The Magma Database file formats

Andrew Gaylard, Brent Pinkney, and Mart-Mari Breedt
Johannesburg, South Africa

15th May 2006

1 Summary

Magma is an open-source object database created by Chris Muller, of Kansas City, U.S.A. Magma is implemented entirely in Smalltalk. These notes explain the formats of the files used by Magma to store its data. This information was obtained by reverse-engineering the Magma source code with Chris Muller's permission. It is sufficient[1] to allow for a clean-room implementation of reading and writing the raw data files. This work was done using the version MagmaTester-157 and its dependencies.

2 Background

The core idea behind Magma – and other object databases – is to allow for objects to be saved to stable storage in a database ("persisted") and later re-created. In Magma, object persistence is "transparent", that is there is no need to explicitly map any object from its memory representation to its file-based representation. Magma uses Smalltalk's detailed class inspection features ("reflection") to persist objects automatically. It makes use of lightweight "proxy-objects" to re-create the objects on-demand from the database: when a proxy-object is "agitated", that is when any access to it is performed, the proxy-object is replaced by the dynamically-created real object.

Magma also allows objects to be persisted in collections, which are analogous to tables in a conventional relational database. These collections may be indexed on one or more given member-variables to facilitate rapid (that is, better than linear) access.

3 Object identifiers

In Magma, each object is identified by an "object identifier", abbreviated to OID. In the revision of Magma used for this work, OIDs are 48 bits wide. They are analogous to pointers in languages such as C, but exist not only in the computer's memory, but also in Magma's data files. OIDs are central to the way Magma manages its object data. There is a one-to-one mapping between instances and OIDs.

OIDs are used not only to identify instances. For certain objects which are both small and frequently-used, the OID not only identifies the object, it also *contains* the object. This is possible because 31-bit integers and 32-bit IEEE-754 floating point numbers, among other classes, fit easily into a 48-bit OID. To make this scheme work, the 48-bit OID-space is divided into ranges which are set apart for each class of object that can be represented. The

[1] well, nearly sufficient

Table 1: Ranges in OID-space

start of range	end of range	object
0	0	*nil*
1	1	*false*
2	2	*true*
3	3	Signifies an unused hash index slot
4	$4 + 2^{16} - 1 = 65539$	DB character set
65540	$4 + 2^{16} + 1000 = 66540$	Reserved for future use
66541	$66541 + 4000000 = 4066541$	New-object OIDs; up to four million and one in a single commit
4066542	$2^{48} - 2^{31} - 2^{32} - 1$	User objects
$2^{48} - 2^{31} - 2^{32}$	$2^{48} - 2^{31} - 1$	32-bit IEEE floating point numbers
$2^{48} - 2^{31}$	$2^{48} - 1$	"Small" (31-bit) integers

details are contained in table 1, which was obtained from the MaOidCalculator Smalltalk class.

4 The `objects.idx` file format

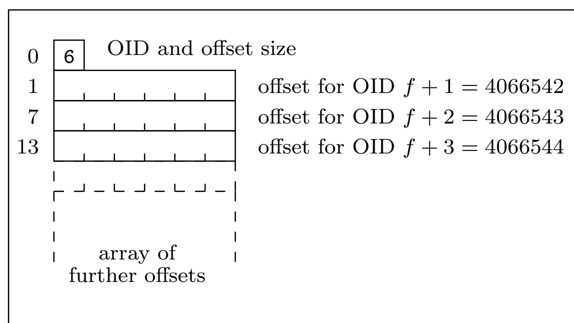

Figure 1: Binary layout of the `objects.idx` file

This file can be considered to be the master index. It is used to map any given OID to an offset in the `objects` file.

The first byte in the file is the OID width in bytes. For this version of Magma, it is always the value 6. Refer to figure 1.

The rest of the file is an array of offsets, each one OID-width (i.e. 6) bytes wide. Each array entry corresponds to an OID. The array is offset from the constant `firstUserObjectOid` in accordance with the formula

$$n = o - f - 1$$

where n is the array entry number, o is the OID in question, and f is the `firstUserObjectOid` constant.

Each array entry contains the offset into the `objects` file where the object corresponding to this OID can be found. The offset is in little-endian byte-order, as are all other binary data in Magma. Since the offsets are 48-bit numbers, the `objects` file is "limited" to a maximum

Table 2: Metadata in the `objects` file header

	starting offset	size in bytes
signature	0	8
version	8	2
boolean flags	10	1
definition OID	11	8
class definition OID	27	8
anchor object OID	43	8

length of $2^{48} - 1$ bytes[2].

5 The `objects` file format

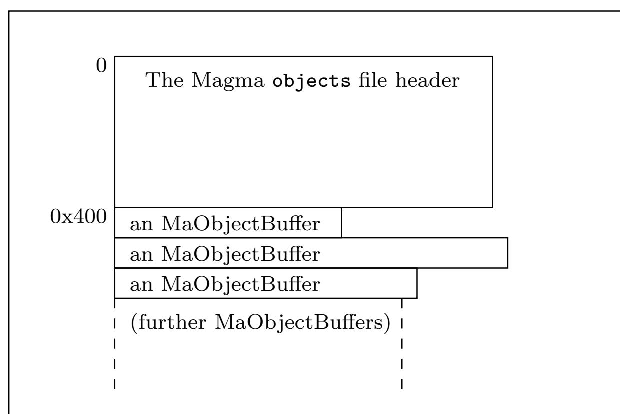

Figure 2: Binary layout of the `objects` file

This file is the only place where objects are actually stored. It consists of a 1024-byte header, and a list of variable-length records. Each record represents one instance. For details, see figure 2.

The header contains a *binary signature*, a *version number*, the *definition OID*, the *class definition OID*, and the *anchor object OID*. For details, refer to table 2. Note that the OIDs are 64-bit numbers in this header; some older versions of Magma used 64-bit OIDs, and the possibility remains of doing so again in the future.

After the header, the objects file is a flat list of variable-length records. Each record represents one object instance. When a record needs to be updated, it can be updated in-place if the new size is identical to the old size. If the new size differs[3], then a new record is simply appended to the file's end, and the offset in the master index is updated to point to the new record. While this scheme does leave stale objects in the file, it does have some benefits: appending to a file is a quick operation, and if the system should fail before writing the offset, the old offset will still point to the old object. Tools also exist for removing stale records.

[2]Strictly speaking, the last record in the `objects` file must *start* no further than $2^{48} - 1$ bytes from the front of the file. It might extend up to $2^{24} - 7$ bytes beyond that limit.

[3]It's not clear to me what happens if the new object is smaller than the old one. While the new one would fit in the space, there would be unused bytes left, and it might be difficult to discover where the next record started (it would no longer be related to the *length* field).

Each record is one of the five possible types shown in figure 3 (solid lines denote concrete classes). Magma uses each type of record for persisting certain types of objects; for instance, strings are always saved as MaByteObjectBuffer records.

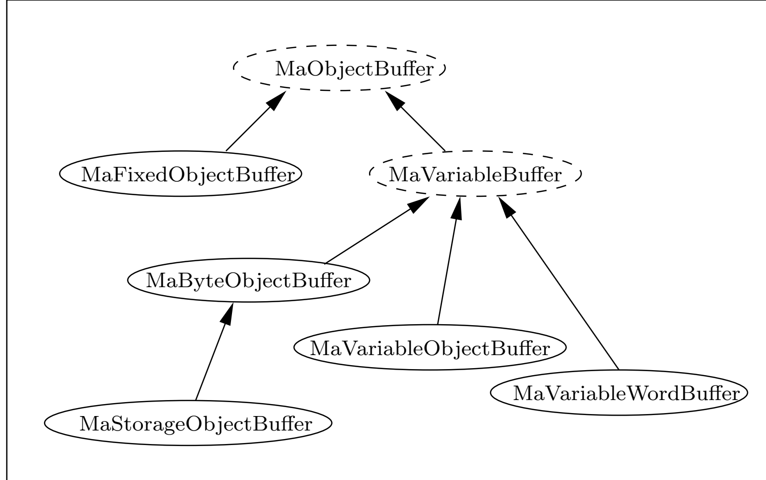

Figure 3: Hierarchy of Magma object record classes

All of these records consist of a size, a control-field, a class ID number, and a class version number. After the header comes either an array of OIDs (each 48 bits wide, as usual), or binary data. For details, see figure 4. Note that all numbers, such as the 24-bit size, are stored in binary, with the MSB first. Since the size field has the OID width (in bytes) added to it, the maximum size of any instance is $2^{24} - 6 = 16777210$ bytes.

The control field maps to the type of object buffer which this record represents. The class ID is used to identify which class to build when re-creating an object from a record. The class version allows for classes to evolve over time; each time a class' structure is changed, the version number is incremented.

In the case of MaByteObjectBuffer records, the header is followed by binary data. In the case of MaFixedObjectBuffer and MaVariableObjectBuffer records, the header is an array of OIDs, which may be "immediate objects" (small integers or 32-bit floats), or may point to other objects. MaStorageObjectBuffer and MaVariableWordBuffer records were not unpacked beyond the header fields during this work, and are likely to differ from this layout.

6 The hash-index (*OID-member*`.hdx`) file format

The other files which are part of Magma's data storage are hash-index files. They allow for large collections of objects to be searched rapidly. The hash-index files do not themselves store objects – the objects are stored only in the main `objects` file. Instead, they store keys (which are a hashed representation of some member variable), and values, which are the OIDs. The OIDs in turn allow for the object to be located. Thus, given a value of a particular member variable, the corresponding OID can be quickly looked up in a hash-index.

Each file is named with the OID of the collection it indexes, and the relevant member-variable. Each indexed collection has an index storing purely the OIDs it contains, to enable rapid confirmation that an OID is in a collection. This file is simply named *OID*`.hdx`. For example, a collection which has the OID 456789, indexed on the "foo" member variable, would thus be stored as two files: `456789.hdx` and `456789foo.hdx`.

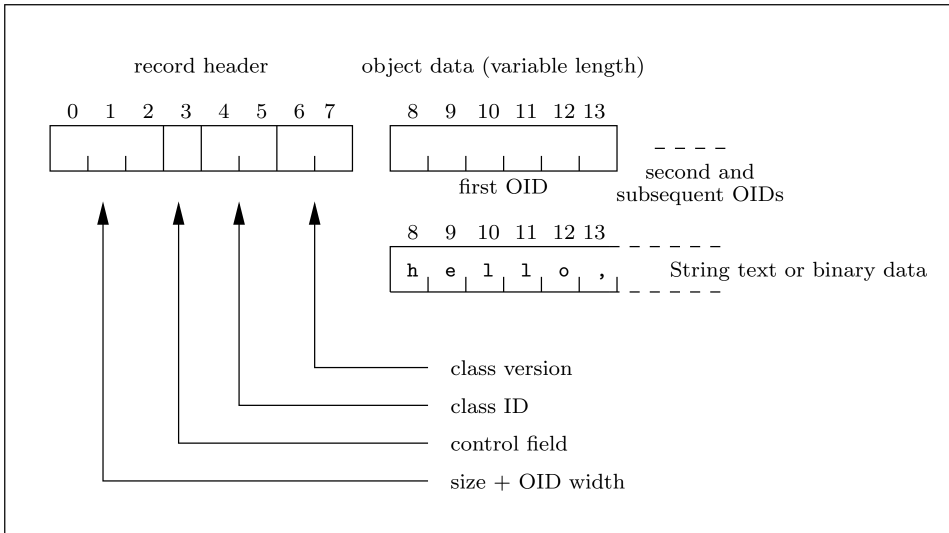

Figure 4: Binary layout of a Magma object record

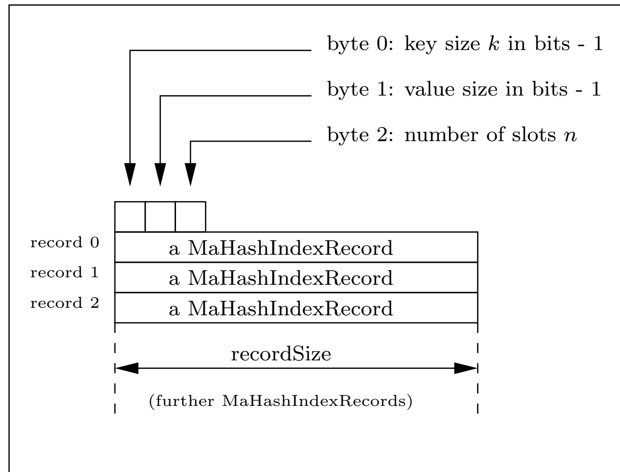

Figure 5: The binary format of Magma's hash-index files

Each hash-index file consists of a three-byte header, and an array of hash-index records, as shown in figure 5. Each record in turn consists of a header and an array of slots, which may be unused or may be hash-index record entries. Each entry contains a single key and value pair. It also contains the number of child entries and a record-number where these may be found. For details, see figure 6.

6.1 The process of adding records to a hash index file

The process of adding records to a hash index file is best illustrated by means of examples. We consider 4 scenarios, as depicted in figure 7.

In scenario 1, we have an empty index, consisting of a single record. The hash of the key can range from 0 to 999 (for simplicity's sake). We add an item where the key hashes to 777. An entry is created containing the hash and the OID. All the other entries which are unused will contain a record number of zero and the OID value "3", both of which reflect an unused hash index slot.

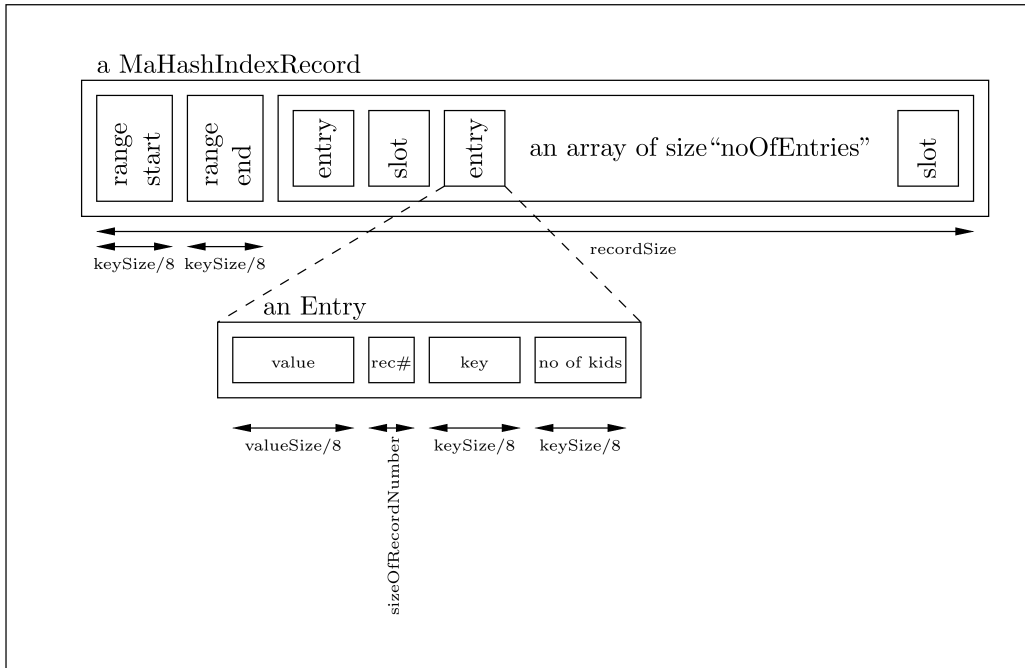

Figure 6: Binary layout of a Magma hash index record

In scenario 2, we start with the index from scenario 1, still consisting of a single record. We add an item where the key hashes to 744; the existing entry is updated to indicate that there are two children, one in this entry, and another one in a subsequent record. A new record is contained to hold entries with keys in the range 700 to 799. A new entry is created to hold the hash of 777 its and OID. Thus the existing entry is "bumped" down to a new record when a new entry with a lower hash takes its place. The "record number" field in the original slot is updated to reflect the record number where further child entries can be found. Record numbers are zero-based (i.e. the first record in the hash-index after the three-byte header is number 0).

In scenario 3, we start with the index from scenario 2, now consisting of two records. We add an item where the key hashes to 721. A new entry is created and added to the second record since it falls within the record's range of 700 to 799. Again, the higher-valued hash entry is bumped down. The first record (i.e. the parent record) gets its *numberOfEntriesInteger* [4] set accordingly.

In scenario 4, we start with the index from scenario 3, consisting of two records. We add an item where the key hashes to 745. A new record is created to span the range from 740 to 749. Two new entries are created to hold keys with the hashes 744 and 745, and are added to the record. This new record's parent (the second one) has its entry for the range 740 to 749 updated to reflect two child-entries. In turn, that record's parent (the first one) has its entry in the range 700 to 799 updated to reflect four child objects.

6.2 Calculations of ranges and slots in the general case

This is, however, not the complete story. Complications arise when a record has a starting and ending range with fewer integer values than it has slots. Magma's code calls this a record with "duplicates", as a single key value can be stored into several possible slots. Consider the third record in scenario number 4: its start key is 740 and its end key is 749, giving a range of 10 integers, which conveniently matches the 10 slots we have available. Since we

[4] This field may have been renamed to *noOfChildren*

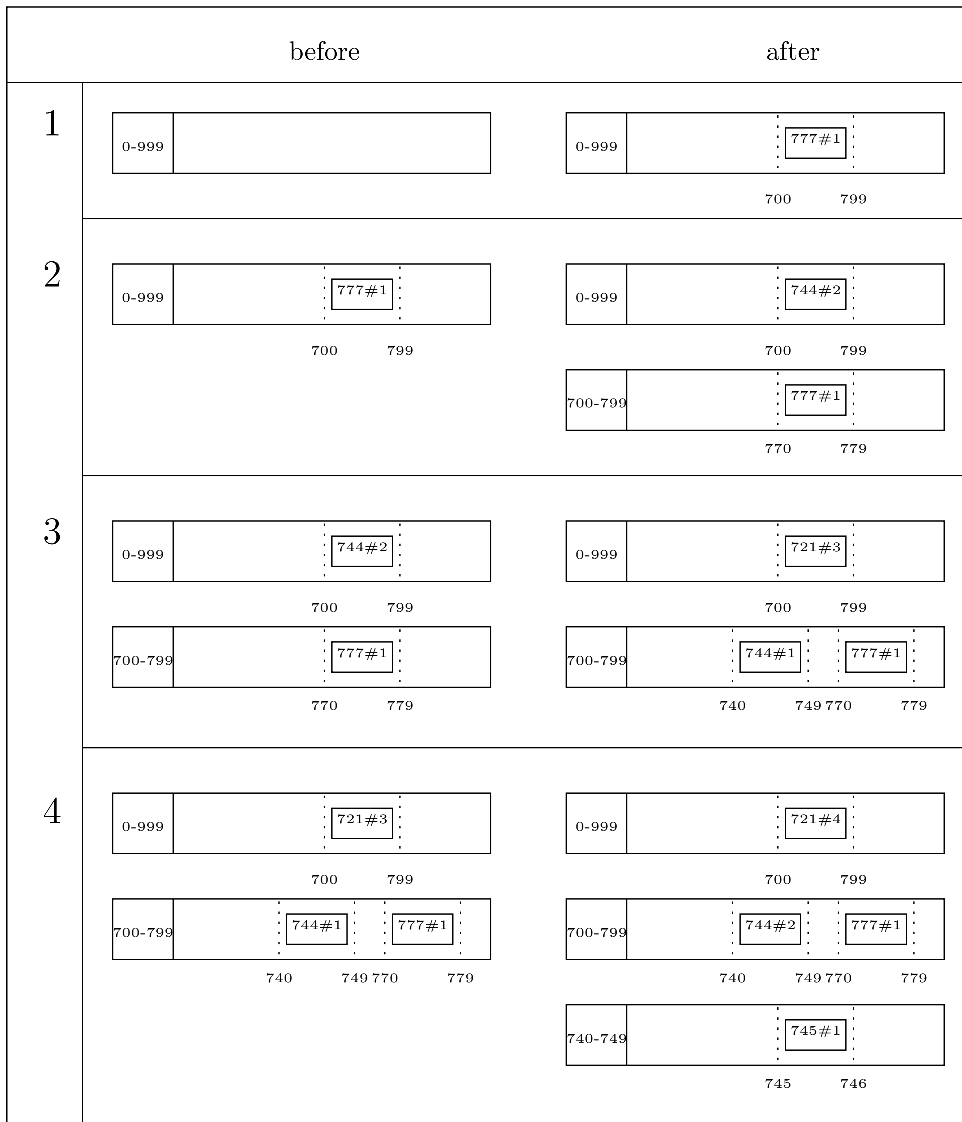

Figure 7: The process of adding records to a hash index file

could continue to add entries with the key 745 (since hash collisions are both possible and permissible), and since we could elect *ab initio* to have more than 10 slots per record, it is clear that further refinement to the process is required.

6.2.1 Adding a new record

Each record's header contains its *range start* and *range end* fields. These values correspond to the lowest and highest possible keys that this record can hold. When adding a record, the new lowest possible key l_n is calculated from the current record and slot number:

$$l_n = l + \frac{s(h+1-l)}{n}$$

	s	is the zero-based slot number,
	n	is the number of slots in a hash-index record,
where	l	is the lowest key in the range,
	h	is the highest key in the range, and
	n	is the number of slots in each record.

The new highest possible key h_n is the same if this record can hold duplicates. Otherwise, it is

$$h_n = l + \frac{(s+1)(h+1-l)}{n} - 1$$

6.2.2 Records with no duplicate slots

If a record has no duplicates for a given key, then the slot number s is given by

$$s = \frac{n(k-l)}{h+1-l}$$

where k is the given key.

In addition, an adjustment to the slot number is done: if the key

$$k > l + \frac{(s+1)(h+1-l)}{n}$$

then

$$s \leftarrow s + 1$$

6.2.3 Records with duplicate slots

If a record has several slots with the same key, then the highest possible slot is used when adding new entries. If it is full, then the next-highest slot is used, and so on down until the lowest possible slot is attempted. If all the permissible slots in a record are filled, then a new record is appended, and the highest slot's *record number* field is updated, and its *numberOfChildren* field is incremented.

The lowest possible slot number s_l is calculated as for s above in 6.2.2.

The highest possible slot number s_h is calculated as

$$s_h = \frac{n(k+1-l)}{h+1-l} - 1$$

Again, the adjustment is done: if

$$k > l + \frac{s_h(h+1-l)}{n}$$

then

$$s_h \leftarrow s_h + 1$$

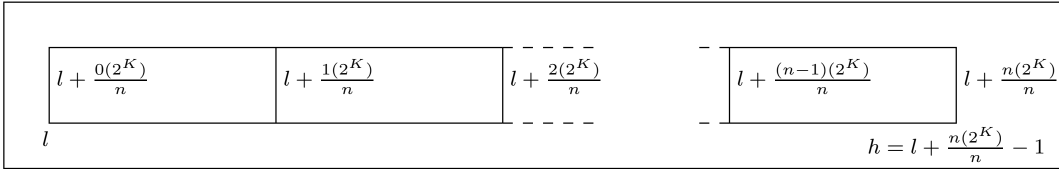

Figure 8: A hash-index record with a lowest key of l, a highest key of h, a key-space of K bits and n slots, showing the lowest-possible keys for each slot in general terms

6.3 Parameter tuning

The first record's lowest (l_0) and highest (h_0) possible keys are a function of the hash index's key size K, which is constrained to be an integral multiple of 8 bits, and hence will always be a power of 2. Thus

$$\begin{aligned} l_0 &= 0 \\ h_0 &= 2^K - 1 \end{aligned}$$

The slots in each record cover this range, as shown in figure 8.

Older versions of Magma used hash-indices containing records with 10 slots each. Each record, therefore, could not divide its keyspace into equally-sized slots. Some slots would be greater than others; that is, they would contain more keys than others. This is because all the quantities (low key, high key, record key, number of slots, slot) involved are by definition integers. This “unevenness” necessitated the adjustments described in 6.2.2 and 6.2.3.

Therefore, if every record r were to contain n slots where

$$\boxed{\log_2 n \in \mathcal{Z}}$$

then the range $(l_r..h_r)$ would always be evenly divisible by n. For illustration, consider the example shown in figure 9. Newer versions of Magma enforce this constraint to ensure that each record contains slots of uniform size; this provides a small performance improvement.

7 Extracting all objects from a database

Each of the (non-stale) objects in the database may be found in one of two ways:

1. Objects which are stored in a hash-index may be obtained by examining each slot in each record. Slots which are not empty will contain OIDs. This operation is an array-walk.
2. The anchor object by definition holds references, in the form of OIDs, to all other “reachable” objects which which are not also in hash-indices. The anchor object's OID is obtainable from the `objects` file header. This operation is a recursive tree-walk.

Given an OID, the offset into the `objects` file may be found by a simple array dereference operation on the `objects.idx` file. Once the offset is known, the data in the `objects` file may be read, starting with the *length* field, which indicates how far to read. After that, the

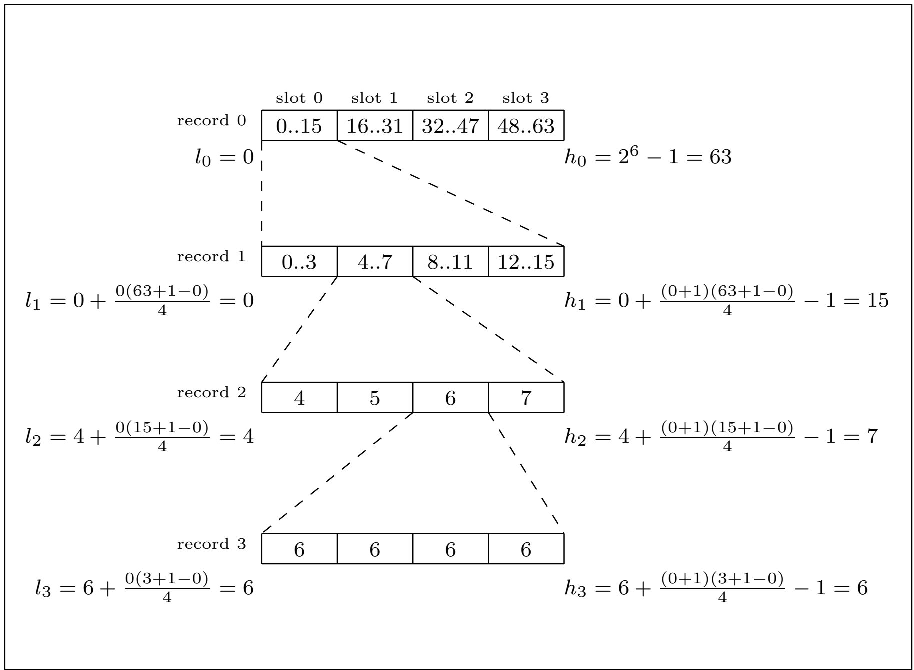

Figure 9: The first four records of a hash-index where the keysize $K = 6$ and the number of slots $n = 4$

class ID and *class version* fields may be read since their offsets are known *a priori*. This provides enough information to convert the remaining bytes in the MaObjectBuffer into the corresponding object.

Thus to extract all objects from a Magma database, it is necessary to walk each (OID-only) hash index and then to traverse the object tree starting at the anchor object.