S3-82

N93-22152

Universal Index System

Steve Kelley
Nick Roussopoulos
Timos Sellis
Sarah Wallace

Advanced Communication Technology Inc.
1209 Goth Lane
Silver Spring, Maryland 20905

ABSTRACT

The Universal Index System (UIS) is a index management system that uses a uniform interface to solve the heterogeneity problem among database management systems. UIS provides an easy-to-use common interface to access all underlying data, but also allows different underlying database management systems, storage representations, and access methods.

1. Introduction

Today, there is a great diversity of computers, operating systems, database management systems, and communication protocols. As a result of this heterogeneity, computer users are required to learn many different data access methods in order to obtain the information they need. This causes an attitude of "it's too much trouble to learn all these different systems," which leads to a significant amount of software and data duplication.

There are several approaches that can be taken to solve the heterogeneity problem: two of which are *standardization* and *uniformization*. *Standardization* is the concept of choosing one specific system to use, and expecting or requiring everyone to follow this standard. This, however, does not provide an adequate solution because it could be extremely costly to change to the standard if a different system was being used. *Uniformization* is the concept of creating a layer on top of current systems that provides uniform access to all data, regardless of the underlying system. This allows the underlying systems to remain unchanged, yet also provides a single common access method for users to access data.

This paper presents the Universal Index System (UIS), an index management system that uses a uniform interface to solve the heterogeneity problem among database management systems. UIS provides an easy-to-use common interface to access all underlying data, but also allows different underlying database management systems, storage representations, and access methods.

2. UIS Components

UIS is a system that manages and maintains indexes, sets, indexsets, and indexkits. An *index* is an object that associates terms with pointers. A simple example of an index is the index of a book. It associates a term used in the book with the page number(s) on which that term appears. Another example of an index is a subject index in a library catalog, which associates library books with different subjects.

A *set* is an object that contains only pointers. Usually sets are created by extracting the pointer field from an index. Using the example of a book's index, a set could be created from the index by the definition all the page numbers that contain the words 'database', 'data model', 'data definition language', or 'data manipulation language'.

An *indexset* is a catalogued collection of indexes and sets. Every index and set **must** be associated with exactly one indexset. In addition to the indexes and sets belonging to an indexset, an indexset also contains an index catalog to maintain all the information for managing indexes, and a set catalog to maintain all the information for managing sets.

An *indexkit* is a logical grouping of an introduction, index, dictionary and thesaurus. The introduction component of an indexkit is an object which contains a textual description of the index. The dictionary component of an indexkit is an object that associates terms given in the index with their definition. It is used to assist the user in accessing the index. The thesaurus component of an indexkit is an object that associates terms given in the index with other terms. The thesaurus supports both generalization and specialization of terms in the index. The thesaurus is also used to assist the user in accessing the index. The introduction, dictionary and thesaurus components are neither managed nor maintained by UIS. Figure 1 shows the relationships among the different objects managed by UIS.

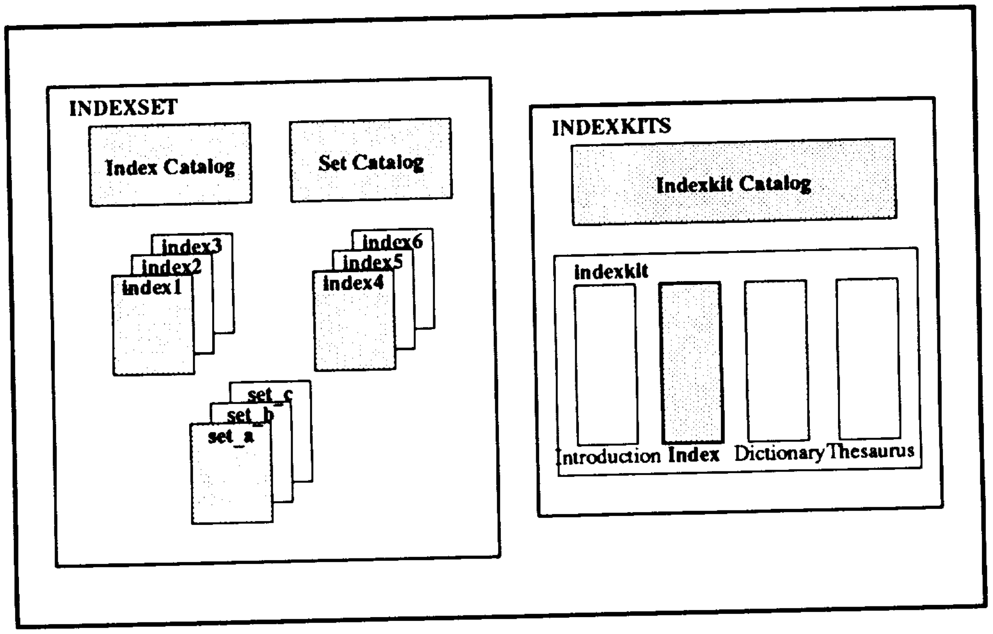

Figure 1 - Relationships among Indexes, Indexsets, Indexkits and Kitsets.

3. UIS Capabilities

UIS provides commands that allow the user to create and manipulate indexes, sets, indexsets, and indexkits.

3.1. Index Commands

UIS uses the notion of *current* objects to simplify the index commands. The user specifies which instance of an object is to be *current* i.e. to be worked on, and then subsequent commands are performed on the *current* object. The index commands rely on the existence of a *current index*, *current index row*, and *current index boolean*.

The *current index row* is set to be the tuple in the *current index* that was most recently accessed by the navigation routines (see below for a description of the navigation routines). The *current index boolean* is a boolean condition chosen by the user to assist in navigation.

UIS provides a relationally complete set of commands for indexes. In addition to commands that allow the user to create, insert into, delete from, save and destroy indexes, there are routines that allow the user to retrieve a previously created index for either modification or read only, return an index (the opposite of retrieve) and pick an index to be the *current index*.

There are commands to allow the user to navigate both forward and backward through an index, accessing a single tuple at a time. UIS provides the user with *index booleans* and *index selects* to assist in this navigation. An *index boolean* is a boolean condition defined by the user to restrict the search to a subset of the index. For example, the user could define an index boolean, camseq = LFP1010" to restrict the search on an index to only those tuples of an IUE index having LFP1010 as camera sequence number. The user can create index booleans during a user session, but they do not persist beyond the end of that session. UIS provides commands to create, modify, list (display), pick (make as current), and delete index booleans. There are also commands to allow the user to reproduce indexes. These include copying and moving an index to an indexset.

To support interfaces to programming languages, there are commands to allow the user to bind attribute values to program variables, i.e. embedding UIS commands in an application written in C. There are two commands for binding to program variables, one for binding a single attribute (column) from an index, and one that allows for binding a whole row from an index. These commands cannot be used during an interactive session.

3.2. Indexset Commands

UIS provides a few commands to manipulate indexsets. At this point a user can only create and destroy indexsets. In the future, we plan to add commands such as include copy, subset, intersect, subtract and union, and commands to copy and move indexsets.

3.3. Indexkit Commands

Although not implemented in the current prototype, several commands to manipulate indexkits have been designed for UIS. In addition to commands that allow a user to create and destroy indexkits, there are commands to allow the user to reproduce indexkits. These include copy, subset, intersect, subtract and union. Subsetting an indexkit is defined to be a new indexkit, whose components are the result of subsetting each of the components in the original indexkit. Intersecting two indexkits is defined to be a new indexkit, whose components are the result of intersecting corresponding components of the two original indexkits. Similar definitions hold for union and subtraction.

3.4. Command Summary

Tables A, B, C, and D and the end of this paper provide a summary of the index, set, indexset and indexkit commands, respectively.

4. The Design of UIS

The development of the UIS prototype was divided into several phases: the requirements phase, the design phase, the implementation phase and the testing and integration phase. This approach was taken in an attempt to resolve any conflicts in the proposed system as early as possible.

The requirements document contains a functional description of what the system should do. The purpose of the design phase is to convert the functional description of **what** the system should do into an algorithmic description of **how** the system should do it.

The design phase primarily concentrated on two tasks. First, we had to determine what information needed to be available to the system during execution and what information needed to be available from one execution to the next (persistent information). Second, we needed to translate the functional requirements of the user commands into design specifications. These two tasks were performed in a stepwise fashion to yield a cohesive and consistent design.

4.1. System Information

UIS manages and maintains four different types of objects: indexes, sets, indexsets, and indexkits. In order to do so properly and efficiently, the system needs to have available certain information about each object. As an example, consider a library: how useful or efficient would a library be if it did not have a catalog that listed what books were contained in the library, or where they were located? Probably not very useful, definitely not very efficient. In the same way that a library catalogs all the **objects** that it manages, so must UIS. This section describes which information UIS needs to efficiently manage its objects.

4.2. System Catalog — Indexes

In Section 2 we defined conceptually what an index is. To determine what persistent information we need for indexes, we need to know what an index is structurally. Structurally, an index is a table in which some of the columns are the items indexed and the last column is the pointer. An index is of type k if it has k-item tuples (columns). The format of an index depends on the internal representation of the index. Examples of formats are B-trees, R-trees, and heaps.

Given this structural definition, we see that some of the information that needs to be stored include the name of the index, its type, and its format. Other information that is necessary are the attribute or column names, their types, lengths and their location within the tuple (offset). This information is necessary when checking whether or not a user's command is valid, and to assist the system in locating and extracting attribute values. Another piece of information used to assist the system in index manipulation and validation is the index's tuple width (the total size of the tuple). In addition, we decided it would be helpful to store whether or not a given index had an indexkit associated with it. This would allow us to remain consistent with the indexkit system information (discussed later).

Because an index can have any number of attributes, we decided it would be easier to have two system catalogs. The first one contains all the information about the index except for the attribute information. A second catalog contains the attribute information. This approach was taken to simplify the catalog access routines (if a single catalog were used, the access routines would have to support variable length entries). Figure 2 describes pictorially the system catalog

Index Catalog

Index Name	Index Type	Index Format	Indexkit Name	Indexkit set	Indexspace Name	Tuple Width
FOLLET_EOTN	2	*BTREE*	*FOLLET_EOTN*	*FOLLET*	*EARLY_WORKS*	44
SUBJECTS	5	*RTREE*	*SUBJECT_LIB*	*LIBRARY*	*PG_COUNTY*	204

Attribute Catalog

Index Name	Attribute Name	Attr Type	Attr Length	Attr Offset
FOLLET_EOTN	*TERM*	*STRING*	40	0
FOLLET_EOTN	*PAGE_NUM*	*INT*	4	40
SUBJECTS	*SUBJECT_TERM*	*STRING*	40	0
SUBJECTS	*AUTHOR*	*STRING*	40	40
SUBJECTS	*TITLE*	*STRING*	100	80
SUBJECTS	*ISBN_NUMBER*	*STRING*	12	180
SUBJECTS	*LC_NUMBER*	*STRING*	12	192

Index Format =
{ BTREE, RTREE, HEAP }

Attr Type =
{ INT, FLOAT, CHAR, STRING }

Figure 2 - System Catalog Information for Indexes.

information for indexes. It contains two example indexes: FOLLET_EOTN (a book index for Ken Follet's *The Eye Of The Needle* and SUBJECTS (a library catalog of subjects which references books).

4.3. System Catalog — Indexsets

An indexset has several components (see Figure 3). It contains an index catalog discussed in the previous section, a set catalog, a transaction log, and then the indexes and sets themselves that belong in the indexset. The transaction log contains information about updates to the indexes and sets in the indexset. It is used in transaction management (currently unimplemented). UIS allows the user to explicitly specify all the buffer management constants needed for the management of indexset components. As a result, the system catalog information for indexsets must store all this information.

Before explaining the system catalog information for indexsets, we need to clarify what is meant by *databook* and *indexspace*. When defining an indexset, the user creates a logical space in which indexes and sets will belong at some point in the future. The *databook* objects are these logical spaces. An *indexspace* is the physical storage space on the disk that corresponds to the logical space defined by the databooks. Indexspaces can contain several databooks, and databooks can span more than one indexspace. Having the user be able to specify both logical space and physical space allows the user to place indexes physically near each other or logically near each other.

Given these new objects, an indexset is composed of the following components: index catalog, set catalog, transaction log, any number of databooks, and any number of indexspaces. For each of these components, the system needs to have information about the names of each of these components, the initial physical size of these components, their maximum size, and the rate at which these components can increase (when an insertion needs to be made and there is no space, an increase is requested and as long as the maximum size has not been reached, the increase is allowed).

Storing all this information creates a complicated system catalog structure. The databook and indexspace information for indexsets is stored in its own catalog. This is due to the fact that there can be any number of these objects in an indexset (similar to the attribute information for indexes). Since the directory, index catalog, set catalog and transaction log components are required for each indexset, and an indexset can contain at most one of each component, all of this information can be stored in a single catalog along with the indexset name. In addition, it was decided to have entries in this catalog for the total number of databooks and indexspaces in the indexset, to assist in retrieval from the other catalogs.

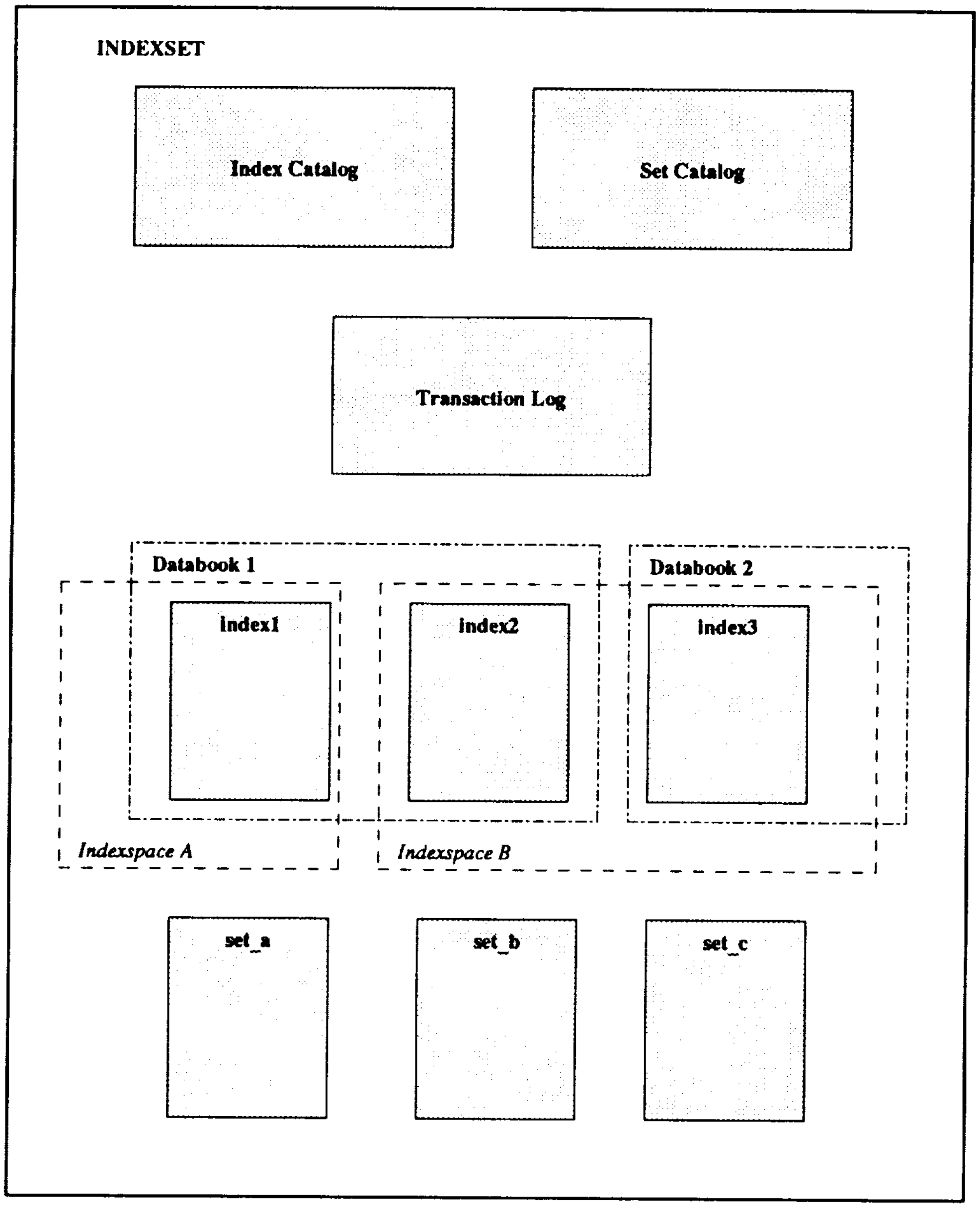

Figure 3 - Physical Structure of an Indexset

Figure 4 describes pictorially the system catalog information for indexsets. It contains two examples of indexsets: FOLLET_SET (an indexset that contains all the index information about Ken Follet's books) and SUBJECT_SET (an indexset that contains all the subject information at a specific library). For example, the index catalog component for FOLLET_SET says that the index catalog is located in the file *FOLLET_IDX*. Its initial size is 4096 bytes, and when the system needs more space for the index catalog, space is allocated in blocks of 1096 bytes. If the size of the index catalog reaches 200000, no more space will be allocated to the index catalog. The **Databook System Catalog** and the **Indexspace System Catalog** contain similar information about the databooks and indexspaces in the indexset.

4.4. System Catalog — Indexkits

As defined an Section 2, an indexkit is a logical grouping of an introduction, index, dictionary and thesaurus. In order for the system to understand this logical grouping, it needs to keep track of which instances of each component

Indexset System Catalog

ISET_name	# databook	indexspace	Directory Info		Index Catalog Info		Set Catalog Info		Log Info	
FOLLET_SET	*1*	*1*	Device Name	*FOLLET DIR*	Device Name	*FOLLET_IDX*	Device Name	*FOLLET_SET*	Device Name	*FOLLET_LOG*
			Initsize	*4096*	Initsize	*4096*	Initsize	*4096*	Initsize	*4096*
			Incrsize	*1096*	Incrsize	*1096*	Incrsize	*1096*	Incrsize	*1096*
			Maxsize	*200000*	Maxsize	*200000*	Maxsize	*200000*	Maxsize	*200000*
SUBJECT_SET	*1*	*1*	Device Name	*SUBJECT DIR*	Device Name	*SUBJECT_IDX*	Device Name	*SUBJECT_SET*	Device Name	*SUBJECT_LOG*
			Initsize	*4096*	Initsize	*4096*	Initsize	*4096*	Initsize	*4096*
			Incrsize	*1096*	Incrsize	*1096*	Incrsize	*1096*	Incrsize	*1096*
			Maxsize	*200000*	Maxsize	*200000*	Maxsize	*200000*	Maxsize	*200000*

Databook System Catalog

ISET_name	Device_name	Initsize	Incrsize	Maxsize
FOLLET_SET	*EARLY_WORKS*	*4096*	*1096*	*200000*
SUBJECT_SET	*COMPUTER_SC*	*4096*	*1096*	*200000*

Indexspace System Catalog

ISET_name	Device_name	Initsize	Incrsize	Maxsize
SUBJECT_SET	*COMPUTER_SCSPC*	*4096*	*1096*	*400000*
FOLLET_SET	*EARLY_WORKSSPC*	*4096*	*1096*	*200000*

Figure 4 - System Catalog Information for Indexsets.

make up this logical grouping. As a result, the system information needed for each indexkit is the name of the indexkit, the introduction name and its location (intro_set), the index name and its location (indexset), the dictionary name and its location (dict_set), and the thesaurus name and its location (thes_set) (Remember that the introduction, dictionary and thesaurus components are not managed by UIS). With this information, the system can efficiently execute all the indexkit commands.

Figure 5 illustrates the system catalog information for indexkits. It contains two example indexkits (they correspond to the two index examples of Figure 2: FOLLET_EOTN (an indexkit corresponding to the index, having the same name), and SUBJECT_LIB (an indexkit corresponding to the subject index of a library catalog). Indexkits are not implemented in the current prototype.

5. Run-Time Information

In addition to persistent information about each object in the system, during execution, there is a need to track additional information about the state of objects currently being manipulated or accessed by the system. Tracking such

Indexkit System Catalog

Indexkit Name	Intro Name	Intro Set	Index Name	Index Set	Dictionary Name	Dictionary Set	Thesaurus Name	Thesaurus Set
FOLLET_ EOTN	*FOLLET_ EOTN_INTRO*	*FOLLET_ INTROS*	*FOLLET_ EOTN*	*FOLLET_ BOOKS*	*FOLLET_ DICTIONARY*	*NOVEL_ DICTS*	*FOLLET_ THESAURUS*	*NOVEL_ THES*
SUBJECT_LIB	*SUBJECTS_ INTRO*	*LIBRARY INTROS*	*SUBJECTS*	*LIBRARY*	*SUBJECT_ DICTIONARY*	*LIBRARY DICTS*	*LIBRARY THESAURUS*	*LIBRARY THES*

Figure 5 - System Catalog Information for Indexkits.

information is essential to maintaining a consistent system. This information will be particularly crucial in a multi-user environment, when it is possible for different users to try to update the same data at exactly the same time. If the system were keeping no information about objects currently in the system, then it would have no way of preventing different users from updating the same data at the same time; there would be no way to guarantee a consistent system. This section describes what information UIS needs during execution to maintain consistency of the objects.

5.1. Run-Time Information — Indexes

As described in Section 3, the index routines support the notion of a *current* index. What this means in terms of execution, is that a user can have any number of indexes retrieved at a time (i.e. open and accessible), of which at most one may be the *current* index. We adopted the notion of using a tag (unique identifier) to identify indexes that have been retrieved to allow us to quickly access the indexes. As a result, anytime an index is retrieved, an index tag is assigned to it. For each index that is retrieved by the system, the tag must be readily available in order to manipulate the index. This run-time variable is designated by **Index Tag**.

A pointer into the index file must also be readily available to the system if the index is to be accessed at all. Clearly, if the index weren't going to be accessed at all, there would be little reason for the user to retrieve it. Therefore, a file descriptor for each index must also be kept as run-time information while the system is being used. This run-time variable is designated by **F_ptr**.

An index can be retrieved for either modification or read only. There are two pieces of run-time information that need to be kept related to the retrieval mode of indexes. The first is the actual retrieval mode. The system needs to know whether an index has been retrieved for modification or read only in order to prevent the user from trying to modify an index that was retrieved for read only. This is especially crucial in a mult-iuser environment, when more than one user may want to access the same index. This run-time variable is designated by **Mode**. Secondly, the system needs to keep track of whether the index has actually been modified (in the case of retrieval for modification). This information is used in the "save index" command. An index that has been retrieved for modification, but not actually modified does not need to be saved even if the user issues the save index command. Having this information available permits the system to detect these occurrences and not waste its time saving an index that has not actually changed. This run-time variable is designated by **Dirty**. **Dirty** is set to TRUE if the index has been modified, but not saved. **Dirty** is set to FALSE if the index has not been modified since the last time it was saved.

Finally, the system needs to know which indexes that are currently in the system have been created, but not saved. The reason for this is as follows. We cannot guarantee that a newly created index will be small enough to be completely contained in main memory. Therefore, when the user creates a new index, all persistent information is entered into the system catalog and the index files are created. The system needs to be able to distinguish these "created but not saved" indexes from those that either have been recently created but saved, or those that were retrieved. This distinction is necessary because if the user quits the system without saving these indexes, the system needs to know that they are to be deleted. This run-time variable is designated by **Saved**. **Saved** is set to TRUE if the index was retrieved during this user session (i.e. created sometime in the past) or if the index was created during this user session and has already been saved. **Saved** is set to FALSE if the index was created during this user session but has not yet been saved.

The remaining run-time information that needs to be available is the information found in the system catalog. Therefore, a pointer to the system catalog information is also needed at run-time. This run-time variable is designated by **SC_info**. Figure 6 shows the information that UIS needs to manage and manipulate indexes correctly.

Index Tag	**Saved?**	**Mode**	**Dirty?**	**SC_Info**	**F_ptr**
i1	*FALSE*	*MODIFY*	*TRUE*	**	*3*
i2	*TRUE*	*READ_ONLY*	*FALSE*	**	*4*

Figure 6 - Run-Time Catalog Information for Indexes.

5.2. Run-Time Information — Indexsets

As defined in Section 2, an indexset is a catalogued group of indexes and sets. Therefore, when an index or set is to be retrieved from an indexset, its system catalog information is found in the catalog components of the indexset (refer to Figure 1). At execution time, the system needs to maintain file descriptors to the catalog components of the indexset in order to be able to retrieve indexes and sets. These run-time variables are designated by **Fd_I_cat, Fd_I_attr_cat**, and **Fd_S_cat**. They correspond to the index catalog, the index attribute catalog and the set catalog components of the indexset, respectively.

If multiple indexes or sets are retrieved from a single indexset, we need to be very careful in making sure that only one set of catalog file descriptors are used for that indexset. If every retrieved index and set has its own file descriptor information for the indexsets catalog, then it would be very easy for the system to encounter read/write conflicts in the indexsets catalog components. Therefore, we need to have a way to maintain a single copy of the indexset information, and still know exactly how many indexes and sets from that indexset are currently retrieved. This suggests a need for run-time variables to count the number of retrieved indexes and sets for each indexset. This has two advantages. First, it prevents having multiple file descriptors to the indexset catalog components and prevents read/write conflicts. Second, it allows us to have the indexset retrieved for as small an amount of time as necessary. By keeping track of how many indexes and sets are currently retrieved, the system is able to return the indexset as soon as those numbers are zero. The run-time variables that designate these counts are **I_count** for indexes, and **S_count** for sets.

The remaining run-time information that needs to be available is the information found in the system catalog. Therefore, pointers to the system catalog information are also needed at run-time. These run-time variables are designated by **SC_info**, **Databook**, and **Indexspace**, which point to the different system catalog entries for the indexset. Figure 7 shows the information that UIS needs to manage and manipulate indexsets correctly.

5.3. Run-Time Information — Indexkits

There is no run-time information needed for indexkits. Because an indexkit is nothing more than a collection of system catalog information, all commands involving indexkits update only this system catalog information. As a result, the catalog is only accessed at the exact moment a request is made. There is no notion of **retrieving** an indexkit, and at some later time making some modification to it.

Fd_I_cat	Fd_I_attr_cat	I_count	Fd_S_cat	S_count	SC_Info	Databook	Indexspace
3	*4*	*2*	*5*	*0*	••		##
6	*7*	*1*	*8*	*1*	••		##

Figure 7 - Run-Time Catalog Information for Indexsets.

Table A: Index Commands

Index Management Commands		
create index	drop index	insert index
update index	move index	delete index

Index Reproduction Commands		
copy index	intersect index	subset index
subtract index	union index	

Index Searching Commands	
find term in index	build set with term
build set with list	build set with range

Index Browsing Commands		
retrieve index	pick index	save index
return index	list indexes	

Index Navigation Commands		
first in index	next in index	fetch using index
last in index	previous in index	
build index boolean	list index booleans	pick index boolean
modify index boolean	drop index boolean	
build index select	list index selects	pick index select
modify index select	drop index select	

Index Run-Time Environment Commands	
bind index column	bind index table

Table B: Set Commands

Set Management Commands		
build empty set	drop disk set	insert set
delete set	update set	

Set Reproduction Commands		
combine sets	restrict sets	sort sets

Set Browsing Commands			
retrieve set	pick set	build empty memory set	save set
return set	list sets	drop set	

Set Navigation Commands		
first in set	next in set	fetch using set
last in set	previous in set	
build set boolean	list set booleans	pick set boolean
modify set boolean	drop set boolean	
build set select	list set selects	pick set select
modify set select	drop set select	

Set Run-Time Environment Commands
bind set column

Table C: Indexset Commands

Indexset Management Commands	
create indexset	drop indexset
alter indexset	move indexset

Indexset Reproduction Commands		
copy indexset	intersect indexset	subset indexset
subtract indexset	union indexset	

Indexspace Commands	
create indexspace	alter indexspace

Table D: Indexkit Commands

Indexkit Management Commands	
create indexkit	drop indexkit
update indexkit	move indexkit

Indexkit Reproduction Commands		
copy indexkit	intersect indexkit	subset indexkit
subtract indexkit	union indexkit	