STRAKT

Infinite Filing Cabinet for CAPS™

Presented at the ACCU Spring Conference, April 2003, UK

Anders Chrigström <ac@strakt.com>
Jacob Hallén <jacob@strakt.com>
Boyd Roberts <boyd@strakt.com>
Ronny Wikh<rw@strakt.com>

AB Strakt
http://www.strakt.com

Abstract

The workhorse of the CAPS™ system is a server module called the Infinite Filing Cabinet (IFC™). Using a regular RDBMS as backend, the IFC provides persistent storage for Python objects. These objects can be stored, retrieved and searched for. In addition, notification of object creation and modification can be requested. More complex actions can also be performed when objects transit specified states at optionally specified times. The IFC does not support the deletion of data; this is performed by deprecation. It is therefore possible to access any object in any of its prior forms.

1. Introduction

The workhorse of the CAPS™[1] system is the database and the supporting software which is called the "Infinite Filing Cabinet", the IFC™. The name stems from the idea that the IFC can be imagined as an infinitely big filing cabinet where any amount of information can be stored. Just like with an ordinary filing cabinet (the useful kind), access to items stored inside is quick and easy.

Information is organised as objects and the IFC is the persistent storage of these objects.

The IFC will contain all the information concerning a particular project, even if that project is done and dead years ago. There is no way of putting information into the system other than through the IFC, even though information which exists outside (e.g. an external customer database etc.) can be referred to from inside the IFC. However, one must remember that any external information is outside the reach of the IFC and thus can change or even disappear with no notice of the fact being given to the IFC, i.e. the information might very well be inaccurate.

Within the IFC all information is always consistent and instantly available[2] .

2. IFC Basics

The IFC acts as a database which informs its clients of changes within the system. The immediate "partner" in this communication scheme is the Business Logic, the BL. There will be several references to the BL throughout the text; however, the IFC is totally self-contained and works as a stand-alone program. Any application that needs persistent Python objects with a sophisticated subscription mechanism can use the services of the IFC.

2.1 IFC Object persistence (deleting objects)

Much in line with modern thinking on the subject of data persistency, an object that is entered to the IFC will remain there forever unless removed by external means. There's absolutely no way of removing an object from the IFC permanently from within CAPS. Objects can be marked as deleted, but they will still exist in the database for later referencing.

Why all this fancy footwork when a simple delete function that anyone could use would appear to be more efficient? The reason is that disks are cheap and mistakes are expensive. It makes a lot more sense to keep as much information in the database, thereby preserving an

[1] Collaborative Approach to Problem Solving – A platform for building multi-user applications for case handling, issue tracking, project management and other workflow oriented endeavours.

[2] As instantly as the system limitations, e.g. network bandwidth allows.

unbroken trace of all information, than to throw away what effectively is hard-won experience which often enough has to be earned again at some later time.

The option where you can erase an object allows for contingencies where large amounts of data has been added which later on turns out to not be necessary in order to preserve the experience of that particular problem. Removing the excess data trims the size of the database without affecting the problem structure as such.

Entirely removing objects from the IFC should be a very rare event which really only is used when truly erroneous or misplaced data has been entered into the IFC.

2.2 The IFC from the database point of view

Given the way a relational database works, and the fact that we as much as possible want to avoid using features of one particular database, there are some considerations that have to be taken into account when creating an object model that in the end lives inside such a relational database. This chapter deals with our way of solving certain problems of IFC object implementation in the relational database.

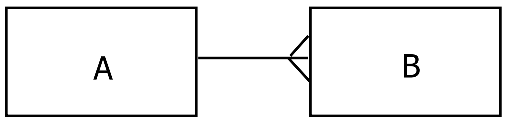
2.2.1 ER graph use with relational databases

In a relational database, data is stored in tables. Each row in a table constitutes one *record*, one data collection for that specific table. Each record can hold a number of *fields* of data, arranged as columns in the row. However, one field can only hold one single data item as such, e.g. one integer or one string etc.

This means that when one wants to represent lists of several instances of a data type, one has to make some kind of database construction based on several records, possibly in more than one table.

A typical such construct becomes necessary when one wants to link one record in a table to several others, a so called *one to many* relation. In the ER (entity relation) graphs used later in this document, such a relation is represented like this, where a record in table A refers to several instances of records in table B, but a record in table B only can refer to one instance of a record A:

Example 1

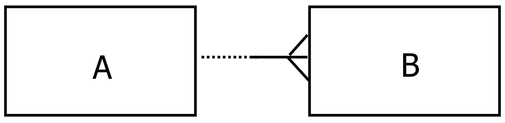


This is accomplished by having a field in B referring to an A record, thus more than one record instance of B can refer to a single record A. Please notice that there is no information stored in A about this relation whatsoever. This means that in order to obtain the list of B records that refer to a specific record in A, one has to perform a database query accessing all table B records and examining their contents; it's not a simple matter of obtaining a field from one single record.

The fully drawn line *starting from* a specific table in an ER graph indicates that the relation *must* exist. In the graph above every record in table A *has to* have at least one record B referring to it, and every record B *has to* refer to one record A.

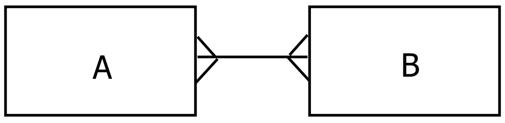
In the graph below, the dashed line starting in table A indicates that a specific record A may or may not have a record B referring to it, but the fully drawn line from the B table still means that every record B *has to* refer to one record A.

Example 2



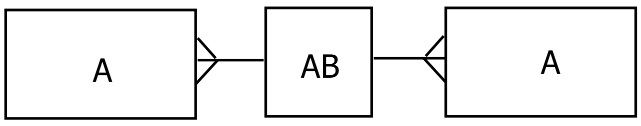
It becomes more complicated when one has a need for *many to many* relations; when records in both tables A and B above should be able to refer to many records of the other table type.

Example 3



This is accomplished by the use of a helper table put in between, playing the part of the *one to many* relation table in both directions. Obviously, this means that searching for records in B referred to by records in A will be that much more costly, since yet another table is involved, but this really is the only way of solving the problem. Again obviously, it's wise to design object models with as few *many to many* relations as possible simply because queries become so costly.

Example 4



When drawing ER graphs, the above construct is usually taken for granted and is never explicitly drawn. If a table is present in an ER graph, drawn like the one above, it is because it fills a purpose other than to work merely as a *many to many* relation workaround.

In the software making use of the database it is more convenient to add a middle layer that hides these considerations as much as possible. For that reason, *objects* representing records of table A in the first graph above should contain an attribute that is a list of all records of type B that is referring to it, having been invisibly queried separately from the database when the record A was obtained. It also means that when storing an object of type A, one has to either remember the original list of B records that was referring to it and compare that to the currently valid one in the object, or perform a new query in the database, and then perform operations on all records of type B that has changed according to the list in the object A.

2.2.2 Common IFC table contents

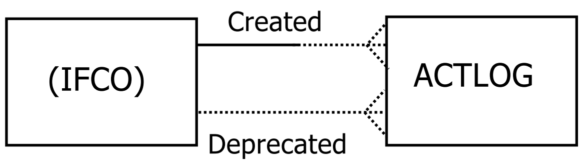
The information stored in the IFC is comprised of a about a dozen object types. Not all of these are intended to be available directly to the outside world; some act in conjunction to support certain types of data available through others.

Most of these tables share a number of common traits, or more specifically common fields.

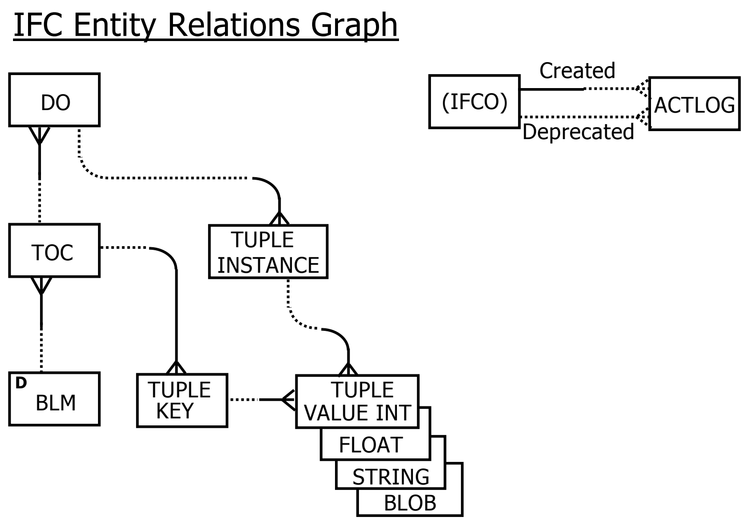
All tables that are intended to be referenced from "outside" the IFC have this in common:

- An identification field named "id", uniquely identifying the record (throughout the database, for any and all tables) with a 64 bit id number. Note that all changed versions of the same record also have the same id number in the same table, but that the "deprecated" field is used to distinguish the currently valid record, as discussed later in this chapter.
- Most have a "name" field; a string which is unique within the table for valid records.
- Two fields named "created" and "deprecated", which are used to handle the "infinity" part of the IFC concept. The "created" record points to a log table record where the details regarding the creation of the record are stored. The "deprecated" record is set to NULL for the active (currently valid) record. When a record is removed, a new log entry is created detailing the circumstances (who, why and when) of the removal and a reference to it is stored in the "deprecated" field. When a record is changed, the same procedure is followed except that all the contents of the record is copied to a new row with the deprecated field set to NULL and the "created" record set to point to the "deprecated" log entry of the previous record. Thus, to find the currently valid record in a table, look for the id (and/or name) of the field in the row where the "deprecated" field is set to NULL. In order to rollback the database, just follow the trail of "created"/"deprecated" entries back in time.

<u>Action log ER graph</u>



3. The IFC data structure



This graph describes (almost) all tables and relations used by the system. It will be broken down into each functional piece later on in the text and described in detail.
Each box in the above diagram represents an IFC object, or IFCO.

The diagram omits one very important detail, which is why a small digression is necessary, in order to make the following ER graphs understandable.
All IFCOs in the diagram, with the exception of the one labelled "DO", only contain information regarding relations to other IFCOs along with some special information determined by their use in the system. User data - what the average customer would like to store in the system - can only be contained in the aforementioned DO (Data Object) structure.

Data stored in an IFCO table directly is called an *intrinsic* value. *Intrinsics* can only exist as single values - one value per IFCO.

Data stored in a DO structure is called a *tuple* value. *Tuples* are always lists, they can be from zero to thousands of values.

3.1 Attributes

Data in the IFC is available as intrinsics and tuples, but only to the IFC and the IFC-oriented part of the BL. For the client and the client-oriented part a transformation of these intrinsics and tuples are made into something called *attributes.*

In this text there are references to intrinsics, tuples and attributes depending on the context. If the data is supposed to be used by the IFC or the IFC-oriented part of the BL, it's referred to as intrinsics or tuples. If it's being used by the client or the client-oriented part of the BL, it's referred to as attributes.

3.2 Business Logic Module, BLM

The specifics of a business, the hierarchy of a company, the intrinsics of how daily tasks are handled are stored as Business Logic Modules, BLMs, in the IFC. The BLMs convert the DO structures into objects that the user in front of the screen can access through his client, and from the client back into DO structures that can be stored in the IFC. This, in fact, is the one of the main purposes of the BL.

A specific BLM organises a number of Task Object Classes, TOCs, which represents the individual data types comprising a BLM. The base BLM, for instance, defines a specific TOC for all of the DO structures used to contain user and access control information. Users access objects as instances based on TOCs, so called Task Object Instances, TOIs.

TOCs may inherit other TOCs. What is inherited then is all attributes from the inherited TOC which cannot be changed.

All and every DO structure that is available to a user as a TOI has to have a BLM: TOC specification in order for the BL to know how to handle them. A user type IFCO is specified as base: user.

The BLM object defines these intrinsics:

- id
  A unique 64 bit identifier. Please notice that as with all IDs used in any table, it's unique only in conjunction with one or more entries, always the deprecated value but sometimes there also exists a unique index in order to establish some kind of context rule.
- name
  A name that is unique among non-deprecated BLM objects. The name may exist in duplicates among the deprecated objects.
- desc
  A description of the BLM, its purpose and usage.
- code
  The BLM python code, stored as a BLOB.
- version
  The current version of the python code, for matching purposes at start-up of the BL.
- created/deprecated
  Entries implementing the "infinite" property of the IFC, as described earlier.

3.3 Task Object Class, TOC

The purpose of the Task Object Class, the TOC, is to represent the individual data types that make up an entire BLM. The TOC object in the IFC only contains a reference to its parent BLM, all actual functionality exists in the python code which resides in the BLM and is executed in the BL.

The BLM object defines these intrinsics:

- id
  A unique (id + deprecated) 64 bit identifier.
- name
  A name that is unique among non-deprecated TOC objects. The name may exist in duplicates among the deprecated objects.
- blm_id
  The parent BLM.
- basetoc
  A reference to the inherited TOC.
- created/deprecated
  Entries implementing the "infinite" property of the IFC, as described earlier.

3.4 Data Object, DO

The Data Object, the DO, is the workhorse of the IFC data object set. The structure based on this object is used to store most types of user-defined data and thereby comprises the core of almost all types of TOCs. Exactly how the DO structure stores attributes will be discussed in a later chapter, it's enough here to know that it does in some way.

The DO object defines these intrinsics:

- id
  A unique (id + deprecated) 64 bit identifier.
- toc_id
  A reference to the TOC specifying the contents of the DO structure.
- created/deprecated
  Entries implementing the "infinite" property of the IFC, as described earlier.

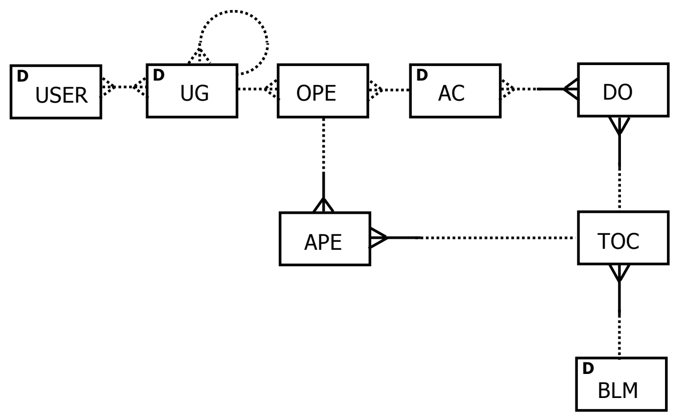
The DO structure may contain attributes of the following types:

- ATypeBool: Integer
- ATypeInt : Integer
- ATypeFloat: Float
- ATypeStr: String
- ATypeBlob: Blob
- ATypeDate: Integer
- ATypeTime: Integer
- ATypeTimestamp: Float
- AtypeTOIRef: Integer

3.5 Object access implementation

Access control is a part of the BL; the only role of the IFC is to supply the objects that control access to other objects. The following description illustrates how the different object types interact. The example shows the Base BLM, which contains the definition of users and all the objects that control user access to all objects in the system (including all the Base BLM objects themselves).

Access Control Relations Graph



The above relations graph describes the access control system.
Objects in the graph that are implemented as DO structures have a small capital 'D' in their top left corner.

A user of the system needs to have a corresponding USER entry containing various parameters like name, authorisation data etc. As can be seen, users can be grouped into user groups (UGs).

UGs are used to implement a company hierarchy. They contain a name and a description. UGs can be grouped into other UGs.

DOs can specify an Access Classification, an AC, to determine which type of object it belongs to.

The AC can be seen very much as a UG but used to organise DOs instead. For the moment we don't allow hierarchies of ACs.

The path from a UG to an AC determines the access for a specific USER. The path is formed by tying together a UG and an AC with an Object Permission Entry, an OPE. There can only be one OPE between a specific UG and AC.

The OPE contains references to one or more Access Permission Entries, APEs.

An APE contains information on the format: blm: toc: attribute: permission. This makes it possible to specify exactly which permission to grant to a specific attribute in a certain TOC. As the OPE can contain references to several APEs, its possible to define a set of standard permissions which then can be combined to form specific solutions for different types of access within a company hierarchy.

3.6 Log objects

A DO structure is just a framework for containing data. The BL makes use of the structuring mechanism in order to build problem relations.

3.6.1 Action Log Object

Any status change of any IFCO causes a log entry to be created. The LOs are tied to an IFCO so that the entire chain of events concerning that IFCO can be reconstructed.
For practical purposes, the LOs are stored in the IFC, but users can not manipulate LOs directly.

LOs consist of the following entries:

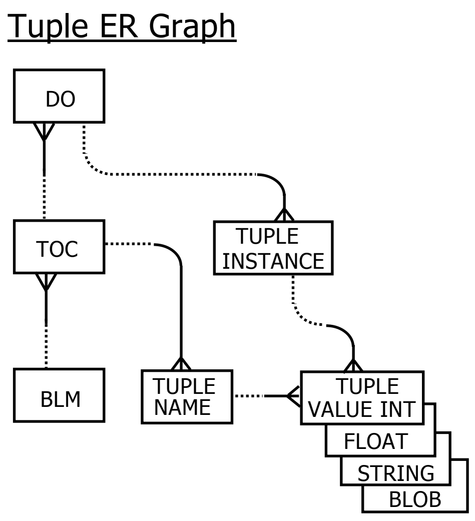
- A reference to the relevant IFCO.
- The event that occurred, such as creation, completion, deletion etc.
- The user session that caused the event.
- A timestamp.

The log is also used to acquire information about creation time and creator of IFCOs, information which is not stored directly in the entities themselves.

Access to an individual LO is the same as the access state of the referenced IFCO.

3.7 Substructures in the IFC

3.7.1 Tuple list



A tuple is a data structure which associates one value, the key, with another value, the data. For example the tuple { "finger" : 7 } associates the value 7 with the key "finger". In our implementation, a key is always a string while the value can be of almost any type.

A tuple list is tied to each DO, as seen above. A tuple list is an extendable list of properties associated with an entity. It's the BL which defines which properties are handled.

The information in the tuple list is typically the subject for searches in the IFC.
The tuple value can be a number of different things:

- string
- integer
- float
- boolean
- time/date value
- reference to another object

While any number of { key : value } pairs can be stored in a tuple list, some names are to be considered as standardized and should not be used for any purpose outside the definition. There are also a few reserved tuple key names which the BL won't accept.

4. Connection management

Individual users can't connect to the IFC directly; they have to work through a Business Logic layer. However, information about connected BLs and user clients is stored in the IFC for the purpose of ensuring security and efficiency.

4.1 Business logic connections and security

When a BL attempts to connect, a private-key encrypted cookie is sent to the IFC, which tries to decrypt it using a previously stored public-key component. If the cookie can be decrypted successfully, the BL is allowed to establish a connection.
Several BLs may connect to the IFC at one time.

4.2 User client connections and security

When a user client attempts to connect to a BL, a login-query is sent to the IFC using private/public key authorization methods as with the BL. If the client successfully identifies itself, the client-BL connection is stored in an internal list as long as the BL-IFC connection remains valid, or until the BL requests the removal of the entry.
The BL requests the removal of a client entry when a client disconnects from the BL.
When the client as part of the normal work process requests information from the IFC, the connection is considered to be secure and no further user connection validation is necessary.

5. Object interface

The IFC provides three basic operations to find, retrieve, change and create objects. The first operation is known as a query which specifies sets of tests to be evaluated against various TOCs and returns a list of object identifiers which satisfy the tests. These identifiers are used in turn refer to the objects themselves.

Using an object's identifier a *request* for or a change to an object's data can be made.
Creation of an object is viewed as modification of the non existent object to a supplied set of values. Change and creation are implemented by the *commit* operation.

Queries and requests are further divided into two classes: *transients* and *subscriptions*. The former is treated as a one-shot enquiry while the later indicates that future changes should be notified as they occur. These two classes are not applied to commits, although it is the commit that triggers notification of change.

5.1 Queries

Object queries are constructed from a query object. This consists of a list of query condition groups which specify expressions to be evaluated against a TOC. Each expression in the query condition group is and-ed with the next as it is evaluated from left to right. Such an expression is termed a query condition which consists of an attribute or tuple name, operator and value allowing constructs of the form:
QC_1 and QC_2 and QC_3 and ... and QC_n[3]
Query condition operators are binary operators of the set:

- equality
- inequality
- greater than
- less than
- greater than or equality
- less than or equality
- set membership
- not a set member

Simple pattern matching is also supported, allowing combinations of:

- a literal character
- any character
- zero or more characters

A query condition group may contain zero query conditions and in this case it is an implicit test which is true for all the objects in the TOC.

A query object may contain multiple query condition groups and these are or-ed together allowing constructs of the form:
CG_1 or CG_2 or CG_3or ... or CG_n[4]

[3]Where QC is a query condition.

Using both constructs allow queries of the form:

action[5] == 'Run diagnostics on printer'

or

action == 'Repair printer' **and** state[6] == 'Finished'

The above query object consists of two query conditions groups which are implicitly or-ed together. The first has one query condition while the second has two. The query conditions in the second are implicitly and-ed together. The query is in *disjunctive normal form*. This simplifies parsing of the query as no operator presence problems arise; there is only ever one operator in any given context.

When a query is made the query object is interpreted into a database query[7] and the query is run. The result is a list of object identifiers and they are returned to the requesting BL. This list of object identifiers is known as a *result set*.
This is the simple case of a *transient* query where neither the query nor the result set are recorded.

Should the query be a *subscription* both the query object and the result set are recorded. Future changes to objects in the IFC may cause objects to be added to or deleted from result sets. Such a change is termed a *result set change*. When such changes occur they are forwarded to the appropriate BLs. This provides dynamic notification of changes to results sets.

5.2 Requests

Objects are requested from the IFC by supplying their indentifiers. Once the object is found it is returned to the requesting BL. This is the simple case of a *transient* request.
Should the request be a *subscription* the IFC notes that the requesting BL wishes to be informed when the object is modified. When such a modification occurs the modified object is sent to all the BLs that have subscribed to this object. This provides dynamic notification of changes to objects.

5.3 Commits

A commit is a creation of a new object and/or a modification to an existing object. It consists of a set of objects to be created or modified. All objects in the set are viewed to be part of a transaction. The transaction ensures that either all objects are processed successfully or none are. Should any operation fail no modification to the IFC's state is made.

There is no explicit transaction setup, completion or revocation. It is assumed that all the objects taking part in a single commit form a transaction. The IFC enforces no other structure; this is a problem to be resolved by the commiter.

---

[4]Where CG is a query condition group.

[5]A TOC attribute or tuple name.

[6]Another TOC attribute or intrinsic name.

[7]The query object is analysed and turned into the relevant SQL.

Objects to be committed are referred to by their identifier. Attributes to be created or modified are supplied and this data is applied to the existing data in the object. Should new objects require creation a set of unique temporary identifiers, one per object, assigned by the BL, are used. These are then mapped to new object identifiers and this mapping is returned at the conclusion of a successful commit.

As the commit is being processed incremental state information is saved so that commit related operations can be carried out should the commit succeed. Should it fail the information is discarded.

Once a commit has been successfully processed change notification and events are processed.

6. Object cache

To improve the efficiency of object access, objects are cached by the IFC. As objects are requested or created they are stored in a simple cache. The object's identifier is used to refer to it in the cache. To minimise the amount of wastage in the cache only object attributes that were requested, created or modified are cached. Should the object exist in the cache but a particular requested attribute be missing it is added.

6.1 Basic design

The size of the cache is fixed and when the *high water mark* is exceeded it is analysed and garbage collected until its size is less than the *low water mark*. This approach is used to ensure that garbage collection yields some useful result for the amount of effort expended. Garbage collection is assumed to be expensive so the *high water* value is 95%[8] while the *low water* value is 80% of the cache size. These values are essentially arbitrary and can be changed, if necessary.

As an object traverses the cache three things are recorded:

1. Current size of the cached object (not the total size).
2. Real time cost spent during database operations for the object.
3. The outstanding requestors with subscriptions for the object.

This information is used to manage garbage collection of the cache. The current size is used to calculate how full the cache is. Once garbage collection begins the cache is analysed and the object's cost is used to determine should it stay in the cache or not. The cost of the object is decayed as a function of the *mean object cost*, a value calculated to be the average cost of requesting objects from the database in the recent past. Periodically this value is recalculated so that it reflects a value that is relevant to the objects in the cache. Again this value is arbitrary (currently 128 object *reads* from the database) but can be changed.

[8]This value was chosen based on a 5% overhead for maintaining the cached objects.

How worthy an object is of remaining in the cache is further mitigated by the number of subscriptions it has. A delta value is calculated from the mean object cost divided by 2 and that value is divided by the number of subscriptions. More subscriptions indicate more interest in the object so it is deemed a better choice to retain. The delta is then subtracted from the object's cost. Should the object cost drop to zero or become negative, the object is removed from the cache.

Removal from the cache is complicated by the outstanding subscriptions for the object. They are recorded with the cached object and must be maintained. In this case the object's data is removed but its remaining cache entry (including subscription information) is retained. Its cost is set to *unknown*[9] and all that is left is a placemarker. When the last subscription is removed this placemarker is removed.

6.2 Interface

An object based interface is provided to the cache. The following operations are provided:

- Find an object in the cache from its identifier. This may return that the object is not in the cache.
- Get an object with optionally specified attributes. If no attributes are specified then all are returned. Either the object is found in the cache or it is fetched from the database and cached.
- Fetch an object with optionally specified attributes. This is similar to the previous operation except it is tied to a requestor. If the request is a subscription the cached object is tagged with the requestor.
- Update an object from a supplied set of values, avoiding unnecessary database access. If the object is not present in the cache it is added. It is used during commits to synchronise the cached object with the data in the database. It is performed *after* a successful commit and is part of the chain of dynamic change notification.
- Finish with an object and return it to the cache. A parameter can specify if the object should be forcibly purged from the cache.

Operations that return an object must be paired with a call to its finish method. An object cannot be garbage collected while a call to its finish method is pending.

6.3 Overall management

Calling an object's finish method also maintains a *least recently used* [LRU] ordering of the cache and triggers garbage collection. When garbage collection is required a *one handed clock* scan of all the objects in the LRU is performed. Older objects are found sooner in the scan because the finish method places the object at the tail of the LRU.[10] The scan continues until garbage collection is no longer necessary. The design of the cache ensures that even though several scans of the cache may take place the scan will terminate.

[9]Chosen to be a value smaller than the absolute minimum cost to retrieve an object. An object's cost is ensured to never be less than this minimum.

[10]The LRU list is actually a doubly linked cyclic data structure. The tail can be considered to be just before the clock hand, while the head is the current position.

A feature of the finish method is that it notices if the object had already been placed in the LRU. If this is true the position is not modified. This is to allow arbitrary placement based on some other heuristic; objects could be penalised by being placed just in front of the clock hand so they would be analysed, possibly being discarded, sooner.

A design goal of the cache was to use simple algorithms that were based on sound theory. The combination of decayed cost and LRU based management is to enable costly, frequently used objects to remain in the cache longer than cheaper, less likely to be used objects.

6.4 Resizing

The cache is also designed so that it can be dynamically resized. It can be grown or shrunk at run time, although there is currently no way to instigate this. Should it be shrunk it is garbage collected in the normal way, if necessary.

7. Dynamic change notification

There are two types of changes that the IFC notifies dynamically. The first is change to an object while the second is a change to a result set. Notification of changes to an object is trivial in that all that is required is a list of BLs that require notification. Once a change to an object is made each BL in this list is notified. Detection of a result set change is considerably more difficult.

It should be remembered that change notification is only provided to queries and requests that were subscriptions and this section pertains only to them.

7.1 Result set changes

A result set is created when a query has been run in the database. Once it has been run the query and the result set are recorded. A naive approach to detecting result set changes would be to re-run all the queries when an object is modified. Clearly, this is unacceptable because of the unreasonable cost involved. The goal is to find a minimal subset of queries that the modified object may satisfy.
However, it is somewhat more complex that that. The result of modifying an object can be:

1. Addition of the object to one or more result sets
2. Deletion of the object from one or more result sets
3. No change to any result set

Cases 1 and 2 can even apply to the same result set which entails filtering of redundant additions and deletions as well as resolution of the resulting conflicts. A conflict occurs when an object is subject to an addition and a deletion. This occurs when one query condition group results in addition while another results in deletion. In this case additions take priority over deletions.

7.2 Finding a minimal subset of queries

As query conditions groups are or-ed together it is possible to view all of them as one query object. What needs to be maintained is an association with the corresponding result set. Once a group is evaluated with the data in the object then the object falls into one of the above three cases. A naive approach could be used and all groups could be evaluated against the modified object and the resulting additions and deletions could be filtered. However, it would be prohibitively expensive.

If the object is examined it can be seen that it consists of:

1. a TOC
2. attribute names
3. values of the attributes

This limits the possible set of groups that it can be matched by because it is constrained by its TOC and the TOC attribute names.

If a query condition group is examined it can be seen that it consists of:

1. a TOC
2. attribute names
3. comparison operators
4. values to be compared

From this information it can be seen that both share the TOC as well as the attribute names. Therefore, a subset of all the groups can be constructed from the groups that contain the TOC as well as the attribute names. This subset is the set of groups that could affect the membership of the object to a result set. All other groups do not contain the requisite information to affect result set membership.

When a query is made each group is analysed to construct a list of (TOC, name) pairs where names are selected from the attribute names. Each pair is then used to reference a list of groups that contain the TOC and the name:

$TOC_t, name_n : CG_1 \dots CG_n$

Each group in turn are linked to their corresponding result set.

In the case where no query conditions are present an entry is constructed from the TOC and the *null* name.

7.3 Evaluating query condition groups with a modified object

When an object is modified it is tested with the set of possible query condition groups that could include it or exclude it from a result set. To construct this the object's TOC as well as the attribute names are used to generate a list of *(TOC*, name) pairs. These are used to find corresponding lists of query condition groups.

In addition to this the *(TOC*, *null*) pair is also tried. This additional case only yields results if there exists a query condition group with no query conditions.

From the resulting set of query condition groups each is evaluated in turn. The evaluation could be performed by the database but this would be inefficient because the query would

examine far more data than necessary. This case is special in that one object is being tested, instead of a set of queries being run against the whole database.
The outcome of the evaluation is an addition list and a deletion list, each containing references to result sets. Should a query condition group evaluate to true a reference to its result set is added to the addition list, otherwise the reference is added to the deletion list. Both are then examined to create a list of result sets that have *changed* and these changes are then returned to the appropriate BLs.

The addition list is examined first and result sets referenced in this list are added to the changed list iff:

1. The object is not in the result set.
2. The result set is not in the changed list.

Condition 1 ensures that the object is not added to a result set that it already exists in.
The deletion list is examined next and result sets referenced in the list are added to the changed list iff:

1. The result set is not in the addition list.
2. The result set is not in the changed list.

Condition 1 ensures that addition takes precedence over deletion because the requirement for addition is that at least one query condition group is true despite a potential plethora of groups evaluating to false.
In both cases, condition 2 is used to ensure references to a changed result set occur in the change list only once.
The changed list is then examined and the result sets referenced by it are dispatched to the appropriate BLs.

7.4 Evaluation Optimisations

There are two optimisations that are used to minimise the set of redundant additions and deletions. This is done as the query condition groups are evaluated.

The first optimisation relies on the premise that a reference to a result set in the addition list renders further tests unnecessary. This operation is idempotent in that more than one addition is the same as exactly one addition. As addition takes preference over deletion, presence of the result set in the deletion list is not an issue so further evaluation can be avoided. This optimisation is designed to minimise the number of query condition group evaluations.

The second optimisation is similar in that it relies on the premise that deletion is also idempotent. Once a result set is referenced by the deletions list further references serve no purpose. This is slightly different than the addition case because this occurs after an evaluation; it only serves to minimise the size of the deletion list.

8. Events mechanism

Events are actions that are taken when a particular condition is true. A condition comprises of a set of expressions which describe an object's state, state transition or state at a given time.

This is further generalised to the state of a set of objects, allowing arbitrarily complex expressions to be constructed. A state is considered to be the value of an attribute. It is not restricted to any one attribute.

An event object is the only object that is treated specially by the IFC. In general, the IFC has no idea what the purpose of a given object is, nor does it have any idea about what purpose the various attributes serve. All it needs to know is how to manage the mapping from an object's TOC to the database implementation. This special handling of event objects can only be done by the IFC as it is the interface to the database and the arbiter who can manages multiple BLs who have no idea of each other's existence.

As event objects are essentially objects like any other they can be created, searched for, requested and modified using the existing interface; no special interface or extra overhead is involved. What makes an event object special is how the IFC treats it, which is invisible to the BL.

An event consists of the following components:

- Issuing BL
- Corresponding BLM
- User who created it
- Transition expressions
- Object identifiers of objects referenced in the transitions expressions[11]
- Action to take

As objects are modified or transition expression timers expire relevant events' transition expressions are evaluated. Should an expression evaluate to true the event is dispatched to the issuing BL. If it is not currently active the event is tagged to be true and will be dispatched to the BL when it connects to the IFC. The BL organises for the event's action to be executed in the context of the user who created it.

This describes the basic event mechanism with is used to implement *rule based* events. The basic mechanism is triggered by object modification or time while rule based events are triggered by object creation. A rule based event is triggered when a new object is created and its transition expressions evaluate to true, usually in reference to the values in the new object.

8.1 Transition expressions

Transitions expressions consist of a list of and-ed conditions that must be satisfied for the event to be dispatched.
Components of a transition expression are:

- Object identifier
- Attribute name
- Operator
- Value(s)

[11]This are calculated by the *IFC*.

A time can be specified so that an expression can be evaluated at a given time. This time is intrinsically linked with its operator.
The operators consist of:

- IN
  The attribute is equal to the value.
- ENTERS
  The attribute has been modified to be the value.
- LEAVES
  The attribute had the value before it was modified.
- CHANGES
  The attribute changes from one value to another. In this case *current* and *next* values are specified.
- AT
  The attribute has the value at a specified time.

The AT operator has an associated time for the test to be done. Other than this it equivalent to the IN operator.

8.2 Management

Efficient management of events is necessary so that acceptable performance is achieved. In the general case, every commit involves examining all the potentially triggerable events and evaluating them. Of course this would become prohibitively expensive as the number of events grow, so a mechanism must be used to quickly target events that could be triggered.

A similar mechanism used with result set changes is re-employed to quickly find all the events a created or modified object refer to. For each case an *index*[12] is used to quickly find the relevant events. The difference between the two cases is the type of key used with the index.

On creation of an object all the rule based events referring to the objects TOC are picked; the key is the TOC:
TOC_t : $Event_1$ ... $Event_n$

Modification is slightly more complex because both the object identifier and its attributes must be matched against similar values contained in event transition expressions; the keys are made from the object identifier and its modified attributes:
$Object_{id}$, *Attribute* : $Event_1$ ... $Event_n$

Both these structures must be updated as event objects are modified.

8.3 Timers

The management of timed events is handled by a different mechanism. Obviously they cannot be handled as commits are processed as they are completely unrelated. A list of timed events is maintained which is sorted by time; earliest at the front, latest at the end.

[12]See 9.

When a timed transition expression is discovered it is added to the list. If necessary, a call to a function to evaluate the event's transition expressions at the allotted time is organised. If they evaluate to true the event is dispatched to the BL. If they evaluate to false the event discarded as it can never evaluate to true as it has failed at an allocated point in time.

This structure must also be updated as event objects are added, modified and as timed events are evaluated.

8.4 Rule based events

Rule based events are very similar to basic events but with a few exceptions:

- An attribute which indicates the TOC the event refers to.
- Transition expressions which implicitly refer to the created object.
- The only supported operator is IN.

When a new object is created its TOC is used to see if there are rule based events for that TOC. If so, each in turn is evaluated. Should it evaluate to be true then the event is *cloned*. Because a rule based event is so similar to a basic event it can be copied an then treated by the normal event processing code.
Cloning involves:

- Copying the rule based event
- Removal of the attribute which indicates the TOC the event refers to.
- Allocation of a new object identifier

A cloned event is an object that can be committed to the database. It can be processed normally after being committed to the database. Should one of the committing objects fail to commit all previously committed object are rolled back. This applies to cloned events that have been committed because they fall into the same transaction as the objects for committal. Hence transaction semantics are maintained.

9. Indexes

The concept of an *index* was developed to enable rapid retrieval of a minimal subset. When faced with the problem of finding query condition groups that may refer to a modified object the insight was made that if a *key* could be constructed from a query condition and then recreated when an object was modified it would be possible to find all the relevant query condition groups quickly. The same problem arose when a similar solution to event management was required.

Once the problem of key construction was solved the implementation of an index was straightforward. The IFC is written in Python so it was obvious that a *dictionary*,[13] indexed by the key, referring to a list of the related data items would be ideal. Dictionaries allow an arbitrary key to quickly locate the related data. As there were more than one use for such a data structure it was implemented in a generic way as a *class*.

[13]An associative array which can be indexed by arbitrary data.

An index's interface is:

- add(key, data)[14]
  Uniquely add *data* to the list of items located by *key*.
- has_key(key)
  Returns *true* if any data can be located by *key*.
- get(key, default)
  Return the list of items located by *key*. If none exist then return *default*.
- remove(key, data)
  Remove *data* from the list of items located by *key*.

Both *key* and *data* are arbitrary values. Should multiple values be required to construct a key the method used is to construct the key from a tuple containing all the values. Data contained in the lists is normally of the same type, but there is no restriction that it need be.

These methods closely follow normal Python dictionary methods.

10. Communication objects

The object interface is managed by entities known as *communication objects*. Each contains an underlying data object which is manipulated by a set of associated signaling methods. The number of methods is small and is built on an *activate*, *update* and *deactivate* model. These objects should not be confused with objects stored by the IFC.

- Activate signals the creation of a new communications object.
- Update signals that the data associated with the communications object should be analysed and updated.
- Deactivate signals that an existing communication object will be destroyed.

A typical communication object transaction consists of an activate, zero or more updates followed by a deactivate. The update method is used to both access and create objects in the IFC, instead of having explicit access and create methods. This approach is similar to the idea that creation can be implemented by modification.

10.1 Marshaling

Marshaling is the layer between the Communication Objects and query and object management. It analyses the requested operation and dispatches it in the correct manner. Only queries and requests are managed by this layer while commits take a more direct path.

[14]This could have been implemented by Python's __*setitem*__ method but as it is not strict assignment an explicit method was chosen to avoid confusion.

10.1.1 Query Marshaling

On query activation the query is run against the database and its result set is returned to the BL. Should the query be a subscription it and its result set are recorded and persist until it is deactivated.

On each update operation the updated query is run, results are returned and it may also be recorded. This allows for a query to be modified.

On deactivation the data pertaining to the query is destroyed.

10.1.2 Request Marshaling

On request activation the requested object is found and returned to the BL. If the request is a subscription the requesting communications object is recorded by the object manager.

11. Platform

The IFC has been written in Python, using the Twisted framework and pyOpenSSL/OpenSSL for communication with the BL. Psycopg is used as the interface to the relational database. Our current implementation uses PostgreSQL as the relational database engine. However, since we use a fairly standard set of SQL, with very few implementation specific constructs, a port to another database engine should be a fairly small matter.

12. Performance

As the amount of data at the installation of our pilot customer grew to over 20,000 objects, we discovered that the IFC was surprisingly slow, sometimes taking seconds to assemble a single object. The slowness was most notable right after a reboot, indicating that the bottleneck was in the RDBMS rather than in the code of the IFC itself. Otherwise cached objects would show the same slowness as the ones retrieved from the database. Analysis has recently shown that unless you explicitly cast incoming 32 bit integers (id's in particular) to 64 bit integers, PostgreSQL will perform a linear search through the table containing 64 bit integer keys, instead of using the intended index.

For this reason, we only have very limited data on performance. However, our pilot customer runs the system with 50 concurrent users. After fixing the cast problem, throughput has become quite reasonable.

13. Future directions

Currently we have a few ideas for future work with the IFC. Some of them are described in this section.

13.1 Timesharing

Currently the IFC will handle query results, requested objects and commits in a strict first-come, first-served order. This can lead to very long delays for small requests, if large requests are ahead in the queue. By serving request in a round-robin between different clients, we should be able to reduce latency for small service requests.

13.2 Optimisation

Optimisation is an obvious field for future work. Currently a lot of time is spent in the database because of the peculiar use that is made of a standard relation database. Instead of having many tables, say one for each TOC, there are a small number (typically 4 to 6). As a result, these tables are heavily used and this is compounded by the fact that no data is ever deleted from them.

A more mundane optimisation is to split the string table into one that contains tuples that the RDBMS is able to index and one with longer strings that remains unindexed. Currently, all strings are in an unindexed table and have to be searched for linearly.

13.3 Concurrency

The IFC of today is strictly single threaded and is only able to serve a single instance of the BL. In a first step, we will allow more than one BL to be connected to the IFC at any point in time. A later step will allow the IFC to service several requests in parallell. For instance, it should be possible to serve objects out of the cache while waiting for the RDBMS to supply data for non-cached objects.

We have also considered allowing several instances of the IFC to operate on the database at once. However, this puts rather high demands on synchronization between the instances of the IFC. We will avoid this step as long as we can maintain acceptable performance with a single instance.

13.4 Redundancy

The CAPS™ system is expected to be mission critical in many environments where it is installed. By using a distributed RDBMS and a fail-over mechanism for the IFC and BL, we should be able to provide a system with no single point of failure.

13.5 Query caching

It would be possible to maintain a cache of previously run queries and examine them before going to the database. Currently this is not done but a prototype has been tested and it has resulted in better performance. The cache comprised of only the currently active result sets, but this could be extended.

13.6 Point in time access

Given that the IFC maintains every copy of every object it would be very useful to be able to access them at an arbitrary point in time. This would allow a view of the world at some time in the past and, as an added benefit, the ability to backup without interrupting normal operation. A *back of the envelope* design has been done and it would appear that with a slight addition to the information currently stored in the database such an interface could be provided.

Queries would have an additional parameter to specify when (in time) the query should be interpreted. Object identifiers would then consist of their traditional identifier plus a version identifier. Query results would return these identifiers and requests would then interpret these to return the object at that point in time.