a) What are the corresponding blocks of the redundant disks?

b) If the third disk's block is rewritten to be 10000000, what steps must be taken to change other disks?

Exercise 13.4.10: Describe the steps taken to recover from the following failures using the RAID level 6 scheme with seven disks: (a) disks 1 and 7, (b) disks 1 and 4, (c) disks 3 and 6.

13.5 Arranging Data on Disk

We now turn to the matter of how disks are used store databases. A data element such as a tuple or object is represented by a *record*, which consists of consecutive bytes in some disk block. Collections such as relations are usually represented by placing the records that represent their data elements in one or more blocks. It is normal for a disk block to hold only elements of one relation, although there are organizations where blocks hold tuples of several relations. In this section, we shall cover the basic layout techniques for both records and blocks.

13.5.1 Fixed-Length Records

The simplest sort of record consists of fixed-length *fields*, one for each attribute of the represented tuple. Many machines allow more efficient reading and writing of main memory when data begins at an address that is a multiple of 4 or 8; some even require us to do so. Thus, it is common to begin all fields at a multiple of 4 or 8, as appropriate. Space not used by the previous field is wasted. Note that, even though records are kept in secondary, not main, memory, they are manipulated in main memory. Thus it is necessary to lay out the record so it can be moved to main memory and accessed efficiently there.

Often, the record begins with a *header*, a fixed-length region where information about the record itself is kept. For example, we may want to keep in the record:

1. A pointer to the schema for the data stored in the record. For example, a tuple's record could point to the schema for the relation to which the tuple belongs. This information helps us find the fields of the record.

2. The length of the record. This information helps us skip over records without consulting the schema.

3. Timestamps indicating the time the record was last modified, or last read. This information may be useful for implementing database transactions as will be discussed in Chapter 18.

4. Pointers to the fields of the record. This information can substitute for schema information, and it will be seen to be important when we consider variable-length fields in Section 13.7.

```
CREATE TABLE MovieStar(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

Figure 13.15: A SQL table declaration

Example 13.15: Figure 13.15 repeats our running `MovieStar` schema. Let us assume all fields must start at a byte that is a multiple of four. Tuples of this relation have a header and the following four fields:

1. The first field is for `name`, and this field requires 30 bytes. If we assume that all fields begin at a multiple of 4, then we allocate 32 bytes for the `name`.

2. The next attribute is `address`. A `VARCHAR` attribute requires a fixed-length segment of bytes, with one more byte than the maximum length (for the string's endmarker). Thus, we need 256 bytes for `address`.

3. Attribute `gender` is a single byte, holding either the character `'M'` or `'F'`. We allocate 4 bytes, so the next field can start at a multiple of 4.

4. Attribute `birthdate` is a SQL `DATE` value, which is a 10-byte string. We shall allocate 12 bytes to its field, to keep subsequent records in the block aligned at multiples of 4.

. The header of the record will hold:

a) A pointer to the record schema.

b) The record length.

c) A timestamp indicating when the record was created.

We shall assume each of these items is 4 bytes long. Figure 13.16 shows the layout of a record for a `MovieStar` tuple. The length of the record is 316 bytes. □

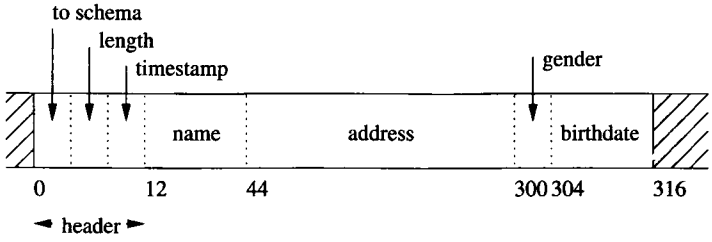

Figure 13.16: Layout of records for tuples of the `MovieStar` relation

13.5.2 Packing Fixed-Length Records into Blocks

Records representing tuples of a relation are stored in blocks of the disk and moved into main memory (along with their entire block) when we need to access or update them. The layout of a block that holds records is suggested in Fig. 13.17.

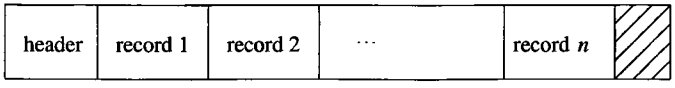

Figure 13.17: A typical block holding records

In addition to the records, there is a *block header* holding information such as:

1. Links to one or more other blocks that are part of a network of blocks such as those that will be described in Chapter 14 for creating indexes to the tuples of a relation.
2. Information about the role played by this block in such a network.
3. Information about which relation the tuples of this block belong to.
4. A "directory" giving the offset of each record in the block.
5. Timestamp(s) indicating the time of the block's last modification and/or access.

By far the simplest case is when the block holds tuples from one relation, and the records for those tuples have a fixed format. In that case, following the header, we pack as many records as we can into the block and leave the remaining space unused.

Example 13.16: Suppose we are storing records with the layout developed in Example 13.15. These records are 316 bytes long. Suppose also that we use 4096-byte blocks. Of these bytes, say 12 will be used for a block header, leaving 4084 bytes for data. In this space we can fit twelve records of the given 316-byte format, and 292 bytes of each block are wasted space. □

13.5.3 Exercises for Section 13.5

Exercise 13.5.1: Suppose a record has the following fields in this order: A character string of length 15, an integer of 2 bytes, a SQL date, and a SQL time (no decimal point). How many bytes does the record take if:

a) Fields can start at any byte.

b) Fields must start at a byte that is a multiple of 4.

c) Fields must start at a byte that is a multiple of 8.

Exercise 13.5.2: Repeat Exercise 13.5.1 for the list of fields: a real of 8 bytes, a character string of length 17, a single byte, and a SQL date.

Exercise 13.5.3: Assume fields are as in Exercise 13.5.1, but records also have a record header consisting of two 4-byte pointers and a character. Calculate the record length for the three situations regarding field alignment (a) through (c) in Exercise 13.5.1.

Exercise 13.5.4: Repeat Exercise 13.5.2 if the records also include a header consisting of an 8-byte pointer, and ten 2-byte integers.

13.6 Representing Block and Record Addresses

When in main memory, the address of a block is the virtual-memory address of its first byte, and the address of a record within that block is the virtual-memory address of the first byte of that record. However, in secondary storage, the block is not part of the application's virtual-memory address space. Rather, a sequence of bytes describes the location of the block within the overall system of data accessible to the DBMS: the device ID for the disk, the cylinder number, and so on. A record can be identified by giving its block address and the offset of the first byte of the record within the block.

In this section, we shall begin with a discussion of address spaces, especially as they pertain to the common "client-server" architecture for DBMS's (see Section 9.2.4). We then discuss the options for representing addresses, and finally look at "pointer swizzling," the ways in which we can convert addresses in the data server's world to the world of the client application programs.

13.6.1 Addresses in Client-Server Systems

Commonly, a database system consists of a *server* process that provides data from secondary storage to one or more *client* processes that are applications using the data. The server and client processes may be on one machine, or the server and the various clients can be distributed over many machines.

The client application uses a conventional "virtual" address space, typically 32 bits, or about 4 billion different addresses. The operating system or DBMS

decides which parts of the address space are currently located in main memory, and hardware maps the virtual address space to physical locations in main memory. We shall not think further of this virtual-to-physical translation, and shall think of the client address space as if it were main memory itself.

The server's data lives in a *database address space*. The addresses of this space refer to blocks, and possibly to offsets within the block. There are several ways that addresses in this address space can be represented:

1. *Physical Addresses.* These are byte strings that let us determine the place within the secondary storage system where the block or record can be found. One or more bytes of the physical address are used to indicate each of:

   (a) The host to which the storage is attached (if the database is stored across more than one machine),

   (b) An identifier for the disk or other device on which the block is located,

   (c) The number of the cylinder of the disk,

   (d) The number of the track within the cylinder,

   (e) The number of the block within the track, and

   (f) (In some cases) the offset of the beginning of the record within the block.

2. *Logical Addresses.* Each block or record has a "logical address," which is an arbitrary string of bytes of some fixed length. A *map table*, stored on disk in a known location, relates logical to physical addresses, as suggested in Fig. 13.18.

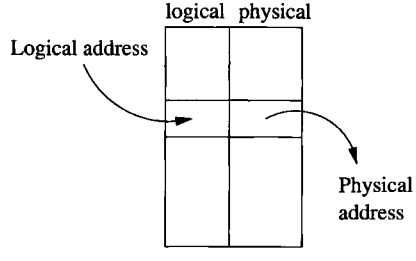

Figure 13.18: A map table translates logical to physical addresses

Notice that physical addresses are long. Eight bytes is about the minimum we could use if we incorporate all the listed elements, and some systems use many more bytes. For example, imagine a database of objects that is designed to last for 100 years. In the future, the database may grow to encompass one

million machines, and each machine might be fast enough to create one object every nanosecond. This system would create around 2^{77} objects, which requires a minimum of ten bytes to represent addresses. Since we would probably prefer to reserve some bytes to represent the host, others to represent the storage unit, and so on, a rational address notation would use considerably more than 10 bytes for a system of this scale.

13.6.2 Logical and Structured Addresses

One might wonder what the purpose of logical addresses could be. All the information needed for a physical address is found in the map table, and following logical pointers to records requires consulting the map table and then going to the physical address. However, the level of indirection involved in the map table allows us considerable flexibility. For example, many data organizations require us to move records around, either within a block or from block to block. If we use a map table, then all pointers to the record refer to this map table, and all we have to do when we move or delete the record is to change the entry for that record in the table.

Many combinations of logical and physical addresses are possible as well, yielding *structured* address schemes. For instance, one could use a physical address for the block (but not the offset within the block), and add the key value for the record being referred to. Then, to find a record given this structured address, we use the physical part to reach the block containing that record, and we examine the records of the block to find the one with the proper key.

A similar, and very useful, combination of physical and logical addresses is to keep in each block an *offset table* that holds the offsets of the records within the block, as suggested in Fig. 13.19. Notice that the table grows from the front end of the block, while the records are placed starting at the end of the block. This strategy is useful when the records need not be of equal length. Then, we do not know in advance how many records the block will hold, and we do not have to allocate a fixed amount of the block header to the table initially.

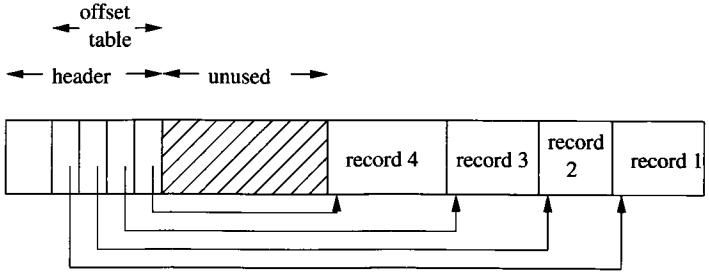

Figure 13.19: A block with a table of offsets telling us the position of each record within the block

The address of a record is now the physical address of its block plus the offset

of the entry in the block's offset table for that record. This level of indirection within the block offers many of the advantages of logical addresses, without the need for a global map table.

- We can move the record around within the block, and all we have to do is change the record's entry in the offset table; pointers to the record will still be able to find it.

- We can even allow the record to move to another block, if the offset table entries are large enough to hold a *forwarding address* for the record, giving its new location.

- Finally, we have an option, should the record be deleted, of leaving in its offset-table entry a *tombstone*, a special value that indicates the record has been deleted. Prior to its deletion, pointers to this record may have been stored at various places in the database. After record deletion, following a pointer to this record leads to the tombstone, whereupon the pointer can either be replaced by a null pointer, or the data structure otherwise modified to reflect the deletion of the record. Had we not left the tombstone, the pointer might lead to some new record, with surprising, and erroneous, results.

13.6.3 Pointer Swizzling

Often, pointers or addresses are part of records. This situation is not typical for records that represent tuples of a relation, but it is common for tuples that represent objects. Also, modern object-relational database systems allow attributes of pointer type (called references), so even relational systems need the ability to represent pointers in tuples. Finally, index structures are composed of blocks that usually have pointers within them. Thus, we need to study the management of pointers as blocks are moved between main and secondary memory.

As we mentioned earlier, every block, record, object, or other referenceable data item has two forms of address: its *database address* in the server's address space, and a *memory address* if the item is currently copied in virtual memory. When in secondary storage, we surely must use the database address of the item. However, when the item is in the main memory, we can refer to the item by either its database address or its memory address. It is more efficient to put memory addresses wherever an item has a pointer, because these pointers can be followed using a single machine instruction.

In contrast, following a database address is much more time-consuming. We need a table that translates from all those database addresses that are currently in virtual memory to their current memory address. Such a *translation table* is suggested in Fig. 13.20. It may look like the map table of Fig. 13.18 that translates between logical and physical addresses. However:

a) Logical and physical addresses are both representations for the database address. In contrast, memory addresses in the translation table are for copies of the corresponding object in memory.

b) All addressable items in the database have entries in the map table, while only those items currently in memory are mentioned in the translation table.

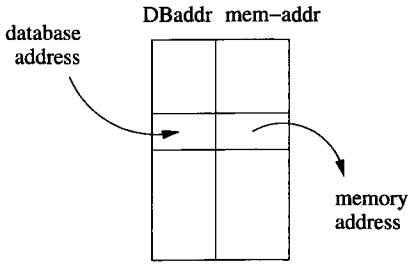

Figure 13.20: The translation table turns database addresses into their equivalents in memory

To avoid the cost of translating repeatedly from database addresses to memory addresses, several techniques have been developed that are collectively known as *pointer swizzling*. The general idea is that when we move a block from secondary to main memory, pointers within the block may be "swizzled," that is, translated from the database address space to the virtual address space. Thus, a pointer actually consists of:

1. A bit indicating whether the pointer is currently a database address or a (swizzled) memory address.

2. The database or memory pointer, as appropriate. The same space is used for whichever address form is present at the moment. Of course, not all the space may be used when the memory address is present, because it is typically shorter than the database address.

Example 13.17: Figure 13.21 shows a simple situation in which the Block 1 has a record with pointers to a second record on the same block and to a record on another block. The figure also shows what might happen when Block 1 is copied to memory. The first pointer, which points within Block 1, can be swizzled so it points directly to the memory address of the target record.

However, if Block 2 is not in memory at this time, then we cannot swizzle the second pointer; it must remain unswizzled, pointing to the database address of its target. Should Block 2 be brought to memory later, it becomes theoretically possible to swizzle the second pointer of Block 1. Depending on the swizzling strategy used, there may or may not be a list of such pointers that are in

memory, referring to Block 2; if so, then we have the option of swizzling the pointer at that time. □

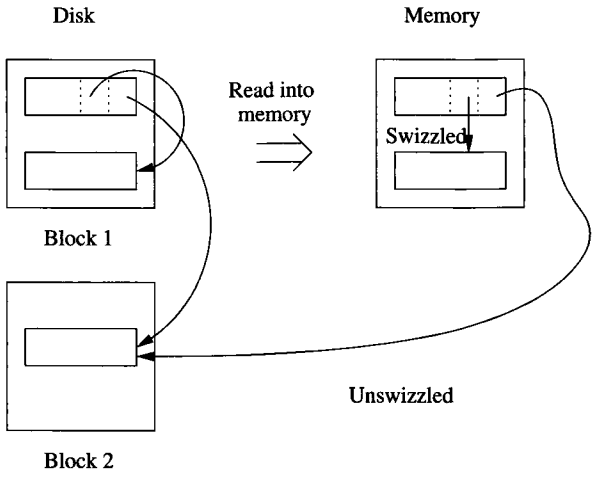

Figure 13.21: Structure of a pointer when swizzling is used

Automatic Swizzling

There are several strategies we can use to determine when to swizzle pointers. If we use *automatic swizzling*, then as soon as a block is brought into memory, we locate all its pointers and addresses and enter them into the translation table if they are not already there. These pointers include both the pointers *from* records in the block to elsewhere and the addresses of the block itself and/or its records, if these are addressable items. We need some mechanism to locate the pointers within the block. For example:

1. If the block holds records with a known schema, the schema will tell us where in the records the pointers are found.

2. If the block is used for one of the index structures we shall discuss in Chapter 14, then the block will hold pointers at known locations.

3. We may keep within the block header a list of where the pointers are.

When we enter into the translation table the addresses for the block just moved into memory, and/or its records, we know where in memory the block has been buffered. We may thus create the translation-table entry for these database addresses straightforwardly. When we insert one of these database addresses A into the translation table, we may find it in the table already, because its block is currently in memory. In this case, we replace A in the block

just moved to memory by the corresponding memory address, and we set the "swizzled" bit to true. On the other hand, if A is not yet in the translation table, then its block has not been copied into main memory. We therefore cannot swizzle this pointer and leave it in the block as a database pointer.

Suppose that during the use of this data, we follow a pointer P and we find that P is still unswizzled, i.e., in the form of a database pointer. We consult the translation table to see if database address P currently has a memory equivalent. If not, block B must be copied into a memory buffer. Once B is in memory, we can "swizzle" P by replacing its database form by the equivalent memory form.

Swizzling on Demand

Another approach is to leave all pointers unswizzled when the block is first brought into memory. We enter its address, and the addresses of its pointers, into the translation table, along with their memory equivalents. If we follow a pointer P that is inside some block of memory, we swizzle it, using the same strategy that we followed when we found an unswizzled pointer using automatic swizzling.

The difference between on-demand and automatic swizzling is that the latter tries to get all the pointers swizzled quickly and efficiently when the block is loaded into memory. The possible time saved by swizzling all of a block's pointers at one time must be weighed against the possibility that some swizzled pointers will never be followed. In that case, any time spent swizzling and unswizzling the pointer will be wasted.

An interesting option is to arrange that database pointers look like invalid memory addresses. If so, then we can allow the computer to follow any pointer as if it were in its memory form. If the pointer happens to be unswizzled, then the memory reference will cause a hardware trap. If the DBMS provides a function that is invoked by the trap, and this function "swizzles" the pointer in the manner described above, then we can follow swizzled pointers in single instructions, and only need to do something more time consuming when the pointer is unswizzled.

No Swizzling

Of course it is possible never to swizzle pointers. We still need the translation table, so the pointers may be followed in their unswizzled form. This approach does offer the advantage that records cannot be pinned in memory, as discussed in Section 13.6.5, and decisions about which form of pointer is present need not be made.

Programmer Control of Swizzling

In some applications, it may be known by the application programmer whether the pointers in a block are likely to be followed. This programmer may be able

to specify explicitly that a block loaded into memory is to have its pointers swizzled, or the programmer may call for the pointers to be swizzled only as needed. For example, if a programmer knows that a block is likely to be accessed heavily, such as the root block of a B-tree (discussed in Section 14.2), then the pointers would be swizzled. However, blocks that are loaded into memory, used once, and then likely dropped from memory, would not be swizzled.

13.6.4 Returning Blocks to Disk

When a block is moved from memory back to disk, any pointers within that block must be "unswizzled"; that is, their memory addresses must be replaced by the corresponding database addresses. The translation table can be used to associate addresses of the two types in either direction, so in principle it is possible to find, given a memory address, the database address to which the memory address is assigned.

However, we do not want each unswizzling operation to require a search of the entire translation table. While we have not discussed the implementation of this table, we might imagine that the table of Fig. 13.20 has appropriate indexes. If we think of the translation table as a relation, then the problem of finding the memory address associated with a database address x can be expressed as the query:

```
SELECT memAddr
FROM TranslationTable
WHERE dbAddr = x;
```

For instance, a hash table using the database address as the key might be appropriate for an index on the `dbAddr` attribute; Chapter 14 suggests possible data structures.

If we want to support the reverse query,

```
SELECT dbAddr
FROM TranslationTable
WHERE memAddr = y;
```

then we need to have an index on attribute `memAddr` as well. Again, Chapter 14 suggests data structures suitable for such an index. Also, Section 13.6.5 talks about linked-list structures that in some circumstances can be used to go from a memory address to all main-memory pointers to that address.

13.6.5 Pinned Records and Blocks

A block in memory is said to be *pinned* if it cannot at the moment be written back to disk safely. A bit telling whether or not a block is pinned can be located in the header of the block. There are many reasons why a block could be pinned, including requirements of a recovery system as discussed in Chapter 17. Pointer swizzling introduces an important reason why certain blocks must be pinned.

If a block B_1 has within it a swizzled pointer to some data item in block B_2, then we must be very careful about moving block B_2 back to disk and reusing its main-memory buffer. The reason is that, should we follow the pointer in B_1, it will lead us to the buffer, which no longer holds B_2; in effect, the pointer has become dangling. A block, like B_2, that is referred to by a swizzled pointer from somewhere else is therefore pinned.

When we write a block back to disk, we not only need to "unswizzle" any pointers in that block. We also need to make sure it is not pinned. If it is pinned, we must either unpin it, or let the block remain in memory, occupying space that could otherwise be used for some other block. To unpin a block that is pinned because of swizzled pointers from outside, we must "unswizzle" any pointers to it. Consequently, the translation table must record, for each database address whose data item is in memory, the places in memory where swizzled pointers to that item exist. Two possible approaches are:

1. Keep the list of references to a memory address as a linked list attached to the entry for that address in the translation table.

2. If memory addresses are significantly shorter than database addresses, we can create the linked list in the space used for the pointers themselves. That is, each space used for a database pointer is replaced by

   (a) The swizzled pointer, and

   (b) Another pointer that forms part of a linked list of all occurrences of this pointer.

   Figure 13.22 suggests how two occurrences of a memory pointer y could be linked, starting at the entry in the translation table for database address x and its corresponding memory address y.

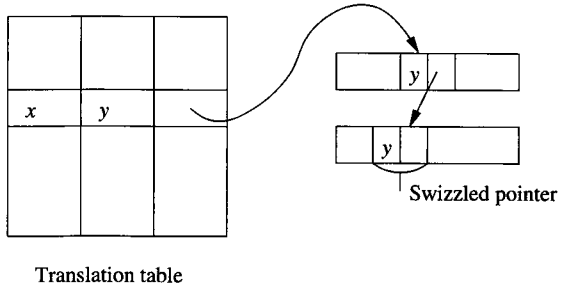

Figure 13.22: A linked list of occurrences of a swizzled pointer

13.6.6 Exercises for Section 13.6

Exercise 13.6.1: If we represent physical addresses for the Megatron 747 disk by allocating a separate byte or bytes to each of the cylinder, track within a cylinder, and block within a track, how many bytes do we need? Make a reasonable assumption about the maximum number of blocks on each track; recall that the Megatron 747 has a variable number of sectors/track.

Exercise 13.6.2: Repeat Exercise 13.6.1 for the Megatron 777 disk described in Exercise 13.2.1

Exercise 13.6.3: If we wish to represent record addresses as well as block addresses, we need additional bytes. Assuming we want addresses for a single Megatron 747 disk as in Exercise 13.6.1, how many bytes would we need for record addresses if we:

a) Included the number of the byte within a block as part of the physical address.

b) Used structured addresses for records. Assume that the stored records have a 4-byte integer as a key.

Exercise 13.6.4: Today, IP addresses have four bytes. Suppose that block addresses for a world-wide address system consist of an IP address for the host, a device number between 1 and 1000, and a block address on an individual device (assumed to be a Megatron 747 disk). How many bytes would block addresses require?

Exercise 13.6.5: In IP version 6, IP addresses are 16 bytes long. In addition, we may want to address not only blocks, but records, which may start at any byte of a block. However, devices will have their own IP address, so there will be no need to represent a device within a host, as we suggested was necessary in Exercise 13.6.4. How many bytes would be needed to represent addresses in these circumstances, again assuming devices were Megatron 747 disks?

! **Exercise 13.6.6:** Suppose we wish to represent the addresses of blocks on a Megatron 747 disk logically, i.e., using identifiers of k bytes for some k. We also need to store on the disk itself a map table, as in Fig. 13.18, consisting of pairs of logical and physical addresses. The blocks used for the map table itself are not part of the database, and therefore do not have their own logical addresses in the map table. Assuming that physical addresses use the minimum possible number of bytes for physical addresses (as calculated in Exercise 13.6.1), and logical addresses likewise use the minimum possible number of bytes for logical addresses, how many blocks of 4096 bytes does the map table for the disk occupy?

! **Exercise 13.6.7:** Suppose that we have 4096-byte blocks in which we store records of 100 bytes. The block header consists of an offset table, as in Fig. 13.19, using 2-byte pointers to records within the block. On an average day, two records per block are inserted, and one record is deleted. A deleted record must have its pointer replaced by a "tombstone," because there may be dangling pointers to it. For specificity, assume the deletion on any day always occurs before the insertions. If the block is initially empty, after how many days will there be no room to insert any more records?

Exercise 13.6.8: Suppose that if we swizzle all pointers automatically, we can perform the swizzling in half the time it would take to swizzle each one separately. If the probability that a pointer in main memory will be followed at least once is p, for what values of p is it more efficient to swizzle automatically than on demand?

! **Exercise 13.6.9:** Generalize Exercise 13.6.8 to include the possibility that we never swizzle pointers. Suppose that the important actions take the following times, in some arbitrary time units:

i. On-demand swizzling of a pointer: 30.

ii. Automatic swizzling of pointers: 20 per pointer.

iii. Following a swizzled pointer: 1.

iv. Following an unswizzled pointer: 10.

Suppose that in-memory pointers are either not followed (probability $1 - p$) or are followed k times (probability p). For what values of k and p do no-swizzling, automatic-swizzling, and on-demand-swizzling each offer the best average performance?

13.7 Variable-Length Data and Records

Until now, we have made the simplifying assumptions that records have a fixed schema, and that the schema is a list of fixed-length fields. However, in practice, we also may wish to represent:

1. *Data items whose size varies.* For instance, in Fig. 13.15 we considered a `MovieStar` relation that had an address field of up to 255 bytes. While there might be some addresses that long, the vast majority of them will probably be 50 bytes or less. We could save more than half the space used for storing `MovieStar` tuples if we used only as much space as the actual address needed.

2. *Repeating fields.* If we try to represent a many-many relationship in a record representing an object, we shall have to store references to as many objects as are related to the given object.

3. *Variable-format records.* Sometimes we do not know in advance what the fields of a record will be, or how many occurrences of each field there will be. An important example is a record that represents an XML element, which might have no constraints at all, or might be allowed to have repeating subelements, optional attributes, and so on.

4. *Enormous fields.* Modern DBMS's support attributes whose values are very large. For instance, a movie record might have a field that is a 2-gigabyte MPEG encoding of the movie itself, as well as more mundane fields such as the title of the movie.

13.7.1 Records With Variable-Length Fields

If one or more fields of a record have variable length, then the record must contain enough information to let us find any field of the record. A simple but effective scheme is to put all fixed-length fields ahead of the variable-length fields. We then place in the record header:

1. The length of the record.

2. Pointers to (i.e., offsets of) the beginnings of all the variable-length fields other than the first (which we know must immediately follow the fixed-length fields).

Example 13.18: Suppose we have movie-star records with name, address, gender, and birthdate. We shall assume that the gender and birthdate are fixed-length fields, taking 4 and 12 bytes, respectively. However, both name and address will be represented by character strings of whatever length is appropriate. Figure 13.23 suggests what a typical movie-star record would look like. Note that no pointer to the beginning of the name is needed; that field begins right after the fixed-length portion of the record. □

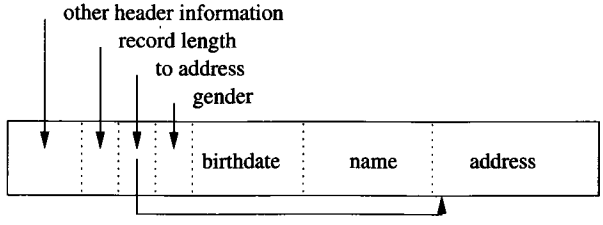

Figure 13.23: A `MovieStar` record with `name` and `address` implemented as variable-length character strings

Representing Null Values

Tuples often have fields that may be NULL. The record format of Fig. 13.23 offers a convenient way to represent NULL values. If a field such as address is null, then we put a null pointer in the place where the pointer to an address goes. Then, we need no space for an address, except the place for the pointer. This arrangement can save space on average, even if address is a fixed-length field but frequently has the value NULL.

13.7.2 Records With Repeating Fields

A similar situation occurs if a record contains a variable number of occurrences of a field F, but the field itself is of fixed length. It is sufficient to group all occurrences of field F together and put in the record header a pointer to the first. We can locate all the occurrences of the field F as follows. Let the number of bytes devoted to one instance of field F be L. We then add to the offset for the field F all integer multiples of L, starting at 0, then L, $2L$, $3L$, and so on. Eventually, we reach the offset of the field following F or the end of the record, whereupon we stop.

Example 13.19: Suppose we redesign our movie-star records to hold only the name and address (which are variable-length strings) and pointers to all the movies of the star. Figure 13.24 shows how this type of record could be represented. The header contains pointers to the beginning of the address field (we assume the name field always begins right after the header) and to the first of the movie pointers. The length of the record tells us how many movie pointers there are. □

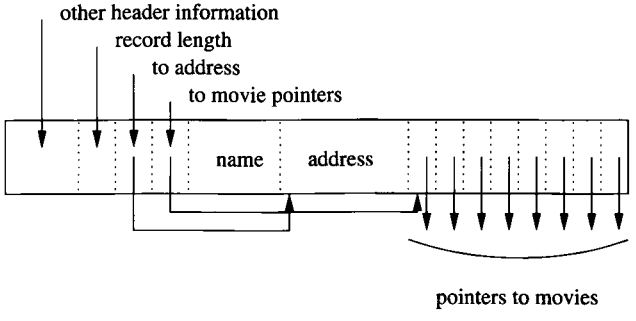

Figure 13.24: A record with a repeating group of references to movies

An alternative representation is to keep the record of fixed length, and put the variable-length portion — be it fields of variable length or fields that repeat

an indefinite number of times — on a separate block. In the record itself we keep:

1. Pointers to the place where each repeating field begins, and
2. Either how many repetitions there are, or where the repetitions end.

Figure 13.25 shows the layout of a record for the problem of Example 13.19, but with the variable-length fields `name` and `address`, and the repeating field `starredIn` (a set of movie references) kept on a separate block or blocks.

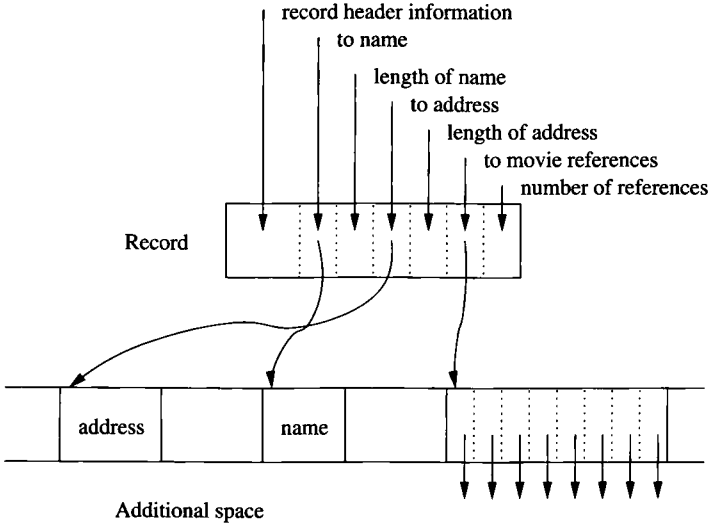

Figure 13.25: Storing variable-length fields separately from the record

There are advantages and disadvantages to using indirection for the variable-length components of a record:

- Keeping the record itself fixed-length allows records to be searched more efficiently, minimizes the overhead in block headers, and allows records to be moved within or among blocks with minimum effort.
- On the other hand, storing variable-length components on another block increases the number of disk I/O's needed to examine all components of a record.

A compromise strategy is to keep in the fixed-length portion of the record enough space for:

1. Some reasonable number of occurrences of the repeating fields,

2. A pointer to a place where additional occurrences could be found, and

3. A count of how many additional occurrences there are.

If there are fewer than this number, some of the space would be unused. If there are more than can fit in the fixed-length portion, then the pointer to additional space will be nonnull, and we can find the additional occurrences by following this pointer.

13.7.3 Variable-Format Records

An even more complex situation occurs when records do not have a fixed schema. We mentioned an example: records that represent XML elements. For another example, medical records may contain information about many tests, but there are thousands of possible tests, and each patient has results for relatively few of them. If the outcome of each test is an attribute, we would prefer that the record for each tuple hold only the attributes for which the outcome is nonnull.

The simplest representation of variable-format records is a sequence of *tagged fields*, each of which consists of the value of the field preceded by information about the role of this field, such as:

1. The attribute or field name,

2. The type of the field, if it is not apparent from the field name and some readily available schema information, and

3. The length of the field, if it is not apparent from the type.

Example 13.20: Suppose movie stars may have additional attributes such as movies directed, former spouses, restaurants owned, and a number of other known but unusual pieces of information. In Fig. 13.26 we see the beginning of a hypothetical movie-star record using tagged fields. We suppose that single-byte codes are used for the various possible field names and types. Appropriate codes are indicated on the figure, along with lengths for the two fields shown, both of which happen to be of type string. □

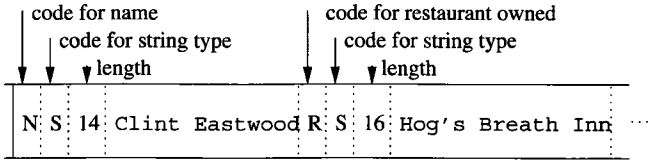

Figure 13.26: A record with tagged fields

13.7.4 Records That Do Not Fit in a Block

Today, DBMS's frequently are used to manage datatypes with large values; often values do not fit in one block. Typical examples are video or audio "clips." Often, these large values have a variable length, but even if the length is fixed for all values of the type, we need special techniques to represent values that are larger than blocks. In this section we shall consider a technique called "spanned records." The management of extremely large values (megabytes or gigabytes) is addressed in Section 13.7.5.

Spanned records also are useful in situations where records are smaller than blocks, but packing whole records into blocks wastes significant amounts of space. For instance, the wasted space in Example 13.16 was only 7%, but if records are just slightly larger than half a block, the wasted space can approach 50%. The reason is that then we can pack only one record per block.

The portion of a record that appears in one block is called a *record fragment.* A record with two or more fragments is called *spanned*, and records that do not cross a block boundary are *unspanned.*

If records can be spanned, then every record and record fragment requires some extra header information:

1. Each record or fragment header must contain a bit telling whether or not it is a fragment.

2. If it is a fragment, then it needs bits telling whether it is the first or last fragment for its record.

3. If there is a next and/or previous fragment for the same record, then the fragment needs pointers to these other fragments.

Example 13.21: Figure 13.27 suggests how records that were about 60% of a block in size could be stored with three records for every two blocks. The header for record fragment $2a$ contains an indicator that it is a fragment, an indicator that it is the first fragment for its record, and a pointer to next fragment, $2b$. Similarly, the header for $2b$ indicates it is the last fragment for its record and holds a back-pointer to the previous fragment $2a$. □

13.7.5 BLOBs

Now, let us consider the representation of truly large values for records or fields of records. The common examples include images in various formats (e.g., GIF, or JPEG), movies in formats such as MPEG, or signals of all sorts: audio, radar, and so on. Such values are often called *binary, large objects*, or BLOBs. When a field has a BLOB as value, we must rethink at least two issues.

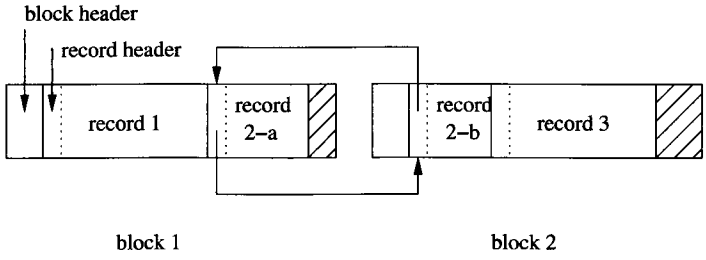

Figure 13.27: Storing spanned records across blocks

Storage of BLOBs

A BLOB must be stored on a sequence of blocks. Often we prefer that these blocks are allocated consecutively on a cylinder or cylinders of the disk, so the BLOB may be retrieved efficiently. However, it is also possible to store the BLOB on a linked list of blocks.

Moreover, it is possible that the BLOB needs to be retrieved so quickly (e.g., a movie that must be played in real time), that storing it on one disk does not allow us to retrieve it fast enough. Then, it is necessary to *stripe* the BLOB across several disks, that is, to alternate blocks of the BLOB among these disks. Thus, several blocks of the BLOB can be retrieved simultaneously, increasing the retrieval rate by a factor approximately equal to the number of disks involved in the striping.

Retrieval of BLOBs

Our assumption that when a client wants a record, the block containing the record is passed from the database server to the client in its entirety may not hold. We may want to pass only the "small" fields of the record, and allow the client to request blocks of the BLOB one at a time, independently of the rest of the record. For instance, if the BLOB is a 2-hour movie, and the client requests that the movie be played, the BLOB could be shipped several blocks at a time to the client, at just the rate necessary to play the movie.

In many applications, it is also important that the client be able to request interior portions of the BLOB without having to receive the entire BLOB. Examples would be a request to see the 45th minute of a movie, or the ending of an audio clip. If the DBMS is to support such operations, then it requires a suitable index structure, e.g., an index by seconds on a movie BLOB.

13.7.6 Column Stores

An alternative to storing tuples as records is to store each column as a record. Since an entire column of a relation may occupy far more than a single block, these records may span many blocks, much as long files do. If we keep the

values in each column in the same order, then we can reconstruct the relation from the column records. Alternatively, we can keep tuple ID's or integers with each value, to tell which tuple the value belongs to.

Example 13.22: Consider the relation

X	Y
a	b
c	d
e	f

The column for X can be represented by the record (a, c, e) and the column for Y can be represented by the record (b, d, f). If we want to indicate the tuple to which each value belongs, then we can represent the two columns by the records $\big((1, a),\ (2, c),\ (3, e)\big)$ and $\big((1, b),\ (2, d),\ (3, f)\big)$, respectively. No matter how many tuples the relation above had, the columns would be represented by variable-length records of values or repeating groups of tuple ID's and values. □

If we store relations by columns, it is often possible to compress data, the the values all have a known type. For example, an attribute `gender` in a relation might have type `CHAR(1)`, but we would use four bytes in a tuple-based record, because it is more convenient to have all components of a tuple begin at word boundaries. However, if all we are storing is a sequence of `gender` values, then it would make sense to store the column by a sequence of bits. If we did so, we would compress the data by a factor of 32.

However, in order for column-based storage to make sense, it must be the case that most queries call for examination of all, or a large fraction of the values in each of several columns. Recall our discussion in Section 10.6 of "analytic" queries, which are the common kind of queries with the desired characteristic. These "OLAP" queries may benefit from organizing the data by columns.

13.7.7 Exercises for Section 13.7

Exercise 13.7.1: A patient record consists of the following fixed-length fields: the patient's date of birth, social-security number, and patient ID, each 10 bytes long. It also has the following variable-length fields: name, address, and patient history. If pointers within a record require 4 bytes, and the record length is a 4-byte integer, how many bytes, exclusive of the space needed for the variable-length fields, are needed for the record? You may assume that no alignment of fields is required.

Exercise 13.7.2: Suppose records are as in Exercise 13.7.1, and the variable-length fields name, address, and history each have a length that is uniformly distributed. For the name, the range is 10–50 bytes; for address it is 20–80 bytes, and for history it is 0–1000 bytes. What is the average length of a patient record?

The Merits of Data Compression

One might think that with storage so cheap, there is little advantage to compressing data. However, storing data in fewer disk blocks enables us to read and write the data faster, since we use fewer disk I/O's. When we need to read entire columns, then storage by compressed columns can result in significant speedups. However, if we want to read or write only a single tuple, then column-based storage can lose. The reason is that in order to decompress and find the value for the one tuple we want, we need to read the entire column. In contrast, tuple-based storage allows us to read only the block containing the tuple. An even more extreme case is when the data is not only compressed, but encrypted.

In order to make access of single values efficient, we must both compress and encrypt on a block-by-block basis. The most efficient compression methods generally perform better when they are allowed to compress large amounts of data as a group, and they do not lend themselves to block-based decompression. However, in special cases such as the compression of a `gender` column discussed in Section 13.7.6, we can in fact do block-by-block compression that is as good as possible.

Exercise 13.7.3: Suppose that the patient records of Exercise 13.7.1 are augmented by an additional repeating field that represents cholesterol tests. Each cholesterol test requires 16 bytes for a date and an integer result of the test. Show the layout of patient records if:

a) The repeating tests are kept with the record itself.

b) The tests are stored on a separate block, with pointers to them in the record.

Exercise 13.7.4: Starting with the patient records of Exercise 13.7.1, suppose we add fields for tests and their results. Each test consists of a test name, a date, and a test result. Assume that each such test requires 40 bytes. Also, suppose that for each patient and each test a result is stored with probability p.

a) Assuming pointers and integers each require 4 bytes, what is the average number of bytes devoted to test results in a patient record, assuming that all test results are kept within the record itself, as a variable-length field?

b) Repeat (a), if test results are represented by pointers within the record to test-result fields kept elsewhere.

! c) Suppose we use a hybrid scheme, where room for k test results are kept within the record, and additional test results are found by following a

pointer to another block (or chain of blocks) where those results are kept. As a function of p, what value of k minimizes the amount of storage used for test results?

!! d) The amount of space used by the repeating test-result fields is not the only issue. Let us suppose that the figure of merit we wish to minimize is the number of bytes used, plus a penalty of 10,000 if we have to store some results on another block (and therefore will require a disk I/O for many of the test-result accesses we need to do. Under this assumption, what is the best value of k as a function of p?

!! **Exercise 13.7.5:** Suppose blocks have 1000 bytes available for the storage of records, and we wish to store on them fixed-length records of length r, where $500 < r \leq 1000$. The value of r includes the record header, but a record fragment requires an additional 16 bytes for the fragment header. For what values of r can we improve space utilization by spanning records?

!! **Exercise 13.7.6:** An MPEG movie uses about one gigabyte per hour of play. If we carefully organized several movies on a Megatron 747 disk, how many could we deliver with only small delay (say 100 milliseconds) from one disk. Use the timing estimates of Example 13.2, but remember that you can choose how the movies are laid out on the disk.

13.8 Record Modifications

Insertions, deletions, and updates of records often create special problems. These problems are most severe when the records change their length, but they come up even when records and fields are all of fixed length.

13.8.1 Insertion

First, let us consider insertion of new records into a relation. If the records of a relation are kept in no particular order, we can just find a block with some empty space, or get a new block if there is none, and put the record there.

There is more of a problem when the tuples must be kept in some fixed order, such as sorted by their primary key (e.g., see Section 14.1.1). If we need to insert a new record, we first locate the appropriate block for that record. Suppose first that there is space in the block to put the new record. Since records must be kept in order, we may have to slide records around in the block to make space available at the proper point. If we need to slide records, then the block organization that we showed in Fig. 13.19, which we reproduce here as Fig. 13.28, is useful. Recall from our discussion in Section 13.6.2 that we may create an "offset table" in the header of each block, with pointers to the location of each record in the block. A pointer to a record from outside the block is a "structured address," that is, the block address and the location of the entry for the record in the offset table.

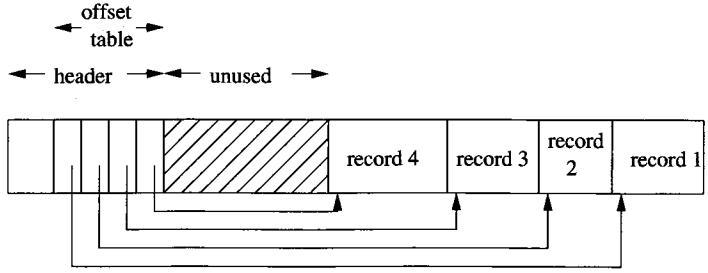

Figure 13.28: An offset table lets us slide records within a block to make room for new records

If we can find room for the inserted record in the block at hand, then we simply slide the records within the block and adjust the pointers in the offset table. The new record is inserted into the block, and a new pointer to the record is added to the offset table for the block. However, there may be no room in the block for the new record, in which case we have to find room outside the block. There are two major approaches to solving this problem, as well as combinations of these approaches.

1. *Find space on a "nearby" block.* For example, if block B_1 has no available space for a record that needs to be inserted in sorted order into that block, then look at the following block B_2 in the sorted order of the blocks. If there is room in B_2, move the highest record(s) of B_1 to B_2, leave forwarding addresses (recall Section 13.6.2) and slide the records around on both blocks.

2. *Create an overflow block.* In this scheme, each block B has in its header a place for a pointer to an *overflow* block where additional records that theoretically belong in B can be placed. The overflow block for B can point to a second overflow block, and so on. Figure 13.29 suggests the structure. We show the pointer for overflow blocks as a nub on the block, although it is in fact part of the block header.

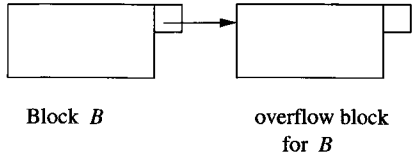

Figure 13.29: A block and its first overflow block

13.8.2 Deletion

When we delete a record, we may be able to reclaim its space. If we use an offset table as in Fig. 13.28 and records can slide around the block, then we can compact the space in the block so there is always one unused region in the center, as suggested by that figure.

If we cannot slide records, we should maintain an available-space list in the block header. Then we shall know where, and how large, the available regions are, when a new record is inserted into the block. Note that the block header normally does not need to hold the entire available space list. It is sufficient to put the list head in the block header, and use the available regions themselves to hold the links in the list, much as we did in Fig. 13.22.

There is one additional complication involved in deletion, which we must remember regardless of what scheme we use for reorganizing blocks. There may be pointers to the deleted record, and if so, we don't want these pointers to dangle or wind up pointing to a new record that is put in the place of the deleted record. The usual technique, which we pointed out in Section 13.6.2, is to place a *tombstone* in place of the record. This tombstone is permanent; it must exist until the entire database is reconstructed.

Where the tombstone is placed depends on the nature of record pointers. If pointers go to fixed locations from which the location of the record is found, then we put the tombstone in that fixed location. Here are two examples:

1. We suggested in Section 13.6.2 that if the offset-table scheme of Fig. 13.28 were used, then the tombstone could be a null pointer in the offset table, since pointers to the record were really pointers to the offset table entries.

2. If we are using a map table, as in Fig. 13.18, to translate logical record addresses to physical addresses, then the tombstone can be a null pointer in place of the physical address.

If we need to replace records by tombstones, we should place the bit that serves as a tombstone at the very beginning of the record. Then, only this bit must remain where the record used to begin, and subsequent bytes can be reused for another record, as suggested by Fig. 13.30.

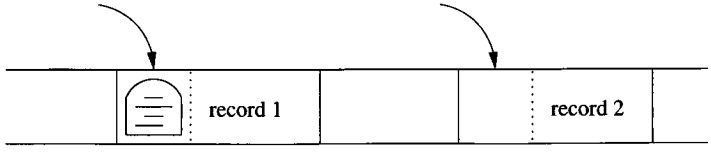

Figure 13.30: Record 1 can be replaced, but the tombstone remains; record 2 has no tombstone and can be seen when we follow a pointer to it

13.8.3 Update

When a fixed-length record is updated, there is no effect on the storage system, because we know it can occupy exactly the same space it did before the update. However, when a variable-length record is updated, we have all the problems associated with both insertion and deletion, except that it is never necessary to create a tombstone for the old version of the record.

If the updated record is longer than the old version, then we may need to create more space on its block. This process may involve sliding records or even the creation of an overflow block. If variable-length portions of the record are stored on another block, as in Fig. 13.25, then we may need to move elements around that block or create a new block for storing variable-length fields. Conversely, if the record shrinks because of the update, we have the same opportunities as with a deletion to recover or consolidate space.

13.8.4 Exercises for Section 13.8

Exercise 13.8.1: Relational database systems have always preferred to use fixed-length tuples if possible. Give three reasons for this preference.

13.9 Summary of Chapter 13

- *Memory Hierarchy*: A computer system uses storage components ranging over many orders of magnitude in speed, capacity, and cost per bit. From the smallest/most expensive to largest/cheapest, they are: cache, main memory, secondary memory (disk), and tertiary memory.

- *Disks/Secondary Storage*: Secondary storage devices are principally magnetic disks with multigigabyte capacities. Disk units have several circular platters of magnetic material, with concentric tracks to store bits. Platters rotate around a central spindle. The tracks at a given radius from the center of a platter form a cylinder.

- *Blocks and Sectors*: Tracks are divided into sectors, which are separated by unmagnetized gaps. Sectors are the unit of reading and writing from the disk. Blocks are logical units of storage used by an application such as a DBMS. Blocks typically consist of several sectors.

- *Disk Controller*: The disk controller is a processor that controls one or more disk units. It is responsible for moving the disk heads to the proper cylinder to read or write a requested track. It also may schedule competing requests for disk access and buffers the blocks to be read or written.

- *Disk Access Time*: The latency of a disk is the time between a request to read or write a block, and the time the access is completed. Latency is caused principally by three factors: the seek time to move the heads to