Peter F. Lemkin[1]
Yecheng Wu[2]
Kyle Upton[3]

[1]**Image Processing Section, LMMB/NCI/FCRDC, Frederick, MD**
[2]**Scanalytics/CSPI, Billerica, MA**
[3]**Program Resources Inc., FCRDC, Frederick, MD**

An efficient disk based data structure for rapid searching of quantitative two-dimensional gel databases

Fast access of two-dimensional (2-D) gel quantitative databases is important for rapid searching for protein differences between sets of 2-D gels from an experiment. The GELLAB-II system organizes corresponding spots from the gels in the database into reference or "Rspot" sets. These Rspot numeric names index fixed regions in the paged composite gel database file. This is adequate for an existing database, but has several problems. (i) Building the initial database requires guessing how much disk space to pre-allocate for each corresponding spot (*i.e.* spots from different gels). If it ever runs out of pre-allocated space during this process, it must expand the size of each corresponding set of spots copying the old database data into the new in-place on the disk. (ii) When adding new gels or editing the database, if a new spot is created, the system may also go into this expansion mode. The time spent and wasted disk space can be appreciable – depending on the size of the database (order of 100 gel database). (iii) Because each set of corresponding spots is the same size, we waste space in most spot sets since they do not require the additional space a few spot sets require which contain additional fragmented spots. We present a new low-level disk object-based structure and algorithm, paged indexed buckets (PIB), which optimizes disk space usage while having similar retrieval speed to the original method.

1 Introduction

A two-dimensional (2-D) polyacrylamide protein gel quantitative database for an experiment may consist of a large number of gels. The data is most usefully organized as sets of corresponding spots from all of the gels in the database allowing corresponding spot data to be quickly retrieved. Such a database lends itself to asking questions of the form "which spots are statistically significantly different?" When searching the database for protein spot changes, retrieval must be reasonably fast since the operation will be repeated thousands of times every time the database is searched. It also lets us edit individual spot data in the database in the context of other gels rather than in the spot list database for a single gel.

This paper presents an efficient object-based low-level disk file storage mechanism which optimizes disk access for a 2-D gel spot database. Another way of stating the problem is that a database should cluster related information so it can be stored and retrieved at the same time. We call this related information an object. The problem then becomes one of efficiently doing database storage and retrieval based on objects with the provision that objects may grow independently. This latter condition is what complicates the problem and led to our new design. We describe the problem and our proposed solution in the context of the GELLAB-II 2-D gel exploratory analysis system [1-3]. GELLAB-II quantitates gel images, pairs spots with respect to a reference gel and finally merges these paired spots into a composite gel database organized by corresponding reference spots where searches for protein differences can take place. A review of other 2-D gel quantitative database organizations is given in [2] and so will not be discussed here.

Correspondence: Dr. Peter Lemkin, NCI-FCRDC, Bldg. 469 Rm. 150, Box B, Frederick, MD 21702, USA

Abbreviations: 2-D, two-dimensional; **DB,** database; **PCG,** paged composite gel; **PIB,** paged indexed bucket; **RDB,** relational database

1.1 Notation

First let us define our notation. In GELLAB, we denote a set of corresponding quantitated spot data from different gels as an *Rspot* set of spots. An Rspot set then is an object. An object contains a set of nodes where a node is a contiguous set of bytes used for packing a spot's data. Spot data includes its: x, y position, integrated density, density range, area, mean background density, shape, *etc.* Figure 1 illustrates a 3-D composite gel database consisting of gels from all experimental conditions – not just gels representing particular experimental conditions.

Each Rspot set may be further partitioned dynamically into a subset of gels from the different experimental conditions used to make up the total composite gel database. Accessing each Rspot set independently lets us ask statistical questions of each Rspot set – treating it as multiple protein concentration distributions for that Rspot. For example, given an Rspot set of spots, we can compute the sample means and variances of each of the experimental conditions and then use these values in computing the t-statistic or other statistics to determine differences between the experimental conditions for that Rspot. By having all spot data available at all times in the database, we are flexible in which subset of gels we can use and how we compute the statistics on this data.

We should also clarify where this low-level database method fits into the analysis scheme. Gel database analysis software makes requests to the low-level database access method for spot data. We will be discussing the low-level database access method – not the higher level spot analysis.

© VCH Verlagsgesellschaft mbH, 69451 Weinheim, 1993

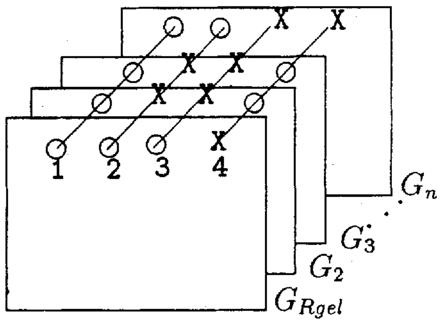

Figure 1. Illustration of a three-dimensional composite gel database (CGL). It consists of *n* gels G_{Rgel}, G_2, ..., G_n from all experimental conditions – not just gels representing a particular experimental condition. A representative gel G_{Rgel} for the set of gels is selected during database construction and is called the Rgel. Spots which are present are marked with an O and those missing with an X. Experimental condition 1 = $\{G_{11}, G_{12}, \ldots, G_{1n}\}$, experimental condition 2 = $\{G_{21}, G_{22}, \ldots, G_{2n}\}$, *etc.* The set of corresponding spots from different gels is called an Rspot set object. Rspots 1, 2, 3, 4, *etc.* are illustrated. Rspots sets which are not present in the Rgel are called eRspot sets (*e.g.* Rspot [4]). Missing spot positions are extrapolated and assigned zero density values.

1.2 The problem

When a gel database of Rspots is extended by adding gels or edited so spots are added, additional space may be needed in a particular Rspot set. There are several methods of organizing this type of data so new spots can be accommodated. We list two here. (i) The first method stores each spot in a separate gel spot list file, or paired spot list file or a superfile storing all of the paired spot lists. Adding a spot implies extending an individual file. However, this implies visiting several files or tracing through the pair spot lists with multiple disk seeks for each Rspot set access. This is not very efficient for processing large numbers of gels and large numbers of spots. (ii) The second method, used in the original GELLAB-II system, preallocates contiguous disk space for each Rspot set based on a factor ≥ 1.0 of the number of gels to be saved in the initial database. Each Rspot set then consists of a list of spot nodes which is kept sorted by a relative node pointer contained in each node. Keeping all gel data available for statistical calculations rather than computing average values for the different experimental conditions of gels means that the database can be easily repartitioned to a different subset of gels or subset of spots from those gels without extensive rebuilding of a sub-database.

This disk allocation strategy meant that only one disk access was required to access the entire Rspot set. However, if the Rspot set ever grew, all Rspot sets would have to be expanded to add space to any single Rspot set – an inefficient expansion operation as well as wasteful of disk space. An Rspot set can grow if new gels are added to an existing database or the user manually edits in new spots to an Rspot set missed by the automatic spot finding/quantitation software. Figure 2 illustrates the original GELLAB-II database design and the problem of expanding the database if any Rspot set grows.

The new method we describe here gives us the flexibility of method (i) but with the access efficiency of (ii). We describe the algorithm and then give some benchmark comparisons between the original and the new method. The original GELLAB-II paged composite gel (PCG) database data access method is described in [1,2,4] and so will not be described here.

2 Methods

2.1 Overview of paged indexed bucket algorithm

The new database method is called the paged indexed bucket (PIB) database. From a gel database user's point of view, the PIB database consists of an indexed list of objects illustrated in Fig. 3. It uses a single-level primary index which indexes database buckets with bucket chaining on overflow. Korth and Silberschatz [5] have a nice discussion of database index methods and the use of buckets in database files. In our context, a bucket is a contiguous region of memory (whether on the disk or in memory) which holds one or more nodes of data. Then, each Rspot object consists of a set of spot nodes accessed as a contiguous array belonging to that object. This is illustrated in Figs. 4.a and b.

In the PIB design, there are two types of buckets: primary and secondary. Fixed size nodes are allocated in buckets and are then manipulated by the gel analysis program using this database. The gel database program can sometimes optimize the I/O rates by requesting that the primary bucket size be exactly that required for each object so that multiple disk accesses are not required. If additional nodes are ever needed to be added beyond the capacity of a primary bucket, then additional secondary buckets are added to the end of the disk file as overflow secondary data buckets. Secondary buckets of an object are not contiguous on the disk but are contiguous in the object cache in memory. If there are few overflows, then additional overhead would be low.

As illustrated in Fig. 3, an object set of nodes is kept in a singly linked list of buckets in a **.pib** node data file. The

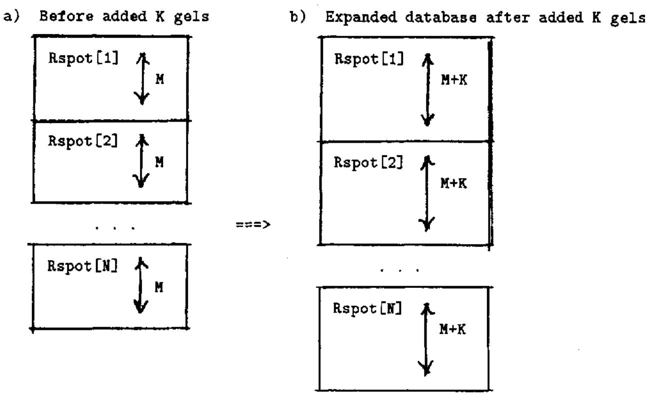

Figure 2. Structure of the original GELLAB-II PCG database where all Rspot sets are the same size (a) database of *M* gels containing *M* spot nodes/Rspot set; (b) expanded database after adding *K* more gels or if any particular Rspot [*i*] was increased to *M* + *K* spots. Note that the database is contiguous on the disk and that each Rspot set is the same size. This makes object access from the disk a simple one-step operation.

object index pointing to primary buckets is saved in a **.idx** file and is kept in memory during PIB operation. The spot node data is itself saved in a **.pib** data file and is cached to memory as needed. The caching mechanism is flexible so that a single cache or multiple caches can be used. On gel database program startup, the entire object index is read into and kept in memory during processing. This means that we can quickly look up the disk addresses of the primary buckets for all objects. This object index is the primary index and there is no secondary index. Rather, overflow or secondary buckets of an object are chained together in the **.pib** file itself. So the only way to access a secondary bucket is to have read the previous bucket which points to it.

2.2 The initial primary and secondary buckets

Initially, when a new object is created, we create a primary bucket (of size N_{pb})with a NULL secondary bucket pointer (and secondary bucket size N_{sb} of 0). That is, the first N_{pb} nodes will be put into the primary bucket. If the initial number of primary nodes is known, the primary bucket can be created with this number of nodes – the optimal allocation. So no secondary buckets are required. (In GELLAB-II, we estimate the number of nodes in a primary Rspot object as the number of gels in the database). When an object is accessed, all of the node buckets for that object are read into and assembled into one of the PIB node caches. As the database is

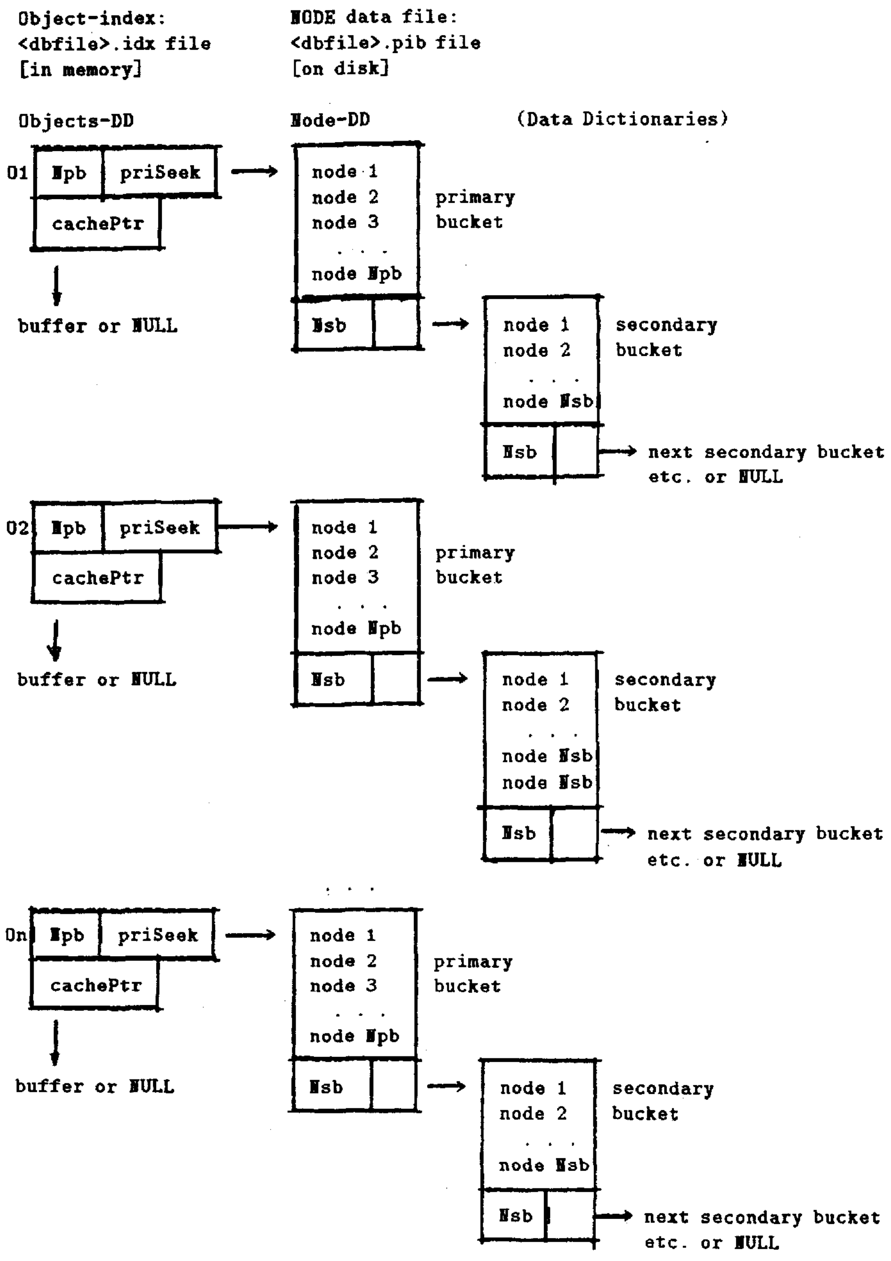

Figure 3. Schematic of PIB data structures. Each composite Rspot set object O*i* has a 3-tuple (N_{pb}, *secSeekPtr, cachePtr*). N_{pb} is the number of nodes in the primary bucket. The *secSeekPtr* is the pointer to the start of the primary bucket on the disk and *cachePtr* is the cache pointer if the object is in the cache.

being constructed, new bucket descriptors are added to the object cache as required. When the object in the cache is written back to the disk, each bucket region of the cache is written out separately – to a different part of the disk. At the end of each disk bucket, a two-word link (N_{sb}, *secSeekPtr*) is also written to the next secondary bucket if it exists (illustrated in Fig. 3). Note that while the secondary buckets for an object are scattered over the disk, all node data in the object cache is contiguous in the cache node buffer. Together, these links specify the complete node data for each object set.

Normally, to access any node in an object set, all of the nodes must be read into the object cache. The cache holds only one object. Therefore, the size of the cache must be at least the size of the largest object and grows dynamically as required. When the nodes are scattered in several buckets, this can be more time-consuming since all buckets have to be read to access the entire object set of nodes and the disks seeked separately to each bucket. Although we do not operate the PIB this way, we could optionally limit access to just the primary bucket if we can ensure that the data we want resides there. This would then allow us to do object caching in one disk access – but at the cost of not retrieving the entire object. The PIB algorithm can work with multiple object caches. If memory is in short supply, it can fall back to one cache and free the memory used in the other caches.

Adding more nodes than the current object cache capacity causes a new secondary bucket to be created in the cache and the cache buffer to be regrown. We generally set a secondary bucket size large enough to hold several nodes. This prevents too much disk fragmentation if we expect more than one additional node per object set.

The cache maintains a "dirty cache" flag which is checked when either: (i) a new object is to be read into the cache; or (ii) the database is to be checkpointed or exited. If the flag is set, then the object is flushed back out to the disk file. The dirty flag is set if the data in the cached object changes since it was last read or is being created.

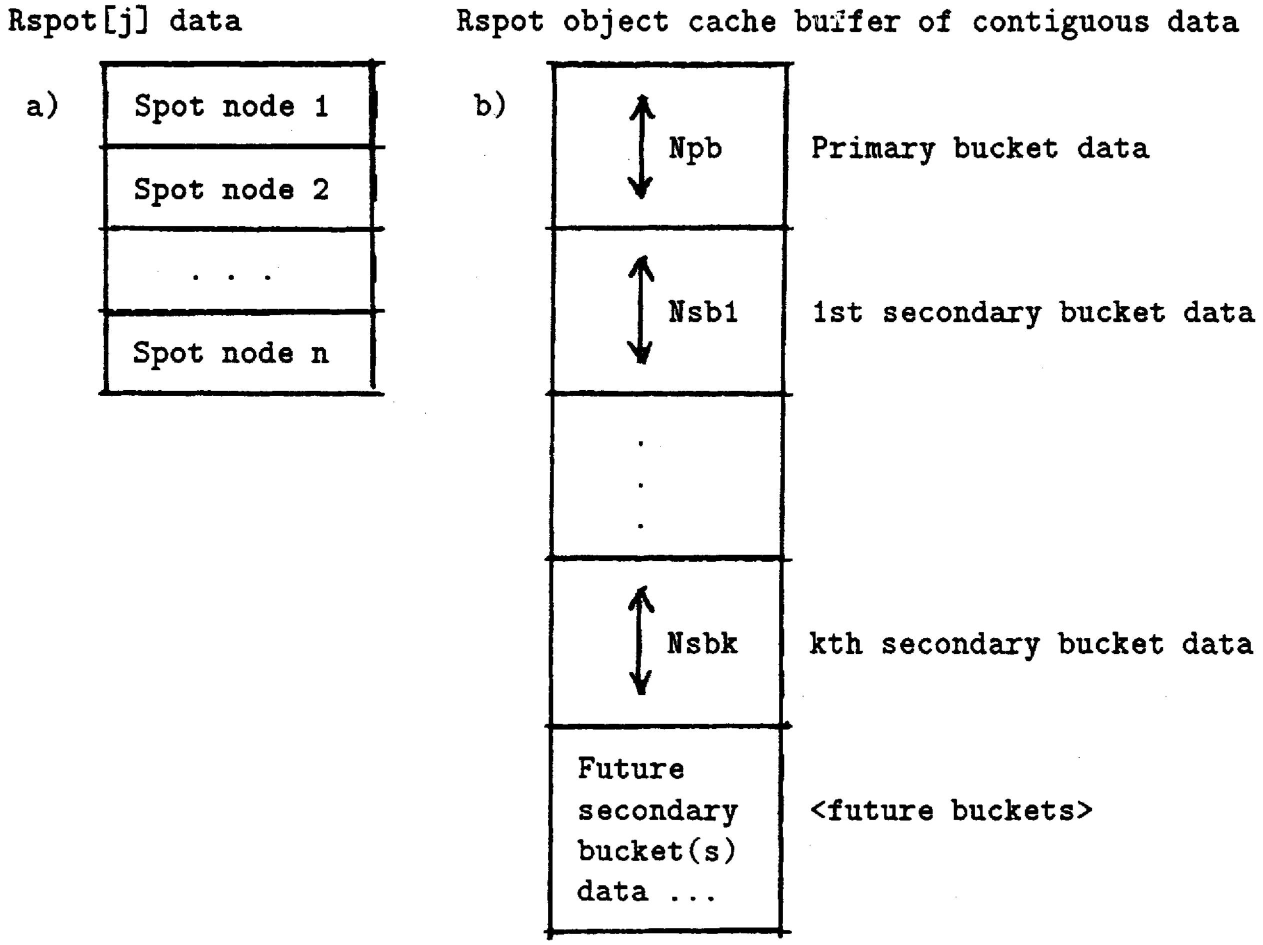
Rspot[j] data
a)
Spot node 1
Spot node 2
. . .
Spot node n
Rspot object cache buffer of contiguous data
b)
Npb
Primary bucket data
Nsb1
1st secondary bucket data
Nsbk
kth secondary bucket data
Future secondary bucket(s) data ...
<future buckets>

Figure 4. Structure of the contiguous Rspot list of spot nodes for some Rspot *j*. Each node is N_{node} words long (16 in GELLAB-II packing over 30 spot features). This list occupies a contiguous memory space. The nodes are sorted within this space by the gel database program using a next-node index in each node relative to this space. (a) Illustrates the abstract Rspot structure for *n* nodes. (b) Illustrates how the contiguous Rspot object *n* node cache buffer is broken up into *k* bucket data regions ($k < n$) which are then gather-scattered in read/write disk operations of the PIB node file.

2.3 Deleting a node

Deleting a node from an object is up to the gel database program which uses the nodes in the object as it wants. No semantics are imposed on the contents of a node except that the first word of each node is non-zero if that node is active data. We use the convention that the word is zero if the node is inactive or deleted. In GEL-LAB-II, deleted Rspot nodes are removed from the gel database linked list of nodes. However, the deleted node is still in the object bucket (and cache). The PIB subsystem zeros the data in the node when told to delete a node.

A deleted node can be recycled as a new node index when the low-level PIB database gets a request for a new node. We only need to check the first word of each node to find a free one (*i.e.* previously deleted) if it exists. So the gel database node usage adopts this zeroed-node convention. If no deleted node is found, we create a new node at the end of the cache buffer (adding a bucket if required to make more node space). Deleted objects are not reused in the PIB file. However, when a PIB database is coalesced (to be discussed) the object space is freed. Once a cache buffer and its bucket-list is grown, it never shrinks since we keep reusing the cache for caching other objects. This is more cost effective than constantly allocating-deallocating the cache.

Object and node low-level PIB access functions do not create objects and nodes if they do not already exist so they must be explicitly created with other low-level PIB functions.

2.4 PIB database handle

Opening the PIB database creates a new database handle – a record of all pertinent information for that database. Multiple databases can coexist in the same gel database program – with different size nodes and buckets, *etc.* and may be optimized differently for different purposes.

2.5 Multiple databases

Because the PIB database handle is allocated dynamically, multiple copies can be created. Multiple database handles are necessary for doing operations such as coalescing a PIB database or in changing the size of a node by copying and expanding/contracting nodes in the new copy. The latter can be useful, for example, if the definition of a node is changed by adding additional spot features (and increasing the size of a node). Then the database can be copied to the new format. This is especially important if we have a large amount of time invested in editing an old database and do not want to have to redo the editing just because we went to a new format.

2.6 PIB database files

There are three required PIB database files: the **.idx** (primary index file) and the **.pib** (node database), and the **.mem** (a separate memo database file which is not discussed here). The **.idx** file must be read first in order to access primary object buckets in the **.pib** file. The memo database contains optional ASCII string data associated with each object and has a similar dynamic allocation/coalesce mechanism. (We actually are specifying the file extensions here – each database has a basename. For example for a basename **fas**, the three files would be: **fas.idx, fas.pib** and **fas.mem.**) Each object can have an optional arbitrarily large memo string associated with it. The memo string is identified by a non-zero memo number. In addition, objects can share the same memo string by having the same memo number. These strings are kept in a common **.mem** database file but their object memo number index is part of the**.idx** database file. If a memo is deleted from any object, the memo number is deleted from all other objects which shared it. Furthermore, the string space is lost in the **.mem** memo string file until it is compressed away with a coalesce operation.

2.7 Coalescing PIB database

After the database is constructed, it may be optimized, *i.e.* coalesced, by being copied into another database file. Multiple buckets for each object in the initial database are coalesced into single primary buckets for those objects of size N_{pb} (r) (r being the |*objectset*| and N_{pb} (r) is part of the primary index). Since multiple databases may reside in the same program, the secondary buckets can be periodically optimized away using a *coalesce* operation. When coalescing a database, we read in the primary and secondary records for each Rspot set from the old database and then write each Rspot set out as a larger primary bucket in the new database with no secondary buckets.

2.8 Dynamic caches

Although the PIB algorithm can be implemented with a single object cache, we do allow a dynamic number of object caches to be used. This can be useful if there is a need to simultaneously keep multiple objects in memory and there is enough memory to support multiple caches. An example of this would be when several objects need to be repeatedly accessed such as with some clustering algorithms. By caching all of the objects being clustered, we help avoid disk thrashing. In this case, the gel database program just accesses the objects using a different low-level function call which puts each object into a separate cache. Finally, it notifies the PIB low-level database manager when it is done with these objects so they can *all* be flushed and the cache buffer space freed.

2.9 Byte-order and database portability

Because byte order of binary data can cause problems with portability when reading data with computers with different byte order, we enforce a standard byte order on all binary data. We use the big-endian (Sparc and XDN standard architecture) as the default byte order. The PIB I/O subsystem automatically translates big-endian to little-endian (*eg.* Intel to Sparc computer architectures) as required for the **.idx** data on a little-endian system.

The PIB node data byte order can be handled either by the gel database program or optimally by calling other PIB access functions.

2.10 PIB data structures

In order to clarify the relationships between the different PIB database classes: objects, buckets, cache and nodes, we now examine parts of the PIB data structures. Table 1 shows key parts of the C language declarations for data structures used in implementing the Paged Indexed Bucket algorithm. Each database is referenced by a dynamically allocated *handle* structure **pibDB_t**. The **pibDB_t** structure is composed of the PIB index, cache(s) and a list of objects. A cache maintains a bucket list as well as buffer space for node data. The low-level database functions manipulate these structures within **pibDB_t**. Six main structure types are used in this database handle composition.

asciiDD_t Ascii data dictionary class
bucket_t Cache class for storing an Object's node bucket data class
cache_t Cache class for storing an Object's node data (in memory) class
pibObject_t Object index entry class
pibIndex_t Primary index limits class for the Node data base class
pibDB_t Paged Indexed Bucket node database handle class

The **pibIndex_t** class defines the **.idx** index file and index sizes. Figure 5 illustrates the structure of the index file.

Table 1. Key data structures of PIB algorithm[a)]

```
typedef struct __ASCIIDD__
   {/* Ascii data dictionary class */
      char *fieldNames;          /* names of each field. Special names are:
                                  *    "$BODD", "nKeys=", itoa(nKeys)
                                  *    "$EODD", "",        ""
                                  */
      char *fieldType;           /* type of each field "BYTE", "long" etc. */
      char *fieldValue;          /* value of each field */
   } asciiDD_t;


typedef struct __BUCKET__
   {/* Cache class for storing an Object's node bucket data  */
     long
       fseekBucket;              /* (BYTE *) disk bucket pointer, -1 is undefined */
       nNodesInBucket,           /* list of bucket sizes (in nodes) of a bucket */
       bOffset;                  /* BYTE offsets of bucket from start of buf*/
   } bucket_t;


typedef struct __CACHE__
   {/* Cache class for storing an Object's node data in memory. */
     unsigned long
       *buffer;                  /* pointer to memory cache buffer if ! NULL*/
     bucket_t
       *bucketList;              /* list of [1:nBucketsUsed] buckets */
     long
       bufferSize,               /* total size (in BYTES) of cache */
       objectNbr,                /* object nbr associated with cache */
       maxNodesAllocated,        /* across all buckets in cache */
       maxBucketsAlloc,          /* # buckets allocated in cache */
       nBucketsUsed,             /* # of buckets used in this Object */
       dirtyCacheFlag;           /* set if cache entry is dirty and needs to be flushed */
     . . .
   } cache_t;


typedef struct __PIBOBJECT__
   {/* Object index entry class */
     cache_t
       *cachePtr;                /* (dynamic) if !=0 cache idx associating
                                  * caches with the object */
     /* The following data is written to the .idx file */
     long
       priFseekPtr;              /* primary DSK Buckets seek ptrs*/
     short
       priBucketSize,            /* size (in NODES) of the primary bucket */
       nNodesUsed;               /* total # of nodes in object  */
```

```
    BYTE
      deletedFlag;             /* if '*' then OBJECT is deleted. */
      . . .
  } pibObject_t;


typedef struct __PIBINDEX__
  {/* Primary index limits class for the Node database  */
    asciiDD_t
      **asciiDD;               /* Ascii data dictionary describing .idx file*/
    asciiDD_t
      **nodeAsciiDD;           /* [opt]. data dict. describing PIB node data */
    char
      idxDBfile[256];          /* pibDBfile&".idx" index file name */
    long
      startOfNodeAsciiDD,      /* fseek start of node asciiDD in .idx */
      startOfBinaryData,       /* fseek start of binary data in .idx */
      nObjectsAllocated,       /* sizeof pibDB->obj[] alloc - not what is used */
      nObjects,                /* # of objects in the index [1:nObjectss] */
      indexDirty;              /* set if ever added objects or changed */
    . . .
  } pibIndex_t;


typedef struct __PIBDB__
  {/* Paged Indexed Bucket node database */
    pibIndex_t
      index;                    /* primary index state for node DB */
    pibObject_t
      *obj;                     /* primary objects [1:nObjects] index */
    cache_t
      **cache;                  /* list of node cache[1:maxCaches] */
    FILE
      *fpPib;                   /* pibDBfile&".pib" file pointer for node file */
    char
      pibDBfile[256],           /* pib Base file name path (no extension) */
      pibFile[256];             /* pibDBfile&".pib" node file name */
    long
      nextFreeByteInFile;       /* next free byte to alloc in .pib file. */
      lastCacheNbr,             /* last cache # allocated */
      maxCaches,                /* # of caches allocated. Start at 1. */
      nWordsPerNode,            /* sizeof(node)/sizeof(long) */
      nBytesPerNode,            /* sizeof(node) */
      nNodesPerPrimaryBucket,   /* initial Pri. bucket size */
      nNodesPerSecondaryBucket, /* default size of secondary buckets */
      lockFileFlag;             /* lock status of .idx PIB DB file  */
    . . .
  } pibDB_t;
```

a) To simplify these data structures for the paper, we have omitted some of the less important details. This is denoted by "...". The **pibDB_t** database handle is dynamically allocated. The primary index is kept in memory and is read and saved from/to the **.idx**

The index file is used only when the database is initially opened, check-pointed or closed. An ASCII data dictionary defined by **asciiDD_t** at the front of the **.idx** index file describes the structure of the remaining index file so that it is portable and can reflect changes in the number of objects, *etc.* The **.pib** node data file on the other hand is opened when the database is opened and remains open until the gel database closes it. When an Rspot object is to be accessed, the low-level database functions check the **pibObject_t's cachePtr.** If it is not NULL for that object then it is in the cache, otherwise the old contents of the cache are flushed and the new contents loaded from the **.pib** file for this object. Figure 6 illustrates a typical organization of the **.pib** file as consisting of buckets allocated as they were required. The structure of an individual bucket is illustrated in Fig. 3 which also shows the (N_{sb}, secSeekPtr) pointer to the successor bucket at the end of each bucket.

3 Results

In order to test the efficacy of the new algorithm, the GELLAB-II system was built as two different versions with all codes exactly the same except where it accessed the low-level database handler. These two versions used the original PCG DB method and the new PIB method. Then different databases were constructed and various operations performed. The first was a 12-gel leukemia database of 1124 Rspots with an average of about 800

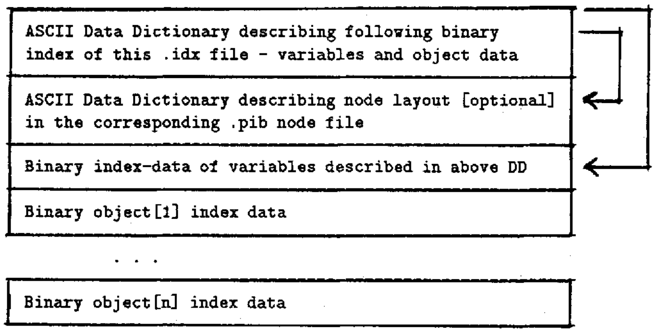
ASCII Data Dictionary describing following binary index of this .idx file - variables and object data
ASCII Data Dictionary describing node layout [optional] in the corresponding .pib node file
Binary index-data of variables described in above DD
Binary object[1] index data
. . .
Binary object[n] index data

Figure 5. Structure of **.idx** PIB index file. The index entry for an Rspot object points to where the primary bucket for that object is on the disk, the primary bucket size and to the cache if the object is in the cache. The use of ASCII data dictionaries helps make this database more portable.

spots/gel and the second was a 52-gel serum database of 2003 Rspots with an average of about 1500 spots/gel. We then compared file sizes, database build and search times. These initial benchmarks are summarized in Table 2. The new PIB database uses approximately half the disk space of the original PCG database format. Access times are roughly twice as long for PIB.

Coalescing the 12-gel database **.pib** file did not take appreciable time – roughly double the time to do a simple search. Coalesce time for the 52-gel data base was also approximately double search time. This makes sense since the coalesce operation is just a copy operation – repeatedly reading an object from the old database and writing it to the new one. Then after coalescing the **.pib** 12-gel database file, the search was slightly faster (17 versus 22 seconds). We have also observed that most pri-

Table 2. Comparison of original PCG and new PIB methods for representing 2-D gel quantitative databases[a)]

(a) 12 gel database of 1124 Rspots with an average of about 800 spots/gel:

	Original PCG method	Paged-Index-Buckets (PIB)
DB size (Mb)	6.18(.pcg)	0.04(.idx) + 1.63(.pib) + 1.34(8K .sdd) or 2.27(16K .sdd)
DB size (ratio)	1.0	0.48(8K .sdd) or 0.64(16K .sdd)
Building DB (time)	13:48 [08:57]	15:38 [07:40]
Sequential search (time)	00:05 [00:04]	00:09 [00:08]
EXTRAPOLATE (adds missing spots to existing DB, but before COALESCE)	00:32 [00:31]	01:51 [00:51]
SEARCH (time) (sorts existing database)	00:04 [00:03]	00:22 [00:17]
COALESCE DB (time)	--	00:58 [00:18]
SEARCH (time) (After COALESCE)	--	00:17 [00:16]

(b) 52 gel database of 2003 Rspots with an average of about 1500 spots/gel:

	Original PCG method	Paged-Index-Buckets (PIB)
DB size (Mb)	14.92(.pcg)	0.06(.idx) + 6.81(.pib) + 2.26(16K .sdd)
DB size (ratio)	1.0	0.61
Building DB (time)	1:00:11[0:57:56]	2:23:02 [1:30:58]
Sequential search (time)	00:18 [00:16]	00:21 [00:20]
EXTRAPOLATE (adds missing spots to existing DB, but before COALESCE)	03:16 [03:11]	05:58 [04:42]
SEARCH (time) (sorts existing database)	--	01:02 [00:58]
COALESCE DB (time)	--	02:11 [00:50]

a) Rough benchmark times are represented as run-time [cpu-time] in minutes:seconds. In part (a) a 12 gel leukemia gel database with roughly 800 spots/gel was used in these tests [Thanks to Dr. Eric Lester for use of this data]. In part (b) a 52 gel FAS serum gel database with roughly 1500 spots/gel was used in these tests [Thanks to Dr. James Myrick for use of this data]. The FAS database did not have any GELLAB-II eRspots so all gels fit into the primary bucket. The PIB versions were built to hold up to either 8,000 or 16,000 Rspots while the PCG version could hold up to 8,000 Rspots.

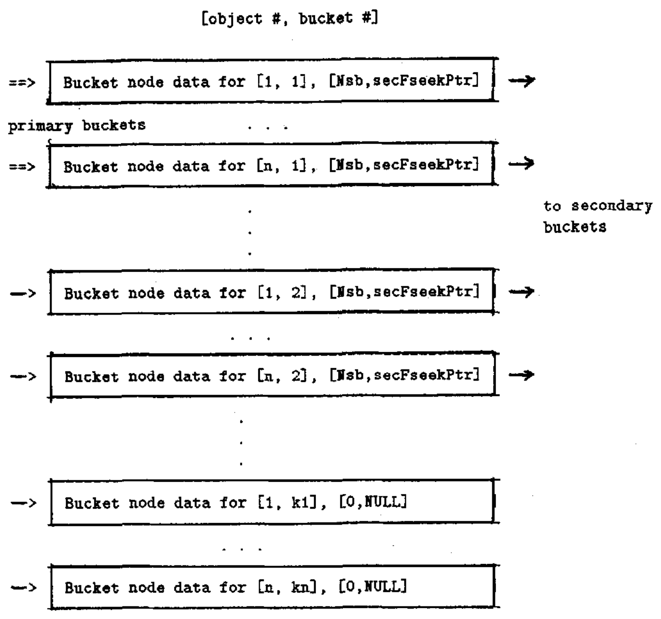
[object #, bucket #]
==> Bucket node data for [1, 1], [Nsb,secFseekPtr] →
primary buckets
...
==> Bucket node data for [n, 1], [Nsb,secFseekPtr] →
to secondary buckets
—> Bucket node data for [1, 2], [Nsb,secFseekPtr] →
...
—> Bucket node data for [n, 2], [Nsb,secFseekPtr] →
—> Bucket node data for [1, k1], [0,NULL]
...
—> Bucket node data for [n, kn], [0,NULL]

Figure 6. Structure of **.pib** PIB mode data file. Object primary buckets are generally clustered in the front of the file while secondary buckets and new primary object buckets are inserted toward the end of the file. Pointers from the primary index are denoted by ==> while secondary pointers are denoted by —>.

mary buckets are at the front of the **.pib** database file. This is consistent with the ways the data is assembled in the PIB database since the initial Rspot objects are all defined by spots contained in the reference gel. With the larger 52-gel database, the ratios of the disk space and database processing times were similar to those of the smaller database.

4 Discussion

As expected, we see from Table 2 that the efficiency (as measured by access time) of the PIB database implementation of a 2-D gel database is somewhat less than for the original PCG DB method. This is because of the existence of secondary buckets for some of the objects. However, the PIB database file takes less space – and the time trade-off is not severe. Because the ratios of disk space and database processing times are similar for the large and small databases, it would seem that the PIB algorithm would scale linearly. By further benchmarking and code optimization, we feel that the PIB performance can be optimized to nearly that of the more efficient GELLAB-II PCG algorithm. The main point, however, is that the performance is acceptable and we get the benefit of an easily extensible object database. Additional benchmarking should be done to get a more accurate estimate of the differences between the two methods.

4.1 Coalesce strategies

After expanding a PIB database, if new gels were added, it may be useful to coalesce it to improve efficiency. The question arises "at what point should coalescing be done?" We would suggest that it would be most productive just after a database was built and possibly after major editing. By coalescing immediately after the composite gel database is constructed, the database is optimized before doing searches or additional analysis. When editing, we add (and delete) spots in individual gels, possibly changing the size of some Rspot objects. Unless a massive editing effort is performed, editing will probably not make much difference in overall PIB database efficiency. Alternatively, by looking at the statistics of the average number of secondary buckets/primary buckets in the cache usage, we could estimate when to coalesce the database and possibly even invoke it automatically. The PIB structure allows extending an individual object without any penalty on other objects. If more nodes are needed than will fit in the last current bucket of an object, it simply adds another bucket. So the worst case is that the object gets additional secondary buckets. If this happened to many objects and performance degrades, the PIB database can always be coalesced.

4.2 ASCII data dictionaries for portability

The index database has two ASCII data dictionaries. The first is for the index **.idx** file itself and describes its size and contents. This makes the database more portable as other programs can read and decode the index data structures. The second ASCII data dictionary describes the structure of a node used as the basic building block of the **.pib** file. This is also useful if the structure of a node changes with the version of the gel database software since it could still read an old database and remap the data to the format of the new node structure.

4.3 Efficiency of PIB *versus* standard relational databases

The B^+ balanced tree database structure [5,6] is an optimal data structure for many types of data. It keeps the tree depth and seach times to a minimum. However, it requires at *least* two disk accesses and probably more as the number of records (Rspot nodes) exceeds the size of the B^+-tree internal node size. Object data is therefore still not as optimally clustered as with the PIB method when coalescing is used, which can reduce object access to one disk access.

4.4 Efficiency of PIB *versus* standard relational databases

Since the PIB methodology clusters related spot node entities in Rspot set objects for efficient retrieval, it would appear to be much more efficient than if implemented using most relational databases (RDB). A standard RDB could store each spot node in a table organized by gel. Then to access all of the spots in an Rspot set, the RDB system must access each of the gel tables to get each record and then assemble them (*i.e.* "join") in a new table. Alternatively, a table could be organized so a record is an Rspot set with each number of fields being separate gels. However, this is awkward since it may be expensive to extend tables to add new gel spot nodes. Unused table entries would also be wasteful of disk space.

However, if the RDB implements record clustering, then its efficiency could be optimized to be similar to that of PIB. By record clustering, we mean the clustering of the node data – not just the clustering of indices to the data. Index clustering still does not give the efficiency we require. One always wants to avoid multiple disk accesses for reading and writing an object which is clearly less efficient. Elmarsi and Navathe [6] describe a clustering index method with separate fixed-sized blocks (similar to fixed-size buckets) for each group of records belonging to the same cluster (*i.e.* object). PIB is similar to this concept except that PIB buckets need not be of a fixed size. Variable bucket size can then be used in optimizing disk usage when a PIB database is coalesced.

The PIB object-based database design is not optimal for general RDB problems because there is no secondary index (although it could be added). However, it would be useful for object-based databases where objects contain related subobjects and when objects can grow arbitrarily large at any time. Hurson *et al.* [7] analyze different problem domains of database management systems and suggest a model for predicting which domains will work better with the relational model and which with the object-oriented model. They indicate that the relational model, while simple and powerful, works best with data where "relations must be at least in first normal form, which inhibits the direct representation of multivalued or set-valued attributes". Although objects can be mapped to first normal form, they suggest that better performance can be achieved using object-buffering and object-caching – both of which are used in the PIB design.

4.5 Future improvements

The Rspot set data in an object could be reorganized for faster I/O. By having the gel database program physically reorganize the node data in the primary and secondary buckets so all nodes of interest are in the primary bucket, it would then be possible to read only the primary bucket for that object. The high-level database program would then instruct the low-level PIB database functions to read only the primary bucket for an object. Yet, the full object data is available when needed using the standard object retrieval method. Low-level PIB functions exist to read/write only the primary bucket, but are not currently used in the GELLAB-II database code.

It is not currently possible to read an arbitrary bucket from the middle of the bucket list because only the primary bucket is accessible from the index for each object. Currently, all buckets in the list for an object are read into the cache. If a specific node were desired, the PIB algorithm might be extended to read only as much as it needs in order to reach that data. Using a secondary index might help, but that could add a great deal of memory overhead for marginal gain in an object-based system. Overall the PIB object-based design improves the overall performance of accessing 2-D gel quantitative data without major overhead and allows smaller composite gel database files to be used. Although this clustering scheme may be available in some expensive relational database systems, the concept is simple enough to implement for small systems which can not justify this cost.

Received June 2, 1993

5 References

[1] Lipkin, L. E. and Lemkin, P. F., *Clin. Chem.* 1980, *26*, 1403–1413.
[2] Lemkin, P. F. and Lester, E. P., *Electrophoresis* 1989, *10*, 122–140.
[3] Lemkin, P. F., in: Endler T. and Hanash, S. (Eds.) *Proceedings of 2-D Electrophoresis.* VCH Weinheim 1989, pp. 52–57.
[4] Lemkin, P. and Lipkin, L., *Comp. Biomed. Res.* 1981, *14*, 407–446.
[5] Korth, H. F. and Silberschatz, A., *Fundamentals of Database Systems*, The Benjamin and Cummings Pub. New York 1989, pp. 802.
[6] Elmasri, R. and Navathe, S. B. *Database System Concepts,* McGraw-Hill, New York, NY 1991, pp. 694.
[7] Hurson, A. R., Pakzod, S. H. and Cheng, J.-B., *IEEE* Computer 1993, *26*, 48–60.