PARADOX 4.x FILE FORMATS

Revision 1
May 11, 1996

Author: Kevin Mitchell

PARADOX 4.x FILE FORMATS

Preface

This document is released to the public domain.

You may do anything you want with it.

IMPORTANT: YOU USE THE INFORMATION CONTAINED IN THIS DOCUMENT AT YOUR OWN RISK. I CANNOT BE HELD RESPONSIBLE FOR ANY DAMAGES THAT RESULT FROM YOUR USE OF THIS INFORMATION.

If you modify the document, please remove my name from it.

There are a couple of illustrations that would have looked better if I had used line draw characters. I avoided using line draw characters because non-US character sets do not display the characters correctly.

Kevin Mitchell
May 11, 1996

Send questions and comments via E-Mail.

My Compuserve Id is 70717,475

I specialize in programming for Paradox/DOS.

PARADOX 4.x FILE FORMATS

Introduction

This document describes the internal formats of the Paradox data and index files:

DB - Contains the data records for a table. The header contains the field names and types and other useful information.

PX - Contains the primary index for a table. If the table is unkeyed, there is no PX file for the table.

MB - Contains BLOB (Binary Large Object) data. For example, type M and B fields are blobs and the MB file contains the data for M and B fields.

X** - Contains a secondary index and has the same basic format as a DB file.

Y** - Contains the index for a secondary index (X**) And has the same basic format as a PX file.

Some of the information in this document may apply to tables created by other Paradox versions, but the information was derived by examining version 4.5 tables.

Standard reverse engineering techniques were used to obtain the information, i.e., view a file with a hex editor, make a change in Paradox, and then look at the file again to see what changed.

Note: I tried about eight different shareware hex editors. My favorite was Hex Workshop 2.10 from BreakPoint Software. It is a Windows application and is available in 16 and 32-bit versions. It can be found in Lib 4 of the PCUTIL forum. The filename is HW16V210.ZIP. It's about 300k so it only takes about 3 minutes to download at 14.4 kbps. I am in no way affiliated with the authors of Hex Workshop.

Not all of the DB and PX header fields are documented. However, there is enough information to perform the following tasks.

* Write a program to retrieve a record and its memo fields, if any.

* Write a program to browse the table in forward or reverse primary key sequence.

Write a program to use the primary index to locate a record.

* Write a program to use the secondary index to locate records.

* If Tutility can't rebuild a table, you might be able to use a hex editor to make enough repairs so that Tutility can proceed.

It would be foolhardy to attempt to write a program to update a table based on the information in this document. It might be possible to do it but it would be dangerous when you consider the number of undocumented header fields. Paradox might be quite sensitive to the content of some of the undocumented fields.

Paradox Field Types and Lengths

The following table shows the Paradox 4.x field types and the number of bytes occupied by each type in a data record.

Type	Length	
N	8	
$	8	
D	4	
S	2	
Ann	nn	1 <= nn <= 255
Mnn	nn+10	1 <= nn <= 240
Bnn	nn+10	0 <= nn <= 240
Unn	nn+10	0 <= nn <= 240 (Only created during IMPORT)

N and $ fields are stored as double-precision floating point numbers and are identical except that Paradox displays $ fields differently. Paradox automatically rounds $ fields to 2 decimal places, uses separators (commas) between three-digit groups, and puts parentheses around negative numbers.

IMPORTANT: the rounding for $ fields is for display only. The stored value is NOT rounded.

D fields are stored as a signed long integer.

Ann fields are stored as fixed-length character strings. Unused positions on the right are filled with nulls (binary zero).

M, B, and U fields are variable length and are called BLOB fields. The length shown here is the fixed-length leader that is stored in the record in the DB file.

A blob uses an extra 10 bytes in the DB record. The extra space is used to hold the length of the blob, the location of the blob in the MB file, and a modification number (used internally by Paradox).

Blob data is stored in the MB file. The leader in the DB record contains a copy of the first part of the data in MB.

A special case occurs when the entire Blob will fit in the leader. In this case the blob is stored in the leader and is not written to the MB file.

In general, memo fields should be defined as M1 to minimize record size. An M1 field takes 11 bytes in the record. A larger memo field, Mnn, can improve performance if all of the following conditions are met.

- Most records have non-blank memos
- Most memos have a length less than or equal to nn
- Most time-critical operations look at the memo fields

Numeric Formats Supported by the 80x86/7

A byte can store a value between 0 and 255 if the value is treated as unsigned. If the byte is treated as a signed value, the value can range from -128 to 127.

Short integers are 16 bits long (2 bytes). Unsigned range: 0 to 65,535. Signed range: -32768 to 32767.

Long integers are 32 bits long (4 bytes). Unsigned range: 0 to 4,294,967,295. Signed range: -2,147,483,648 to 2,147,483,547.

Double precision floating point numbers are 64 bits long (8 bytes). Floating point numbers are always interpreted as signed values. They provide approximately 15 decimal digits of precision with a decimal exponent in the range -307 to 308.

Note: Single precision (32-bits) and extended precision (80-bits) numbers are also supported but are not used by Paradox. Single precision is pretty useless because it only provides for 6 decimal digits of precision. Extended precision is not

used because it is actually intended only for intermediate results during computations. Internally, the 80x86/7 uses extended precision for all computations.

Floating Point

Paradox uses double precision floating point (64 bits) for type N and $ fields.

This floating point format has a sign bit, 11 bits for the exponent, and 52 bits for the significand.

In order to avoid having two sign bits (one for the number and one for the exponent), the 80x86 (or 80x87) uses a bias for the exponent. The bias is subtracted from the exponent value to obtain the true exponent. The true exponent is the power of 2 that the significand must be multiplied by.

The largest unsigned number that can be expressed with 11 bits is 2047. The smallest is zero. The 80x86 disallows exponents with all bits set to either 0 or 1 (there are a few exceptions to this rule). Thus, the 11-bit exponent can range from 1 to 2046. The 80x86 uses a bias of 1023. This means that numbers with binary exponents between -1022 and +1023 can be represented. This is a big enough range for most applications.

The significand is always normalized. This means that the binary point (binary equivalent of a decimal point) is placed immediately to the right of the most significant "1" bit in the number. Since normalization forces all numbers to have a 1 to the left of the binary point, the 1 is not stored in the field - it is assumed to be there. Effectively, this increases the significand length to 53.

Consider the following floating point number (the "h" after the number denotes hexadecimal notation):

4059200000000000h

The sign bit is zero.

The exponent is 405h = 1029. We subtract the bias to get 6 as the true binary exponent.

The significand is 92h (We can ignore trailing zeros). In binary this is: 10010010.

When we put back the assumed 1 and the binary point we get:

1.10010010

We multiply this by 2 raised to the 6th power. This is the same as shifting the binary point 6 places to the right.

1100100.10

In decimal this is 100 (1100100) + .5 (.10) = 100.5

Note: Binary digits to the right of the binary point correspond to negative powers of 2. The following table shows some examples.

Binary	Decimal
.1	.5 (1/2)
.01	.25 (1/4)
.001	.125 (1/8)
.0001	.0625 (1/16)

For example, decimal .75 is written in binary as .11

A floating point number is treated as zero if its exponent and significand bits are all zero. This is an exception to the rule that the exponent cannot be all "0" or all "1".

Certain operations can cause the exponent to be all "0" or all "1". The special handling that the 80x86/7 uses for these numbers is beyond the scope of this document.

Date Format

Paradox stores a date as a long integer. The integer contains the date expressed as the number of days since January 1, 1 (the year 1 A.D.).

Although dates are expressed (internally) as the number of days since 1/1/1, the lowest year that Paradox allows you to enter is 100. If a value less than 100 is entered, it is treated as 19xx, where xx is the value.

The internal representation of 1/1/100 (the lowest valid date) is 36,160 (00008D40h). The internal representation of January 2, 100

is 36,161. The internal representation of May 4, 1996 is decimal 728,783 (000B1ECFh).

Paradox accepts dates between Jan 1, 100 and Dec 31, 9999.

Blob Fields

Type M (Memo) and B (Binary) fields are blob fields.

In a DB record, a blob field is stored as a fixed-length data field (called the leader) followed by 10 bytes with the following fields:

An unsigned long integer (32 bits) that contains the offset
* of the blob's data block in the MB file and an index value.

An unsigned long integer that contains the length of the
* blob.

* An unsigned short integer (16 bits) that contains the modification number from the MB file header.

The length of the leader may be zero for type B fields. The leader for a type M field must be at least one byte.

If you define a memo field as M40, then the length of the leader is 40. If the memo data is over 40 bytes long, then the entire memo is stored in the MB file and the leader contains a copy of the first 40 bytes. If the memo data is less than 41 bytes long, then the leader contains all of the data and nothing is stored in the MB file.

The MB file is described later in this document.

Although the numeric data in a DB record data is stored in modified big endian format, the 10 bytes of blob information are stored in little endian (native 80x86) format.

We'll refer to the first four bytes after the leader as MB_Offset. MB_Offset is used to locate the blob data.

If MB_Offset = 0 then the entire blob is contained in the leader.

Take the low-order byte from MB_Offset and call it MB_Index. Change the low-order byte of MB_Offset to zero.

If MB_Index is FFh, then MB_Offset contains the offset of a type 02 block in the MB file.

Otherwise, MB_Offset contains the offset of a type 03 block in the MB file. MB_Index contains the index of an entry in the Blob Pointer Array in the type 03 block.

Refer to the MB file description for block formats.

Big and Little Endians

80x86 processors normally store numeric fields in "little endian" format. This means the least significant byte of the number has the lowest address (the little end comes first).

For example, a short integer containing decimal 10 has the following hexadecimal representation: 000Ah (the lower case h after the number means it is a hex number). An 80x86 processor would store this as 0A00. The least significant byte has the lowest address.

Many processors (like the Motorola processors used in Macintosh computers) store numbers is "big endian" format. The most significant byte has the lowest address (the big end comes first).

Processors that use big endian format store 000Ah as 000A.

Little endian format causes some complications when you attempt to sort a number as a string. 256, expressed as a short integer, is 0100h. This is stored (little endian) as 0001. If you sort a file that contains 10 and 100, then 10 (0A00) sorts after 256 (0001).

Big endian notation is a partial solution to this problem. 10 (000A) will clearly sort before 256 (0100). Sorts using big endian notation fail to work correctly when signed numbers are used. -1 is expressed as FFFFh and will sort higher than any other number.

If all numbers are signed and all floating point numbers are normalized, then there is a simple modification to big endian format that makes sorting work correctly. The sign bit (leftmost bit) is complemented (reversed) when numbers are stored. (It is complemented again before the numbers are used in computations).

-2 becomes 7FFE
-1 becomes 7FFF
 1 becomes 8001
 2 becomes 8002

Note that negative numbers will sort before positive numbers. "Larger" negative numbers will sort before "smaller" negative numbers.

Since double-precision floating point numbers on the 80x86/7 are ALWAYS signed and normalized, they will also sort correctly.

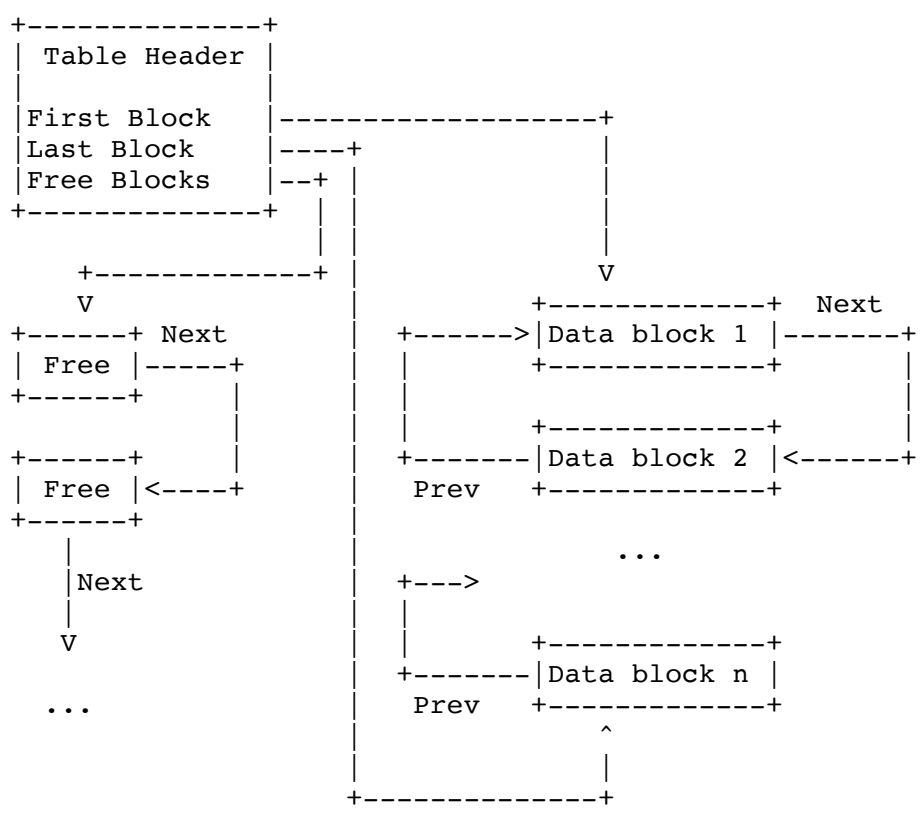
Because it is important, I will repeat that this modification to big endian format only makes sorting work correctly if all numbers are signed. This is probably why Paradox doesn't support unsigned fields.

Paradox uses little endian format (the natural 80x86 format) for control structures like file headers and block headers. It uses modified big endian format for numeric data, i.e., type N, $, D, and S fields in data records.

The DB File

The DB file contains the data records for a table. The first block in the DB file is the table header. The table header is followed by the data blocks.

The DB file has the following logical structure.



The table header contains the block number of the first data block, the last data block, and the first free block.

Blocks are numbered. The first block after the table header is block 1.

All blocks (except the table header) are the same size. This means the block number can be used to compute the offset of the block within the file:

block offset = block length * (block number - 1) + table header length

Data blocks are organized as a bi-directional linked list, i.e., the block header in each block contains the block number of the next and previous blocks in the linked list. The blocks are linked in ascending key sequence based on the first record in each block.

Within a block, records are stored in ascending sequence. The block header contains the offset of the last record in the block. When a record is deleted, any records that follow it "move up" to overwrite it and the record length is subtracted from the last record offset in the block header.

Free blocks are organized as a linked list. A free block contains the block number of the next free block but does not contain the block number of the previous free block.

A block is added to the free block list when all of the records in the block are deleted. When records are inserted into the table and a new block must be allocated, the first free block is plucked from the list. The next free block becomes the new first block.

If a new block is needed and there are no free blocks, then a new block is added to the end of the file.

Note: Block 1 is never allowed to be a free block. If block 1 is emptied, then the data from the next block in the linked list is copied to block 1. The block whose data was copied to block 1 is then added to the free list.

When you add a record, Paradox tries to put it in the data block that contains the record that is just before it in key sequence. If there is no room in the block, then:

* If you are inserting a record in the last data block, a new block is allocated and the existing block does not split.

* Otherwise, the block is split. Half of the records remain in the original block. A new block is created for the remaining records.

Of course, linked list pointers are adjusted whenever a new block is inserted. The indexes will also be updated. The primary index contains one record for each data block. The key value stored in

the index is the key of the first record in the block. (More about this later.)

Table Header

The table header is the first block in the DB file. Some of the fields in the header are described below. (The list is far from complete.)

Field type codes are: UB = Unsigned byte. US = Unsigned Short integer. UL = Unsigned Long integer.

Hex Offset	Field Type	Description
000000	US	Record length
000002	US	Length of the header block. Usually 2k (even if the data block is not 2k). The size may increase if there are a lot of fields with long field names. Worst case: 10k for 255 fields with 25-character names.
000004	UB	File type 00 - DB file for an keyed table 02 - DB file for an unkeyed table
000005	UB	Data block size code 01 - Block size is 1k 02 - Block size is 2k 03 - Block size is 3k (not used in 4.5) 04 - Block size is 4k
000006	UL	Number of records in DB
00000A	US	Number of blocks in use
00000C	US	Total blocks in file
00000E	US	First data block (always 1)
000010	US	Last block in use
000021	UB	Number of fields
000023	UB	Number of key fields
00004D	US	Block number of first free block
000078		Start of field description array

The field description array contains two bytes for each field.

The first byte contains the field type code.

Code	Field Type
01	A
02	D
03	S
05	$
06	N
0C	M
0D	B

The second byte contains the field length.

The length for $ and N is always 8.
The length for D is always 4.
The length for S is always 2.
The length for A ranges from 1 to 255 (01h to FFh).
The length for M ranges from 11 to 250 (0Bh to FAh).
The length for B ranges from 10 to 250 (0Ah to FAh).

The length for a B or M (blob) field includes 10 bytes used to hold the blob's length and its location in the MB file.

Field names start at offset 120 (78h) plus 83 plus 6 times the number of fields. Field names are in field number sequence. Each field name is a null-terminated string (00h marks the end of the string).

No data records are stored in the table header block.

DB Data Blocks

The following table describes the format of a DB data block.

Hex Offset	Field Type	Description
000000	US	Next block number (Zero if last block)
000002	US	Previous block number (Zero if first block)
000004	US	Offset of last record in block.
000006		First data record.

Records are stored contiguously. There are no gaps between records. Records contain no slack bytes between fields.

The last record offset is relative to the end of the header. Add 6 to calculate the offset from the start of the block.

If the block is empty, the offset is set to 0 minus record length.

A zero in offset means that the block contains one record.

Since Paradox knows the record length and the block length, it can use the last record offset to compute the number of records in the block and the amount of free space in the block.

If you use a hex editor to look at the data, remember that the fields in the block header are in little endian format but any numeric data fields are in modified big endian format.

If a record is deleted, records after it move up in the block and the record length is subtracted from the last record offset in the block header.

Records are stored in key sequence. If a record is inserted in the block, then records with higher keys "move down" to make room for the new record. Record length is added to the last record offset in the block header.

The PX File

The PX file contains the primary index records for a keyed table. The first block in the PX file is the index header. The index header is followed by the index data blocks.

The format of the PX file is very similar to the format of the DB file. Index blocks are chained in a bi-directional linked list and free blocks are chained in a linked list.

In addition to the list structure, the index blocks are organized into a hierarchical (tree) structure. The tree structure is the primary structure used when accessing the index.

A tree is a typical index structure. Almost any book about data bases will describe the creation and maintenance of a tree-structured index, so I won't go into too many details here. However, I will show a simple example.

In this example we will assume that we have a set of 10,000 records sorted in key sequence. The key is an integer between 1 and 10,000.

We will also assume that a data block holds 10 data records and an index block holds 10 index records. (This is unrealistic - the index block would actually hold many more index records. However, assuming 10 indexes per block simplifies the example.)

There is one index record per data block and the index record contains the key of the first record in the data block.

When we insert the first data record in the DB file, an index record is created in the PX file. The index record contains the block number of the first data block (block 1 in DB) and the key of the first record in that data block (key = 1).

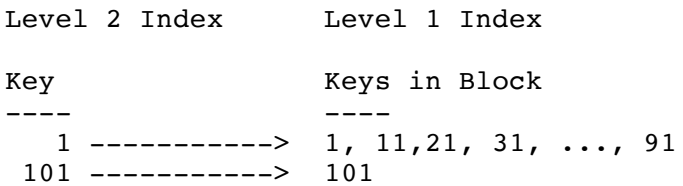
When records 2 through 10 are inserted, they are placed in the first data block in DB. No additional index records are generated.

When record 11 is inserted, a new data block is created. We must insert an index record containing the key of the first record in the block (key = 11) and the block number of the second data block. The index now contains two records.

If we continue inserting records, then we will eventually insert the record whose key is 101.

When data record 101 is inserted, there is no room for the index record in the first index block. A new index block is created to hold the index record for 101. We now have two index blocks.

An index block is created to index the two index blocks. We will refer to this as the level 2 index and refer to the first two index blocks as the level 1 index. The level 2 index contains the first key from each level 1 index block. We now have a structure that looks like this:

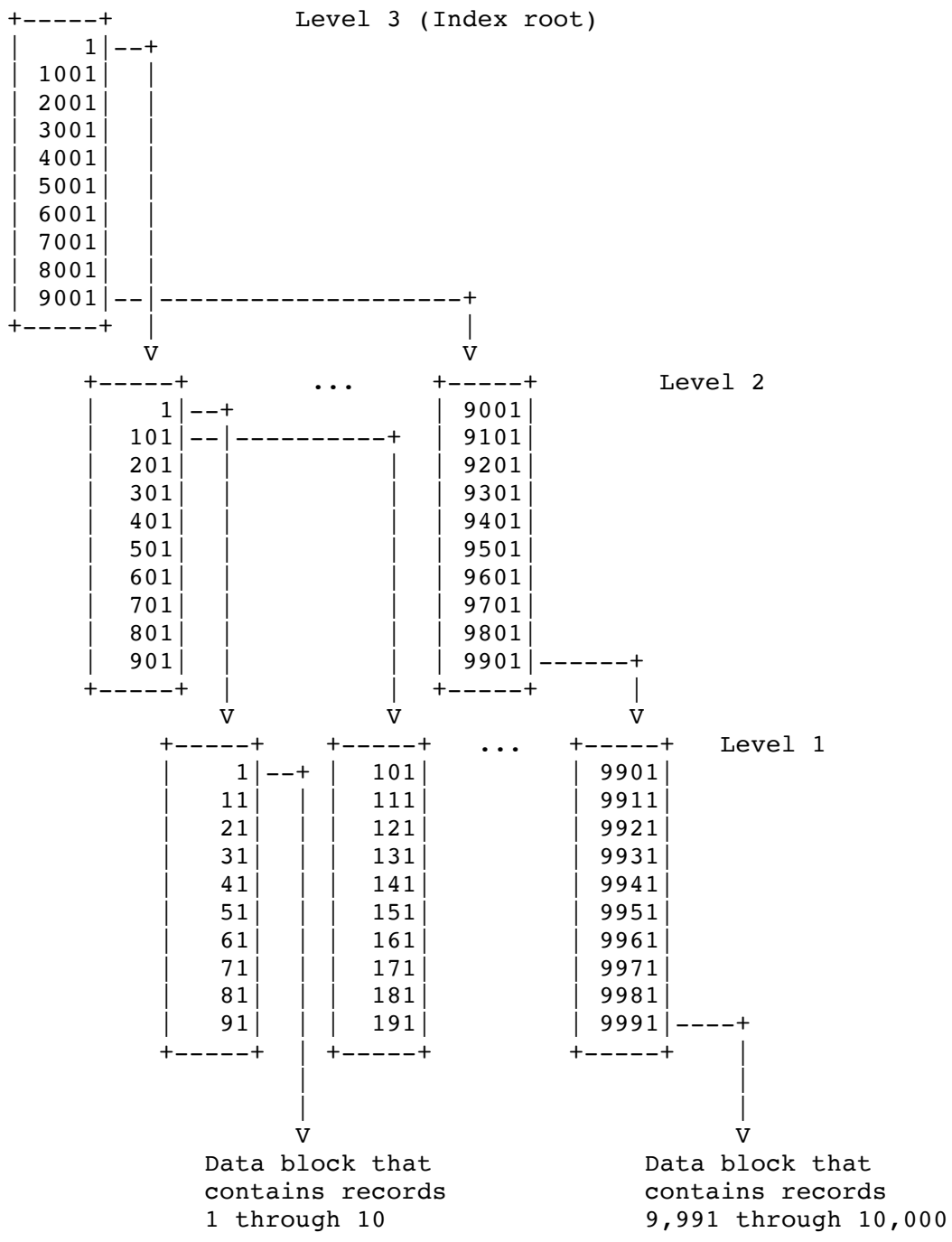


The level 2 record points to a block in the level 1 index. The "pointer" is the block number of the level 1 index block.

When record 1001 is inserted, a second level 2 index block will be created and a level 3 index block will be created to index the level 2 blocks.

After 10,000 records have been inserted, we will have a structure like the figure on the next page.

Index Structure for 10,000 Records



To find a record via the index, we start at the root (level 3) index and, at each index level, pick the entry that has the highest key that is less than or equal to the key we are trying to locate. Keep in mind that the block number we get from index records above level 1 will refer to a block in the PX file. The block number we get from the level 1 index refers to a block in the DB file.

For example, to find the data record with key 185 we proceed as follows:

* From the root index we pick 1. This points to the level 2 index that contains 1, 101, 201, ...

* From the level 2 index we pick 101. This points to the level 1 index that contains 101, 111, 121, ...

* From the level 1 index we pick 181. This points to the DB block that contains data records 181 through 190.

* Find the record with key 185 in the DB block. If the record has not been deleted, we will find it.

Since the entries in all blocks (PX and DB) are stored in key sequence, a binary search can be used to locate the desired index or data record in a block.

Statistics

Paradox keeps statistics that are (probably) used to optimize queries. The statistics are stored in the index and are updated as records are inserted and deleted.

A record in the level 1 index contains the number of records in the DB block that it points to.

A record in the level 2 index contains the sum of the statistics (DB record counts) from the level 1 index block that it points to.

A record in the level n index contains the sum of the statistics from the level n-1 index block that it points to.

For example, in the sample structure each level 1 index record contains 10. Each level 2 record contains 100. Each level 3 record contains 1000.

If the record with key = 128 is deleted, then the level 1 record with key 121 contains 9. The level 2 record with key 101 contains 99. The level 3 index with key 1 contains 999.

Index Header

The index header is the first block in the PX file. Some of the fields in the header are described below. (The list is far from complete.)

Hex Offset	Field Type	Description
000000	US	Index record length
000002	US	Length of index header size (2k)
000004	UB	File type 01 - PX file
000005	UB	Index block size code 01 - Block size is 1k 02 - Block size is 2k 03 - Block size is 3k (not used in 4.5) 04 - Block size is 4k
000006	UL	Number of records in PX
00000A	US	Number of blocks in use
00000C	US	Total blocks in file
00000E	US	First index data block (always 1)
000010	US	Last block in use
00001E	US	Block number of index root
000020	UB	Number of index levels
000021	UB	Number of fields in index

The index record length is six greater than the sum of the lengths of the key fields.

The number of fields in the index is the same as the number of key fields for the table.

Most of the block is filled with nulls.

No index records are stored in this block.

Index Blocks

Within each PX block the records (primary key values) are stored in ascending sequence. Records following a deleted record "move up".

The block header is just like the one used in the DB file.

Hex Offset	Field Type	Description
000000	US	Next block number (Zero if last block)
000002	US	Previous block number (Zero if first block)
000004	US	Offset of last record in block.
000006		First index record.

An index record has the format:

<primary key fields> followed by six bytes used as follows:

Bytes	Contents
1-2	Unsigned short integer that contains the block number associated with the key field. For a level 1 block, this is a DB file block number. For a block above level 1, this is a PX block number.
3-4	Unsigned short integer containing statistics.
5-6	Unsigned short integer. Purpose unknown. Usually contains zero.

If you use a hex editor to look at the data, remember that the fields in the block header are in little endian format.

An index record is treated as data. Numeric fields, including the data block number and statistics, are stored in modified big endian format.

The MB File

The MB file is used to store BLOB (Binary Large Object) data.

The format of the MB file is very different from the format of the DB and PX files. The blocks are not chained and the block length varies. The header is always 4k (1000h) long. Blocks that follow the header have a length that is a multiple of 4k.

Each block has the following information in the first three bytes.

UB - Record type
- 00 - Header block
- 02 - Single blob block
- 03 - Suballocated block
- 04 - Free block

US - Number of 4k chunks in this block.
The maximum size is FFFFh x 1000h = 65,535 x 4096.
This is 256 megabytes (the maximum length of a blob).

Two methods are used to allocate space for blobs.

A separate block (record type 02) is allocated for a blob over 2k bytes long. The length of the data block is the smallest multiple of 4k that is larger than the blob.

One 4k block may be suballocated (record type 03) to hold up to 64 small (under 2k) blobs.

The Blob Header

The blob header is the first block in the MB file. It is 4096 (4k) bytes long. The block contains the following fields.

Hex Offset	Field Type	Description
000000	UB	Record type = 00h (Header block)
000001	US	Size of block divided by 4k 1 because the header is 4k
000003	US	Modification count This is reset to 1 by a table restructure. Every time a blob is updated, this field is

```
            incremented. I don't know why this is done.
            The mod number is stored with the blob data.
            Again, I don't know why.

            *** ALL OF THE FOLLOWING ARE GUESSES ***

00000B US   Base size of data blocks (1000h).
00000D US   Size of suballocated data blocks (1000h).
000010 UB   Suballocation chunk size (10h)
000011 US   Number of suballocations per block (00040h)
000013 US   Suballocation threshold (0800h)
            The border line between "big" and "small" blobs.
            Big blobs get their own blocks.
            Several small blobs may be stored in one block.
```

Note: I can't even guess at the rest of the header block's fields. I assume that there's some kind of garbage collection scheme that saves data in the header block, but I haven't been able to figure it out.

Single Blob Block (Type 02)

A single blob block can appear anywhere in the MB file.

A "long" blob is stored in this kind of block.

The block length is the smallest multiple of 4k that is greater than or equal to the length of the blob.

The block header contains the following fields.

Hex Offset	Field Type	Description
000000	UB	Record type = 02h (block contains one blob)
000001	US	Size of block divided by 4k
000003	UL	Length of the blob.
000007	US	Modification number This is reset to 1 by a table restructure.
000009		Blob data starts here

Suballocated Block (Type 03)

A suballocated block can appear anywhere in the MB file.

Up to 64 short blobs may be stored in this type of block.

A suballocated block is 4k bytes long. It has a 12-byte header followed by an array of up to 64 5-byte blob pointers.

The DB field that "owns" a blob contains the offset of the block (from the start of the MB file) and the index of one of the entries in the blob pointer array. The array entry points to the blob data.

This method of using indirect pointers (a pointer to a pointer) is quite common. It simplifies garbage collection. Paradox can move data around within the block to consolidate non-contiguous chunks of free space. All it has to do is update the pointer array within the block.

The 12-byte block header contains the following fields.

Hex Offset	Field Type	Description
000000	UB	Record type = 03h (Suballocated block)
000001	US	Size of block divided by 4k 1 because the block size is 4k

Note: There are nine more bytes in the header. I have no idea what they contain.

The blob pointer array follows the header. The array has 64 entries numbered from 00h to 3Fh. Entries are used in reverse order. The 3Fh entry is used first. Then the 3Eh entry, then 3D, and so on ..

The offset (from the start of the block) of the entry indexed by i is calculated as: offset = 12 + (5 * i)

Each entry is 5 bytes long and has the following format.

Hex Offset	Field Type	Description
000000	UB	Data offset divided by 16

		The offset is measured from start of the 4k block. If this is zero, then the blob was deleted and the space has been reused for another blob (which is associated with another entry in the array).
000001	UB	Data length divided by 16 (rounded up)
000002	US	Modification number from blob header This is reset to 1 by a table restructure.
000004	UB	Data length modulo 16. If this is zero, then the associated blob has been deleted and the space can be reused For an active blob, this value will be between 01h and 10h

Note: Suballocations are made in 16-byte chunks. The first available chunk is at offset 0150h in the block. Multiply the first byte of the pointer array entry by 16 to get the offset. The next byte is the number of chunks. The last byte tells you how many bytes of data there are in the last chunk. I don't know the purpose of the modification number.

For example, if an array entry looks like: 25030F0007 then the data associated with the entry starts at offset 0250h (25h times 10h) and has 10h times 03h bytes allocated (48 bytes). The actual data length is 27h (39 bytes) because there are only 7 bytes of data in the last 16 byte chunk. The modification number is in little endian format and is 000Fh (15).

Free Block (Type 04)

A free block can appear anywhere in the MB file.

The block length is a multiple of 4k.

If there are several contiguous free blocks, then the combined length is placed in the first one.

The block header contains the following fields.

Hex Offset	Field Type	Description
000000	UB	Record type = 04h (free block)
000001	US	Size of block divided by 4k

If the blob in a type 03h block (single blob block) is deleted, then its block becomes a free block. If all of the blobs in a type 02h (suballocated block) are deleted, then the block becomes a free block.

Deletions happen more often than you might imagine. Whenever you modify a blob, the original blob is deleted and the modified version is saved as a new blob.

Paradox never updates a blob "in-place".

X** File

An X** file contains the data records for a secondary index. There is one record for each record in the DB file.

An X** file has the same logical format as a DB file.

The X** data record contains the secondary index fields followed by the primary index fields. An additional type S field named "Hint" is the last field in the record. All fields except "Hint" are included in the record key.

For example, if your data record has the key "Custid" and you define a compound secondary index on "Last Name" and "First Name", then the X** record contains four fields: [Last Name], [First Name], [Custid], and [Hint]. The first three fields are in the primary index for the X** file.

[Hint] contains the block number of the DB file block that contains the record associated with the index record. This means that the DB record can retrieved directly. It doesn't have to be located via the primary index in the PX file.

Although [Hint] is defined as a type S field, it is treated as an unsigned integer by Paradox. Paradox knows it's a block number.

Note: If you specify more than 16 secondary index fields, then only the first 16 fields are included in the index. Primary index fields may be included in the index but the first primary index field may not be the first secondary index field.

X** File Header

The X** file header is the first block in the X** file. It has the same format as the Table Header (DB File Header).

Hex Offset	Field Type	Description
000000	US	Record length
000002	US	Length of the header block. Usually 2k (even if the data block is not 2k).
000004	UB	File type 08 - Secondary index data file

```
000005 UB    Data block size code
             01 - Block size is 1k
             02 - Block size is 2k
             03 - Block size is 3k (not used in 4.5)
             04 - Block size is 4k
000006 UL    Number of records in X**
00000A US    Number of blocks in use
00000C US    Total blocks in file
00000E US    First data block (always 1)
000010 US    Last block in use
000021 UB    Number of fields
000023 UB    Number of key fields
00004D US    Block number of first free block
000078       Start of field description array
```

The field description array contains two bytes for each field.

(See the Table Header description.)

Field names start at offset 120 (78h) plus 83 plus 6 times the number of fields. Field names are in field number sequence. Each field name is a null-terminated string (00h marks the end of the string).

For a compound index (more than one secondary index field), the fields have the same names that they have in the DB file.

For a simple index (only one secondary index field), the name of the index field is replaced by the name "Sec Key". This is really stupid!

The primary key field names follow the secondary index field names. "Hint" follows the primary key field names.

Immediately after the terminating null for [Hint], there is a series of n unsigned short integers, where n is the number of fields in the record. The first m of these integers are the field numbers (in DB) of the secondary index fields, where m is the number of fields in the secondary index.

Immediately after the integers, there is a null-terminated string that contains the name of the sort order, e.g., "ascii".

The name of the index follows the sort order string. The index name (a.k.a., label) is a null-terminated string.

No secondary index data records are stored in the X** header block.

X** Data Blocks

The data blocks have the same format as a DB data block.

Y** File

A Y** file is the primary index for an X** file.

Its logical format is identical to the format of the PX file.

Y** Header

The Y** header is the first block in the Y** file. It has the same format as the PX file header. Some of the fields in the header are described below. (The list is far from complete.)

Hex Offset	Field Type	Description
000000	US	Index record length
000002	US	Length of index header size (2k)
000004	UB	File type 05 - Y** file
000005	UB	Index block size code 01 - Block size is 1k 02 - Block size is 2k 03 - Block size is 3k (not used in 4.5) 04 - Block size is 4k
000006	UL	Number of records in Y**
00000A	US	Number of blocks in use
00000C	US	Total blocks in file
00000E	US	First index data block (always 1)
000010	US	Last block in use
00001E	US	Block number of index root
000020	UB	Number of index levels
000021	UB	Number of fields in index

The index record length is six greater than the sum of the lengths of the key fields.

No index records are stored in this block.

Y** Index Blocks

Same format as the index blocks in the PX file.