composite key in a "doc" sub-keyspace, with the `doc_id` as the next element, followed by the remainder of the path. You can store the leaf (the last element of the tuple) as the value, which enables storage of larger data sizes (see Performance guidelines for keys and values):

```python
@fdb.transactional
def getSubHier(tr, prefix):
    return from_tuples([fdb.tuple.unpack(k)[1:] for k,v in tr[ fdb.tuple.range( "hier", prefix ) ]])
```

To retrieve an entire document by its `doc_id`, you just reverse the process, reading all key-value pairs in the "doc" sub-keyspace with the given `doc_id`, and converting the resulting list of tuples back to a nested data structure:

```python
@fdb.transactional
def getDoc(tr, doc_id):
    return from_tuples([fdb.tuple.unpack(k)[2:]+fdb.tuple.unpack(v) for k,v in tr[ fdb.tuple.range( 'doc', doc_id ) ]])
```

**Indirection**

It is sometimes beneficial to add a level of indirection to a data model. Instead of using key-value pairs to directly store application data, you can instead store a reference to that data. This approach can be used to model any data structure that would normally use references. You just need to perform any modifications to the data structure in a transaction that leaves it in a consistent state.

Suppose you want to maintain data in a singly linked list. The application data can use a tuple structure like those of single-valued relationships. Links will be similar but will use node identifiers as their values. Here is an example of removing the next node from the list:

```python
def from_tuples( tuples):
    first = tuples[0]  # The first tuple will tell us what kind of object we have
    if len(first) == 1: return first[0]  # Primitive value
    if first == (EMPTY_OBJECT, None): return []
    if first == (EMPTY_ARRAY, None): return []

    # For an object or array, we need to group the tuples by their first element
    groups = [ list(g) for k,g in itertools.groupby( tuples, lambda t:t[0] ) ]
    if first[0] == 0:  # array
        return [ from_tuples([t[1:] for t in g]) for g in groups ]
    else:  # object
        return dict( (g[0][0], from_tuples([t[1:] for t in g])) for g in groups )

@fdb.transactional
def insertDoc(tr, doc):
    if type(doc) == str:
        doc = json.loads(doc)

    if not "doc_id" in doc:
        doc["doc_id"] = random.randint(0, 100000000)

    for tup in to_tuples(doc):
        tr[ fdb.tuple.pack( ("doc",doc["doc_id"]), tup[:-1] ) ] = fdb.tuple.pack((tup[-1],))

    return doc["doc_id"]
```
# data model
# ("node", nodeID, attribute ) = values
# ("node", nodeID, "next") = nextNodeID

def nextNodeKey(nodeID):
    return fdb.tuple.pack( ("node", nodeID, "next") )

@fdb.transactional
def removeNextNode(tr, nodeID):
    nextID = tr[ nextNodeKey(nodeID) ]
    if nextID != "":
        nextNextID = tr[ nextNodeKey(nextID) ]
        tr[ nextNodeKey(nodeID) ] = nextNextID
        del tr[ fdb.tuple.range(("node", nextID)) ]

FoundationDB's transactional guarantees ensure that, even when multiple clients are concurrently
modifying the same linked list, the structure will be maintained in a consistent way.

Performance guidelines for keys and values

How you map your application data to keys and values can have a dramatic impact on performance. Below
are some guidelines to consider as you design a data model. (For more general discussion of performance
considerations, see Performance considerations (developer-guide.html#developer-guide-performance-
considerations).)

- Structure keys so that range reads can efficiently retrieve the most frequently accessed data.
  - If you perform a range read that is, in total, much more than 1 kB, try to restrict your range as much
    as you can while still retrieving the needed data.

- Structure keys so that no single key needs to be updated too frequently, which can cause transaction
  conflicts.
  - If a key is updated more than 10-100 times per second, try to split it into multiple keys.
  - For example, if a key is storing a counter, split the counter into N separate counters that are
    randomly incremented by clients. The total value of the counter can then read by adding up the N
    individual ones.

- Keep key sizes small.
  - Try to keep key sizes below 1 kB. (Performance will be best with key sizes below 32 bytes and cannot
    be more than 10 kB.)
  - If your key sizes are above 1 kB, try either to move data from the key to the value, split the key into
    multiple keys, or encode the parts of the key more efficiently (remembering to preserve any
    important ordering).

- Keep value sizes moderate.
  - Try to keep value sizes below 10 kB. (Value sizes cannot be more than 100 kB.)
  - If your value sizes are above 10 kB, consider splitting the value across multiple keys.
  - If you read values with sizes above 1 kB but use only a part of each value, consider splitting the
    values using multiple keys.
  - If you frequently perform individual reads on a set of values that total to fewer than 200 bytes, try
    either to combine the values into a single value or to store the values in adjacent keys and use a
    range read.

A data modeling example: class scheduling

The class scheduling tutorial (tutorial.html) provides an example of data modeling for a sample application
using the tuple layer. It uses techniques applicable to all of the languages (api-reference.html) supported by
FoundationDB.

© 2013 FoundationDB. Privacy policy. (/about/privacy/)