PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53(c).

INVENTOR(S)

Given Name Family Name or Surname	Residence
Sumeet Sohan Singh	Saratoga, California

TITLE OF THE INVENTION (500 characters max):

A method of storing, retrieving, arranging in storage, managing and asynchronously replicating data.

CORRESPONDENCE ADDRESS

Direct all correspondence to:

___ The address corresponding to Customer Number: ________________

OR

X Firm or Individual Name: Sumeet Sohan Singh

Address: 18518 Montpere Way
City: Saratoga
State: California
Zip: 95070
Country: United States (US)
Telephone: 650-520-8656
Email: sumeet@singhonline.info

ENCLOSED APPLICATION PARTS (check all that apply)

___ Application Data Sheet. See 37 CFR 1.76___ CD(s), Number of CDs _____________

___ Drawing(s) Number of Sheets _______ ___ Other (specify): _____________

_X__ Specification (e.g. description of the invention) Number of Pages ____13____

METHOD OF PAYMENT OF THE FILING FEE AND APPLICATION SIZE FEE FOR THIS PROVISIONAL APPLICATION FOR PATENT

X Applicant claims small entity status. See 37 CFR 1.27. **TOTAL FEE AMOUNT ($)**

___ A check or money order made payable to the *Director of the United States Patent and Trademark Office* is enclosed to cover the filing fee and application size fee (if applicable) **$125**

___ Payment by credit card. Form PTO-2038 is attached.

___ The Director is hereby authorized to charge the filing fee and application size fee (if applicable) or credit any overpayment to Deposit Account Number: ___________________________________.

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

PROVISIONAL APPLICATION COVER SHEET

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

X No.

___ Yes, the name of the U.S. Government agency and the Government contract number are: ________________________

WARNING:

Petitioner/applicant is cautioned to avoid submitting personal information in documents filed in a patent application that may contribute to identity theft. Personal information such as social security numbers, bank account numbers, or credit card numbers (other than a check or credit card authorization form PTO-2038 submitted for payment purposes) is never required by the USPTO to support a petition or an application. If this type of personal information is included in documents submitted to the USPTO, petitioners/applicants should consider redacting such personal information from the documents before submitting them to the USPTO. Petitioner/applicant is advised that the record of a patent application is available to the public after publication of the application (unless a non-publication request in compliance with 37 CFR 1.213(a) is made in the application) or issuance of a patent. Furthermore, the record from an abandoned application may also be available to the public if the application is referenced in a published application or an issued patent (see 37 CFR 1.14). Checks and credit card authorization forms PTO-2038 submitted for payment purposes are not retained in the application file and therefore are not publicly available.

SIGNATURE ____ Date June 08, 2013

TYPED or PRINTED NAME Sumeet Sohan Singh REGISTRATION NO. _________
(if applicable)

TELEPHONE 650 520 8656 Docket Number: ________________

Privacy Act Statement

The **Privacy Act of 1974 (P.L. 93-579)** requires that you be given certain information in connection with your submission of the attached form related to a patent application or patent. Accordingly, pursuant to the requirements of the Act, please be advised that: (1) the general authority for the collection of this information is 35 U.S.C. 2(b)(2); (2) furnishing of the information solicited is voluntary; and (3) the principal purpose for which the information is used by the U.S. Patent and Trademark Office is to process and/or examine your submission related to a patent application or patent. If you do not furnish the requested information, the U.S. Patent and Trademark Office may not be able to process and/or examine your submission, which may result in termination of proceedings or abandonment of the application or expiration of the patent.

The information provided by you in this form will be subject to the following routine uses:

1. The information on this form will be treated confidentially to the extent allowed under the Freedom of Information Act (5 U.S.C. 552) and the Privacy Act (5 U.S.C 552a). Records from this system of records may be disclosed to the Department of Justice to determine whether disclosure of these records is required by the Freedom of Information Act.
2. A record from this system of records may be disclosed, as a routine use, in the course of presenting evidence to a court, magistrate, or administrative tribunal, including disclosures to opposing counsel in the course of settlement negotiations.
3. A record in this system of records may be disclosed, as a routine use, to a Member of Congress submitting a request involving an individual, to whom the record pertains, when the individual has requested assistance from the Member with respect to the subject matter of the record.
4. A record in this system of records may be disclosed, as a routine use, to a contractor of the Agency having need for the information in order to perform a contract. Recipients of information shall be required to comply with the requirements of the Privacy Act of 1974, as amended, pursuant to 5 U.S.C. 552a(m).
5. A record related to an International Application filed under the Patent Cooperation Treaty in this system of records may be disclosed, as a routine use, to the International Bureau of the World Intellectual Property Organization, pursuant to the Patent Cooperation Treaty.
6. A record in this system of records may be disclosed, as a routine use, to another federal agency for purposes of National Security review (35 U.S.C. 181) and for review pursuant to the Atomic Energy Act (42 U.S.C. 218(c)).
7. A record from this system of records may be disclosed, as a routine use, to the Administrator, General Services, or his/her designee, during an inspection of records conducted by GSA as part of that agency's responsibility to recommend improvements in records management practices and programs, under authority of 44 U.S.C. 2904 and 2906. Such disclosure shall be made in accordance with the GSA regulations governing inspection of records for this purpose, and any other relevant (i.e., GSA or Commerce) directive. Such disclosure shall not be used to make determinations about individuals.
8. A record from this system of records may be disclosed, as a routine use, to the public after either publication of the application pursuant to 35 U.S.C. 122(b) or issuance of a patent pursuant to 35 U.S.C. 151. Further, a record may be disclosed, subject to the limitations of 37 CFR 1.14, as a routine use, to the public if the record was filed in an application which became abandoned or in which the proceedings were terminated and which application is referenced by either a published application, an application open to public inspection or an issued patent.
9. A record from this system of records may be disclosed, as a routine use, to a Federal, State, or local law enforcement agency, if the USPTO becomes aware of a violation or potential violation of law or regulation.

PROVISIONAL PATENT APPLICATION

Invention Type

This invention is an invention of the following type(s):
* Software

Invention Title

The title of the invention is: A method of storing, retrieving, arranging in storage, managing and asynchronously replicating data.

Background

Definition: For the purpose of this document, the term *information system* or *system* is defined as the application of electronic, computer and telecommunications equipment and software to store, retrieve, transmit and manipulate data. It includes all computer programs, all computer hardware, all computer software, all software applications, all telecommunication equipment, devices and software as well as systems comprised thereof.

Definition: For the purpose of this document, the term *computer application* shall mean *information system* as defined above.

Definition: For the purpose of this document data-store shall mean a repository of data.
Note: For the purpose of this document the term *data* shall include meta-data.

Definition: For the purpose of this document a data-record shall mean a piece of data identified by a single key.

Definition: This patent application describes a data management system which shall be called *FragmentDB* in this document.

Definition: DMS shall mean Data Management System. It includes all systems that perform storing, retrieving, arranging, managing and asynchronously replicating data.

There are several information systems (e.g. software applications) that store and retrieve data and make the data accessible on multiple devices. Examples of said devices are:
1. personal computers,
2. mobile phones,
3. electronic tablet devices,
4. notebooks,
5. netbooks,
6. laptops,
7. set-top-boxes for entertainment,
8. smart-TVs,
9. gaming consoles,
10. streaming media devices,
11. network storage devices,
12. ‘cloud’ drives,
13. electronic entertainment devices such as CD, DVD or Blue Ray players
14. Smart Cards
15. Any device that stores information in electronic format

Examples of said information systems are:

1. Password manager applications
2. Note taking applications
3. Applications that store personal contact information
4. Mobile phone applications with user-editable settings or configuration
5. Any information system that accesses data can use the invention described under this patent application

Examples of the types of said data are:
1. Web bookmarks
2. Website URLs and corresponding usernames and passwords (Web logins)
3. Application or service credentials (passwords, passphrases, security codes, PINs)
4. credit card numbers
5. bank account numbers
6. ATM PINs
7. personal notes
8. application settings
9. etc.

Target Applications
This invention targets computer applications that adhere to the following data model:

DATA-MODEL-TYPE

1. Data Structure Characteristics
    a. Data Storage: Data is stored kept in a persistent storage medium.
    b. Stored data is modeled as a dictionary, also called map or associative array,
    c. Data is stored in multiple data-store replicas that need to be periodically synchronized.
    d. Remote Data Store: The data-store may be located on a remote network location such on a LAN, WAN, an ad-hoc network or a Bluetooth network. The data-store may also be located on a local computer drive. Examples are:
        i. Data located on a Network Attached Storage (NAS)
        ii. Data located on a Windows Share, Windows Homegroup or Windows Workgroup etc.
        iii. Data located on a local drive, thumb-drive etc.
        iv. Data located on a peer-to-peer network
        v. Data located on a Bluetooth or NFC device
    e. Cloud Data Store: The data may be located on a file-system that is synchronized by a third-party e.g. Dropbox.
2. Data Access Characteristics
    a. Retrieval: Data is retrieved from one or more data-stores,
    b. Updates: Data of one or more data-stores is edited: that is, create, update or delete pieces of data, provided the following condition is met:
        i. Multiple data-stores may be updated in parallel including updates to the same data-record but the new record values are independent of pre-existing data. For example, incrementing a count or updating statistical data does not meet this requirement. However, a wide variety of practical cases fall into this category. Personal data such as bookmarks, passwords, usernames, notes, settings etc. all fall into this category because updates to these values does not depend on pre-existing data.
    c. Concurrent Access to one Data Store: Multiple system components or multiple systems such as multiple devices, multiple computer programs etc. may need to access the same data-store concurrently or in parallel barring the restrictions noted above,
    d. Parallel Access to Different Data Stores: Multiple system components or multiple systems may access data-store replicas either sequentially or in parallel barring restrictions noted above. The replicas would later be synchronized.
    e. Multi-master replication is desirable,

    f. History of data updates is desirable,
3. Data Size Characteristics
   Every dictionary that is accessed by the application must be fully loaded into memory. Therefore this technology is suited for small data sizes such as those already mentioned.

Definition: For the purposes of this document the term *personal data dictionary* shall mean data that is modeled as described under section DATA-MODEL-TYPE above and whose size is and is expected to stay small enough over time in order for an application to practically store it using the dictionary data management system described herein. Specific data-size cannot be quantified for this definition because it will vary on a case-by-case basis depending upon 1) the specific requirements on the application (e.g. how quickly data query results are required) 2) the complexity of the data-model 3) the complexity of operations on the data and 4) the amount of information processing resources available to the application. For example, an application (first application) that has access to a cluster of very fast data-center server grade CPUs, RAM and network resources will be able to process huge amounts of data as compared to an application executing on a mobile phone (second application). If all other requirements on the two applications were equal, then a much larger data-size would qualify as *personal data dictionary* for the first application than for the second application. Hence, a data-size of several gigabytes may qualify as a *personal data dictionary* for the first application, whereas a few hundred megabytes of data may be too much for a mobile phone and therefore not qualify as *personal data dictionary* for its case. Additionally, as computers and devices become more and more powerful, the upper-bound on the size of *personal data dictionary* will keep sliding upwards. Therefore no attempt is made here to quantify the size of *personal data dictionary*.

Invention Summary

Several data storage, replication and synchronization technologies are available today, but they all suffer from great complexity and corresponding cost, which while unavoidable for several classes of applications, can be entirely avoided for personal data dictionary applications.

This invention introduces a novel data management system called FragmentDB that is free from the above alluded complexities and costs.

FragmentDB is a light-weight client-only personal data dictionary management system with multi-master asynchronous passive data replication, portability and time-based automatic conflict resolution.

Invention Description

What does the invention do?

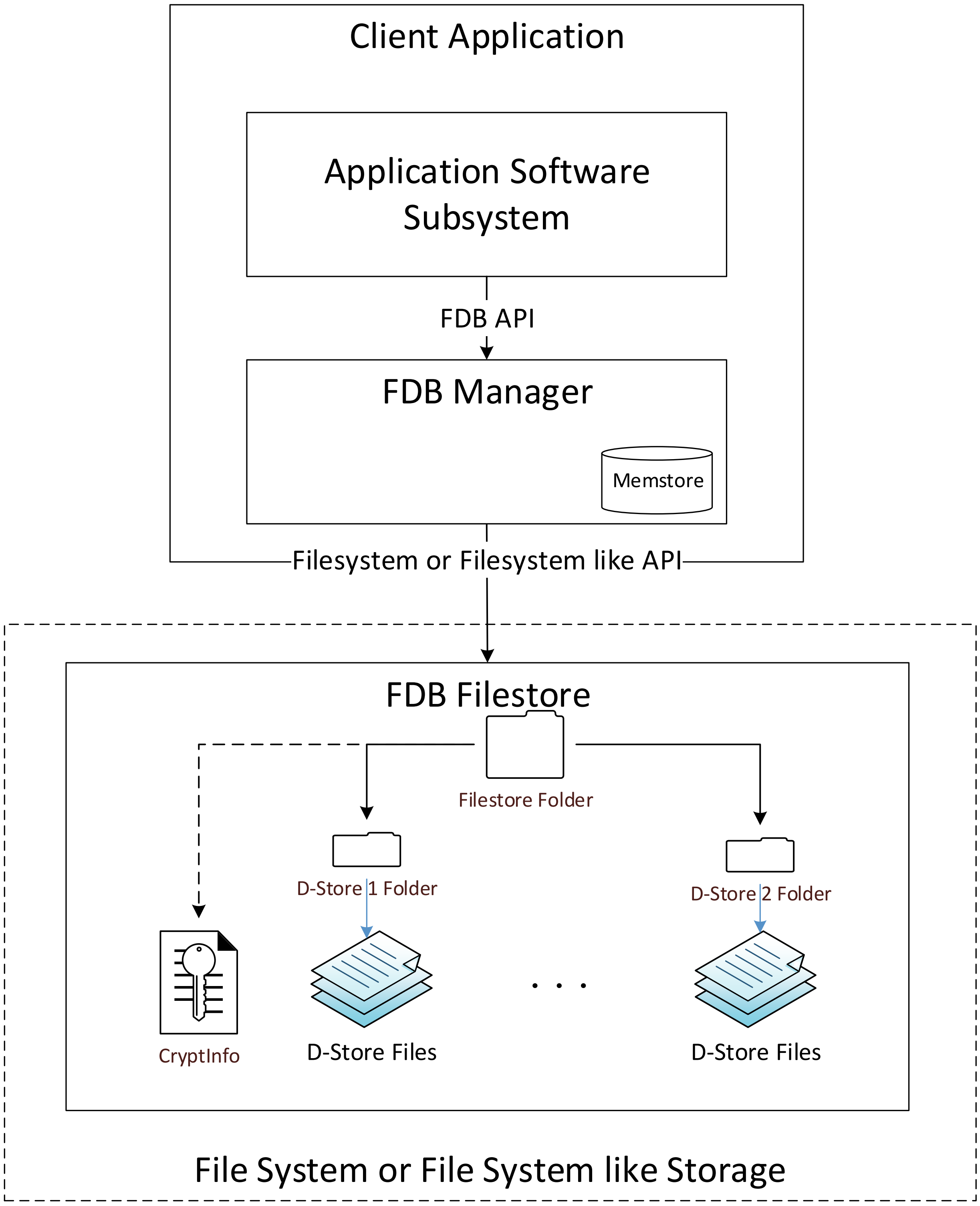


Figure 1 FragmentDB System Architecture

The invention is a data management system called FragmentDB.

FragmentDB is comprised of the following components of Figure 1:

1. FDB Manger. It is the FragmentDB's software subsystem. It exposes a software API which can be invoked by a software application

2. Memstore. It is a copy of the dictionary in the computing device's memory accessible to the FDB Manager.
3. FDB API. The application software uses this API to interact with FragmentDB.
4. FDB filestore and everything shown inside it.
    a. Filestore Root is a container of all of a filestore's data files. It contains multiple D-Stores and optionally cryptographic information.
    b. A D-Store Folder such as D-Store 1 contains all data pertaining to one conceptual data dictionary. The data is stored in one or more files under the folder. There can be any number of D-Store folders. Each is associated with exactly one dictionary.
    c. Each dictionary is a collection of data-records of one type – for e.g. a web-credentials data record containing the URL, username and password of a website. Data records are conceived as a collection of one or more fields. One or more fields constitute its 'key' and the remaining constitute its 'value'.
    d. CryptInfo: This is a file that contains a cryptography key that is used to encrypt and decrypt action records. It also contains all other information necessary for encryption and decryption – e.g. the cipher algorithm used and its parameters, key size, initialization vector size etc. The file may be kept inside the Filestore Folder or outside it.

FragmentDB enables an application to store and retrieve any number of data-dictionaries in a file system. Each dictionary is conceptually a collection of data-records. The application can efficiently lookup data-records by their key. It can insert new data records and change and delete existing ones.

Any number of dictionaries can be created and deleted from storage.

FragmentDB maintains historical values of each data-record. The client application can retrieve these values. Additionally, it may delete the entire history can be deleted as well.

Multiple applications may access the same FDB filestore instance in parallel.

Multiple FDB filestores can be synchronized at any point in time. This will cause each instance to obtain data updates from the other. Thus each instance will receive all updates that it didn't already have. Duplicated updates will not affect the state of the data. At the end of synchronization, all participating filestore instances will have a consolidated set of updates and thus they all will have identical data. Moreover the instances would have been restored to the same state that an instance would've reached had all the data-updates been directly performed on it from the beginning. That is, it would be as if none of the instances had ever missed an update.

Definition: Synchronization described above is called multi-way synchronization. Typically only two filestores participate in synchronization, in which case it is called two-way synchronization.

Definition: Filestore instances that are synchronized or are intended to be synchronized are called replicas.

Synchronization may also be performed one-way. In this case only one instance receives all updates from all others.

If synchronization is desired, then there is a restriction on the types of updates that can be performed on the replicas. Filestore replicas may be updated in parallel including updates to the same data-record but the new record values must be independent of pre-existing data. For example, incrementing a counter does not meet this criterion. However, a wide variety of practical cases fall into this category. Personal data such as bookmarks, passwords, usernames, notes, settings etc. all fall into this category because updates to these values does not depend on pre-existing data.

The filestore may be located 'locally' as on a computer's hard drive or memory stick or it may be located on a device on the network – LAN, WAN or the internet. The filestore may also be located in a third-party synchronized folder – e.g. a Dropbox folder.

How does the invention work?

<u>Novel Data Model</u>
FragmentDB does not store or manipulate data-records. It only stores change records, called action-records or simply, actions. The basic innovation in data-model is the following:

1. Structures (i.e. data-records) are viewed as the result of a series of events pertaining to the data-record. Events relevant to a data-record are data-record creation, updates, deletion and permanent deletion. Other actions – e.g. 'move' - may be added later as well.
2. A data-record never gets stored or transferred, rather the history of events pertaining to that data-record. Each such event is called an 'action'. FragmentDB Filestore is a collection of such actions. Actions are the fundamental building block of FragmentDB.
3. There are four basic actions pertaining to a data-record:
    a. 'Assert' action. This action is generated for creating or updating a data-record. In data-base parlance this action causes a 'record level change'. 'field level changes' are not supported. If field-level changes are desired then the data-schema should be completely de-normalized into multiple data-types (and corresponding data dictionaries) with each data-type having the original key but only one value field from the original. The application can later join the various de-normalized data-records in order to create the original. This is a miniscule trade-off compared to the enormous benefit of reliable and extremely simple asynchronous and passive master-master replication that one gets in return. Also, the computing resources available to modern day devices are large enough and the size of personal data small enough that this is a non-issue.
    b. 'Delete' action. This action is generated for deleting a record. However, the history of the record is kept (in case the user may want to 'undelete'). The record may be viewed as being 'in the trash'.
    c. 'Permanent Delete' action. This action causes record deletion but in addition, also causes its entire history to be purged at the earliest opportunity (during compaction operation).
    d. More action types can be added as well to denote some other outcome.
4. A data-record is never retrieved. Rather, it is reconstructed by replaying the actions in the history of the data-record's life. Given the above, FragmentDB may also be called:
    a. 'action-base',
    b. 'event-base' or
    c. 'replay DB'.
5. Creating a new data-record or making changes to an existing one is achieved by inserting a new 'assert' action along with the record key and value.
6. Deleting a data-record is achieved by inserting a delete action along with the record's key.

Replaying actions (change-logs) to construct data is an expensive process in traditional data-models for large and complex data. However, the typical size, complexity and usage pattern of personal data is small enough and the computing resources available to personal devices large enough for this to not be an issue in practice. Furthermore, all records are read only once initially when a Dictionary File Store is loaded into memory. After that they can be grouped by key into Action History data structures and can be saved in an in-memory map data-structure with efficient read/write performance. See Figure 3.

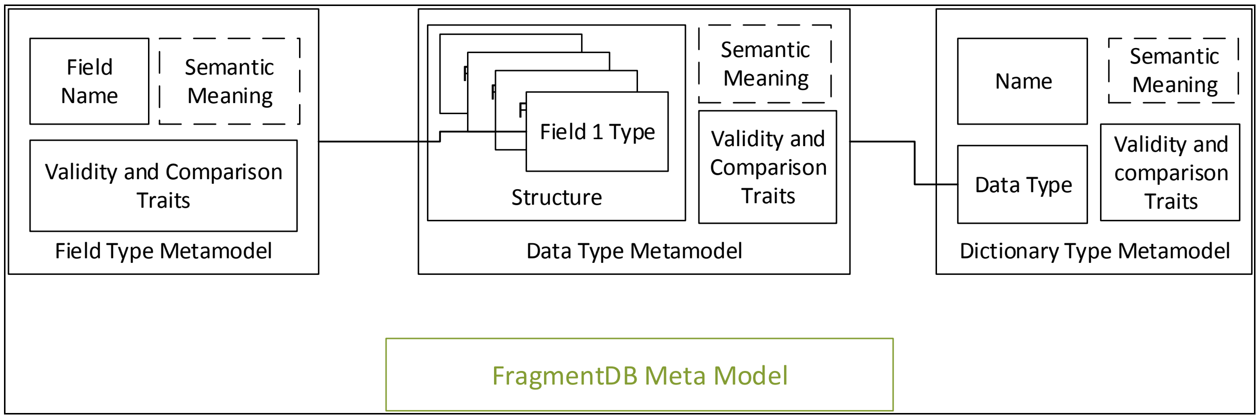
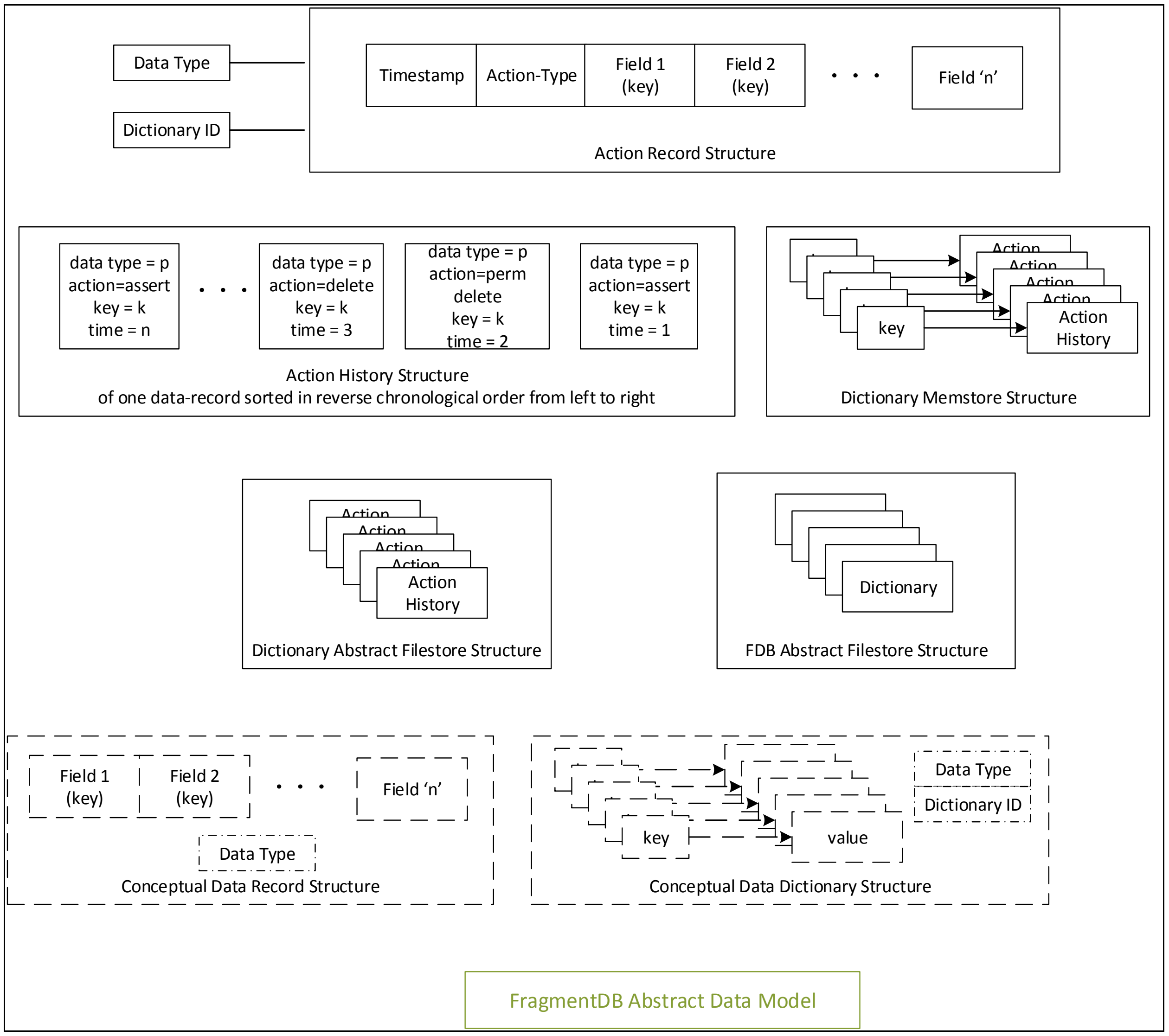


Figure 2 FragmentDB Meta Model



Data Type
Dictionary ID
Timestamp
Action-Type
Field 1 (key)
Field 2 (key)
. . .
Field 'n'
Action Record Structure
data type = p action=assert key = k time = n
. . .
data type = p action=delete key = k time = 3
data type = p action=perm delete key = k time = 2
data type = p action=assert key = k time = 1
Action History Structure
of one data-record sorted in reverse chronological order from left to right
key
Action History
Dictionary Memstore Structure
Action History
Dictionary Abstract Filestore Structure
Dictionary
FDB Abstract Filestore Structure
Field 1 (key)
Field 2 (key)
. . .
Field 'n'
Data Type
Conceptual Data Record Structure
key
value
Data Type
Dictionary ID
Conceptual Data Dictionary Structure
FragmentDB Abstract Data Model

Figure 3 FragmentDB Abstract Data Model

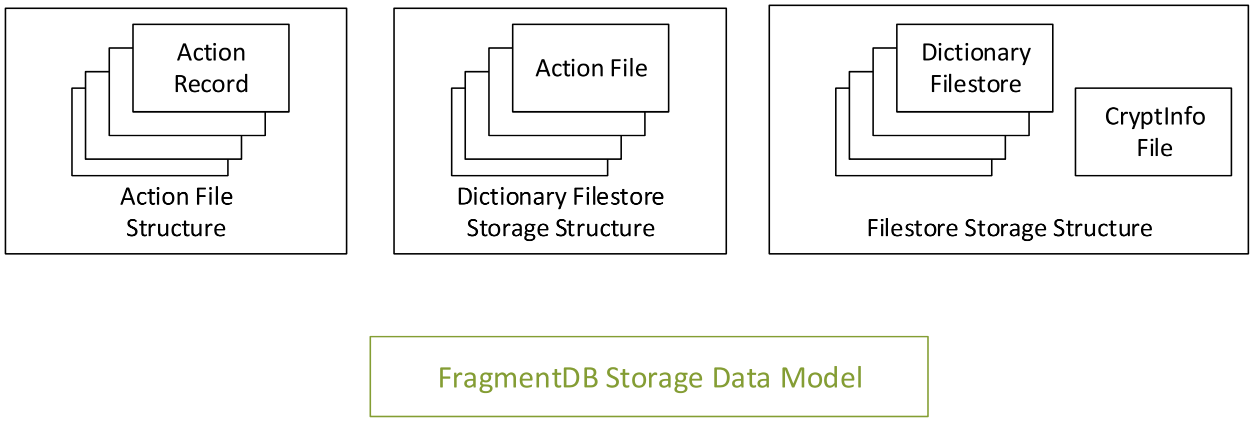

Action Record
Action File Structure
Action File
Dictionary Filestore Storage Structure
Dictionary Filestore
CryptInfo File
Filestore Storage Structure
FragmentDB Storage Data Model

Figure 4 FragmentDB Storage Data Model

FragmentDB API allows a client application to invoke the following operations. Description of operations, and steps to perform them are described next.

Load DB into Memstore
A fragment DB instance needs to be loaded into memory before it can be used. The typical usage is to load the DB into memory and use it until it is no longer needed, at which time it can be unloaded from memory.
This operation requires the following parameters:

1. Location of the FDB Filestore Folder

Steps:
For each dictionary folder under the FDB filestore in the DB, do the following:

1. Read all actions from all files in the dictionary filestore folder and store them in Memstore.
2. If efficiency is desired, group all actions by key into corresponding Action History objects.
3. If efficiency is desired, insert the action history objects into a map structure with record-key as the key and action history object as value.

Reload DB into memory
In order to apply updates to the DB that were concurrently made by other applications after the DB was loaded, it needs to be re-loaded.

Instead of reloading the entire DB, only the updated dictionaries may be reloaded for greater efficiency. Such dictionaries can be found by comparing the time-stamps on dictionary files with those at the time when the DB was last loaded (the old timestamps would've had to be kept in memory).

Unload DB from memory
Steps:

1. Purge all action records from Memstore.

Create Action Record
Creates an action record for insertion into DB.
Parameters Required:

1. Action Type
2. Dictionary ID
3. Data Type
4. Data Type fields (all key and value fields)
5. Current time

Steps:

1. Create a suitable in-memory record with fields shown in Figure 3.

Insert Action Record into DB
Inserts an action into the currently loaded DB.
Parameters required:
1. Action Record
2. Dictionary ID

Prerequisite: A DB should be loaded into memory.
Steps:
1. Store the action record in Memstore.
2. If needed, serialize the action record into a form suitable for storage.
3. Locate a file in the target dictionary's filestore folder that is in 'open-for-write' state. If no such file is found then create one. If the file is found but has already reached a preconfigured maximum size, then its state may be changed to 'closed-for-edits'. In this case create a new file in the 'open-for-edits' state.
4. Append the (optionally serialized) record data to the end of the file in 'open-for-write' state in the target dictionary's filestore folder using a suitable serialization/de-serialization scheme.

Retrieve Latest Action Record from DB
Retrieves latest action on the given key.
Parameters required:
1. Record Key
2. Dictionary ID

Steps:
1. Retrieve from Memstore, the latest action record for the provided key and the dictionary ID.

Retrieve Action History from DB
Retrieves latest action on the given key.
Parameters required:
1. Record Key
2. Dictionary ID

Steps:
1. Retrieve from Memstore, the all action records pertaining to the provided key and the dictionary ID.

Merge DB
Merge In DB
Merge Out DB
Create DB
Delete DB
Compact DB
Clean DB

How would someone use this invention?

Create a new data-record
1. Create a 'assert' action record as already described.
2. Insert the action record into DB as already described.

Update a data-record
Same procedure as data-record creation.

Delete a data-record
1. Create a 'delete' action record as already described.
2. Insert the action record into DB as already described.

Permanent Delete a data-record

1. Create a ‘permanent delete’ action record as already described.
2. Insert the action record into DB as already described.

Improvements on Pre-Existing Inventions

Pre-Existing data management systems with master-master asynchronous replication (such as various RDBMSs, blob-stores, key-value stores, Hadoop, Cassandra, Couch DB and Berkley DB) cater to a wide variety of complex applications and correspondingly their multi-master replication methods tend to be complex, resulting in high capital and operating costs and sometimes incorrect data. One source of complexity is the fundamental concept that a data-record is a concrete structural entity that may be created, modified or deleted. Indeed this is the definition of data or datum in the ontology of languages but it is also a structural rigidity that causes lot of complexity and inefficiencies which can be entirely avoided in *personal data dictionary* applications. In personal data dictionary applications it is possible to have a much simpler data-model immensely simplifies data management with master-master replication.

Below is a description of the relevant features and drawbacks of pre-existing methods that provide asynchronous master-master replication.

Pre-Existing Data Models

1. When a piece of data is created it is associated with an identifier, which is also a piece of data. Depending on the implementation, the identifier may either be a part of the originally created data or it may be generated separately. In most cases a unique identifier is internally generated by the data management system. The identifier is referred to by various names, e.g.:
    a. identifying field(s)
    b. key, key column(s)
    c. etc.
2. The identifier and the data that it referrers to are collectively known as a data record.
3. In this document the identifier of a record is called ‘key’ and the remaining data in the record is called its ‘value’ and the data record is simply called ‘record’.
4. In addition to the key and value, the data-record also includes a record version which is typically an integral number. When a record is generated for the first time it is given an initial version number (e.g. 1). The version number is then incremented each time the record is updated. At any point in time a data-record has a ‘current version number’. This model is a source of a lot of problems and complexity in DBMS vis-à-vis master-master replication and otherwise. For example, if an application needed to update a data-record stored in a remote data-store via a computer network connection, it would first need to fetch the record in order to get its current version# and then send the updated value along with the current version number. The initial data-fetch is required not because the end-user wanted the data but merely to satisfy the technical constraints of the data management system.
5. The data-record is a concrete data-structure/type/class[1] of the data-model. Several operations will be performed on this data-structure.
6. When the record is updated, the key stays unchanged but the record’s value is changed in the data-store.
7. If the data is to be replicated (also called data ‘synchronization’) to another data-store, then:

[1] This aspect of the data-model is where *FragmentDB* differs from all pre-existing data storage methods. Within *FragmentDB* a data-record is not a concrete data-structure and is never actually stored; only changes to the abstract data-record are stored. FragmentDB never manipulates data-records. This eliminates a lot of behavioral and structural complexity from the data management system. Since a data-record is never stored, version numbers are not necessary and completely absent from *FragmentDB*. Record key is taken from the record itself, no record identifiers (UUIDs) are internally generated.

    a. A change record is either generated right away or will be inferred at a later point in time. In any case, the change-record is associated with a specific version of the data-record.
    b. The change-record refers to the record's key and its version number.
    c. The change-record may have the updated value in its entirety (a.k.a. record-level change) or parts of the value that changed (a.k.a. field-level changes). It also has additional field 'change-action' to denote the type of record update (e.g. 'add', 'delete', 'update', 'move').
    d. Some implementations may include a time-stamp of when the change occurred, while some others do not do so owing to the possibility that clocks may not be synchronized across devices.
8. Issues during Synchronization (a.k.a. asynchronous replication)
    a. If the record was also updated in separate data-stores, each would produce a change-record referring to the same version number. Later when the two data-stores attempted to synchronize, this would be flagged as a conflict, a situation that would be dealt with using complex conflict resolution logic which do not always yield desired results.
    b. Conflict resolution is especially complex and error-prone when data-models allow field-level changes. During conflict resolution the system has to essentially guess the intentions of the end-user and in some cases it just flags an error and expects a human operator to fix the issue. In some cases it would just coalesce fields from both change-records – but that may not have been the user's intention because the user may have
    c. Another issue is that if data-records of the same key were to be generated on two different data-stores – especially if each had a unique identifier (UUID) – then:
        i. At synchronization time the system would have to go through complex and in some cases heuristic and error-prone logic in order to discover that they were the 'same' record.
        ii. Then appropriate modifications would need to be applied to one of the two sides in order to ensure that both sides used the same identifier (UUID) for the record. In synchronization terminology this is known as de-duplication.
        iii. Some systems do not perform de-duplication and in those cases the systems end-up with duplicate records.
        iv. De-duplication also needs to deal with conflict-resolution in case the two records had differing values.

Novel Data Model

FragmentDB does not store or manipulate data-records. It only stores change records, called action-records or simply, actions.

Data-records only exist as a concept in the end-users mind and on an application's user-interface. The application's user-interface should give the user options to insert, delete and edit the conceptual data-records. Each such operation on a conceptual data-record begets a new action. Creating or updating a new conceptual data-record generates a 'insert' action. Conceptually deleting a data-record generates a new 'delete' action. A conceptual permanent delete of a data-record begets a 'permanent delete' action. As time goes by new actions are generated, but never deleted (except during compaction or synchronization).

A FragmentDB store is just a collection or pool of actions. These are never updated or deleted from the pool (except during compaction and synchronization). There are no version numbers. All conceptual operations on data-records result in new actions being added to the pool therefore there is no need for central locking of a data-record. Actions are simply end-user actions.

When the FragmentDB client loads the FragmentDB into memory, it reads all the actions into memory and groups them by key. Thus all actions pertaining to a key are grouped together and stored inside a data-structure called 'action history'. On being requested to fetch a key (and its value), the most recent action in its action history is retrieved. If the action-type was an 'insert' action, then the corresponding value is the 'current' value of the data-record. If the action-type was a 'delete' action, then the action history is deemed to be in the ‘trash’. If the action-type was a ‘permanent delete’ action, then the data-record is deemed to have been permanently deleted and the action history shall be purged during the next store compaction operation.

Multiple action pools may exist in disparate locations. These can later be pooled together to get a consolidated pool of actions. Asynchronous multi-master server-less data replication is achieved thus. By definition, there is no de-duplication or conflict-resolution because there are no data-records to de-duplicate or resolve (there are only actions).

Server-less Architecture
If data were to be located on a remote network location, then traditional technologies would require a ‘server’ software to be executing on the remote device in order to mediate and orchestrate access from remote locations. The server is usually able to only access local file-systems for reasons of efficiency and direct disk-storage access. A server is also necessary for orchestrating concurrent access to the data-store.

This additional complexity and cost is completely absent from FragmentDB. FragmentDB has no ‘server’ component. It is a ‘client-only’ solution and it supports network-remote as well as cloud file-systems for file-system-like data-stores.

Generic File System as the Mediator
FragmentDB does rely on generic file-system-like APIs to enable concurrent access as well as remote access. By cleverly leveraging file-system APIs for data arrangement, data-access and management, the inventor has created a DMS that does not require a server at all, and yet can provide concurrent access to local as well as network-remote data! File systems and file-servers are readily available on all modern off-the-shelf computing devices and appliances (e.g. NAS devices). Moreover, FragmentDB does not require a file-system, it only requires a file-system-like API, i.e.

1) A hierarchical layout of data-files. This does not have to necessarily be a file-system, it could also be a cloud-storage like WebDav, Amazon S3 etc.
2) Ability to create and rename files
3) Ability to append to files
4) Ability to move and copy files
5) Ability to delete files
6) Ability to either lock files or make atomic updates

Designed for Access from Across a Network
FragmentDB’s data organization is designed to accommodate access from remote devices including from across the internet. All actions are packed together and stored in files of preconfigured size. These sizes can be reconfigured with different values to accommodate different use-cases. For example, some limited devices like mobile phones may not be able to fit very large data-sizes into their network stacks and the max file-size of the FragmentDB could be adjusted accordingly.

Data updates also are also designed to be network efficient. When an action is performed on a data-record, then only that action is transmitted (optionally encrypted) to the file-system across the network. This is a very small amount of data and it is only appended to a file and only to one file. Therefore, the operation is as efficient as possible.

No data is ever inserted into a file. The only way a file can change is by having data appended to it. A file will never be overwritten or have data written anywhere other than at its end. This makes for very efficient updates to the file-system making it very useful if the data-store was located on a remote network device. It also reduces the probability of data corruption.

Support for Synchronized / Cloud File Systems
Several products are available in the market that offer automatic synchronization of files and file folders. A DMS that targets consumer masses should be able to take advantage of these third party products. However, this would require allowing the third-party file-synchronization technology to copy, move, update

and rename database files while completely ignoring all mutex/locking mechanisms used and enforced by the DMS system. This would spell 'disaster' for any such DMS.

Concurrent Access
Traditional DMSs need to carefully orchestrate and mediate any access to the storage. For this reason they require Client-Server architecture to enable concurrent access. They employ locking at various levels or optimistic-concurrency-control which causes updates to be rejected if the client had a stale version of the record it was updating. This is added complexity, difficult architecture and ultimately leads to cap-ex and op-ex costs culminating in diminished user experience.

FragmentDB is a client-only solution and does not resort to locking or versioning except for file-locking provided by the file-system. It implements append-only updates all of which are appended to the end of files regardless of the client that produced them. It leverages the operating systems file-system to perform mediation / orchestration.

Drag and Drop Replication
As mentioned previously, FragmentDB only deals with action-records. It never stores or manipulates data-records directly. Change-records are stored in file-system-like files. Hence synchronization of two data-stores may be achieved merely by collecting all the files together in their respective folders. This has never been done before.

Moreover, this process may be performed even by a third-party – such as a drag-and-drop gesture by a person or synchronization by a third-party cloud-drive such as Dropbox. It clearly does not require orchestration / mediation by a 'server' software. FragmentDB uses the same file system API as the third-parties and therefore uses the file-system to perform mediation and orchestration on its behalf.

Fault Tolerance
All operations performed by FragmentDB that access the file-system may be aborted half-way with the following guarantees:

1) No data loss.
2) No data corruption.
3) Partial results will be achieved, based on how many actions were successfully transferred.

Retrying an operation – aborted or completed - is perfectly alright. Duplicate actions caused by retries will not corrupt the state of the (conceptual) data-records. Duplicate actions will impact efficiency of the system however, until purged.

Other Uses or Applications for This Invention

Other uses or applications.