# DavisBase Nano
# File Format Guide
## Storage Definition Language (SDL)

**Version 1.3**

**Chris Irwin Davis**

## 1. Introduction

DavisBase Nano is a relational database management system (DBMS) that implements a small subset of the SQL commands specification. DavisBase is not intended for commercial applications—but instead, for use in an academic setting for use by students to gain an understanding of relational database *internals*. This document assumes that the reader already possesses a basic understanding of (1) basic SQL, (2) relational model theory, and (3) hexadecimal file viewers/editors ("hex editor") such as the built-in Unix command `hexdump`.

This File Format Guide describes only the Storage Definition Language (SDL) of DavisBase. It is intended to be a subsection of the overall DavisBase Specification, which also includes a data query language (DQL), data definition language (DDL), data control language (DCL), and data manipulation language (DML). Additionally, an API will be included in a later version that is based on a subset of the JDBC specification.

The DQL

DavisBase SDL file architecture is similar to that of several widely used open-source databases, including SQLite, MySQL, and PostgreSQL. In fact, one of the primary goals of DavisBase is to be an educational platform for learning the internals of relational databases. Implementing file structures inspired by those of widely known DBMSs provides insight into the mechanics of relational databases in general.

We note that DavisBase is not fully ACID[1] compliant! There is neither a system log nor a formal mechanism for rollback or recovery of failed transactions.

## 2. Database Files

All DavisBase files are implemented as ***page-based files***. That is, files are subdivided into same-sized logical sections called *pages*. DavisBase supports page sizes of $2^n$ bytes, where $n$ is an integer in the range 9-15. Therefore, page size is configurable from 512B–32768KB in $2^n$ byte increments.

Despite its similarity in name to the real world *physical object* called a "file" in paper form that can be physically touched and held, a computer file is a *virtual* construct. The term "file" can be defined in various ways since it is a logical construct whose data is stored in same-sized chunks of memory called *pages*. These pages may be either in volatile memory (RAM, "main memory") or non-volatile memory (hard disk, SSD, CD-ROM, Blu-Ray, "thumb drive", etc.). The pages are not necessarily stored in contiguous, adjacent addresses in memoery.

DavisBase SDL uses two types of files to store information: ***table files*** and ***index files***. These are used to store table data and index data, respectively. We note that full-featured commercial databases utilize additional file types—e.g. view files, script files, macro files, etc. These additional file types are not supported in v1.*x* of DavisBase Nano.

DavisBase uses a *file-per-table* approach. That is, each table and each index are each stored in a single, separate file in the underlying OS. Each Table File is stored in an OS file whose name is *`table_name`*`.tbl`, and each Index File is stored in a a file whose name is *`index_name`*`.ndx`.

[1] https://en.wikipedia.org/wiki/ACID_(computer_science)

All pages of a given file are required to be the same size. All files within the same table space must share the same page size. Once a table space has been initialized, the page size of all of its constituent files is static and constant.

A consequence of this is that the memory footprint of any file must be a multiple of the page size. Files cannot be comprised of partial pages. If data exceeds the capacity of a page, by even one byte, the file must be increased in size by an additional full page to accomodate it.

Most modern operating systems use a similar strategy. They have a file system that employs a 4k page size (4096 or $2^{12}$ bytes, exactly). This is the reason that Windows and MacOS display two different file sizes in their "file properties" attributes—(a) *file size*, and (b) *size on disk*.

Pages are numbered using a zero index convention, like C-style array indexes. That is, the first page in a file is numbered page 0.

Page size is represented in bytes.

All page numbers are 16-bit two's complement integers.

### 2.1. Offsets

***Offsets*** are the mechanism used to locate specific data, cells, or other elements within a file.[2] Offsets are expressed as non-negative integers that represent the number of bytes from a given point of reference. There are two kinds of offsets used by DavisBase, ***file offsets*** and ***page offsets***.

***File offsets*** indicate the location of an element expressed as the number of bytes from the beginning of a file. File offsets are most commonly used to identify the location of an entire page within a file—by multiplying page number time page size. For example, in a file with a page size of 1024 bytes, page 7 will begin at 7168 (i.e. 7*1024) bytes from the beginning of the file.

***Page offsets*** indicate the location of an element within a page expressed as the number of bytes from the beginning of a page. The absolute location of an element within a file is therefore its ***page offset*** plus the file offset of the page in which it resides.

Each record in a table file and each index entry in an index file is stored in a ***cell***. The location of cells on a page are stored at the bottom (high address) of each page. The location of each cell on a page is encoded in a page offset. A list of the page offsets for the cells that page are located in a list of offsets at the top (low address) of each page. The list of page offsets for the cells are maintained in sorted order according to the ***key of the cell***.

- The ***key*** of a table file cell is the `rowid` of the record in the cell.
- The ***key*** of an index file cell is the value of the indexed attribute.

### 2.2. Table Files

A ***table file*** stores the records (i.e. rows) of a given table in a single file.[3] Each record in a table file has an automatically generated unique identifier that is used by DavisBase for internal housekeeping. Each

[2] Offsets are the mechanism used by almost *all* relational databases to locate elements within a file.

[3] In the relational model, the "tuples of a relation".

`rowid` is represented by a 32-bit two's complement integer.[4] This is essesntially a hidden extra column that is included in every table. Various databases use different names for this unique identfier: `rowid` , `row_id`, and `uid` are all common names. DavisBase uses the name `rowid`. Each `rowid` is automatically generated based on the following strategy—The first record inserted into a given table is numbered "1", then each subsequent `rowid` increases in one integer increments with no gaps. If a record is deleted, its `rowid` is ***never*** reused or re-purposed. If a record is deleted, its "deleted" byte is set to 1 (true), but the data is not purged from the page.

Table files are implemented as a variation of $B^{+}$-trees which we call a ***$B^{+1}$-tree***. Each node of $B^{+1}$-tree is a page of the file. Table records reside solely in leaf pages of the $B^{+1}$-tree. Initially, before any records are inserted, a table file is exactly one page in size. Records may be added to the single-page table file until it reaches capacity. At this time the next record insert will cause an overflow of the page (i.e. $B^{+1}$-tree node). The result will spawn a new sibling page (node) to store the new record, and a new parent node which contains a single cell—the rowid key where the demarcation is between rowid's in the left child and rowid's in the right child. Because `rowid` values are monotonically increasing, all new records will always be inserted in the last leaf node of a table file. Therefore, each page split does not need to move half of the records to a new page when an overflow occurs. Hence, all overflow-split actions of a table file page are "plus one" where the single record that caused the overflow is inserted into a new page by itself. Table files

### 2.3. Index Files

Index files provide efficient access paths to records within table files. All DavisBase indexes provide access paths to records based only on *single columns*. Multi-column indexes are not supported by DavisBase Nano.

Index Files are implemented as page-based ***B-tree*** files where each B-tree node is a page of the file. Index Files store indexing information that provide an access path to a related Table File via the `rowid`.

Index files in DavisBase Nano use ***dense indexes***. An index file will contain an entry for every record in its associated table file. If there are multiple records with identical non-unique index values, each record will have an entry in the index file.

Table Files are not required to have any Index Files. However, every Index File must have ***exactly one*** Table File that it indexes. An index file is created under two scenarios:

1) A column is declared with either the `PRIMARY KEY` or the `UNIQUE` constraint in the `CREATE TABLE` statement.
2) A `CREATE INDEX` statement is executed that references an existing table.

If a table schema is created without an explicit PRIMARY KEY, users may insert records that contain duplicate values for all user-defined columns. This would seem to violate relational theory which requres that all tuples (records) be unique. However, every record is still unique since each contains a unique value for its hidden `rowid`. Most relational databases use this strategy of a hidden unique identifier for internal housekeeping.

[4] Therefore, no table may contain more than 231–1 (2147483647) total unique records over the lifetime of a table.

### 2.3.1. Row IDs

Each record in a table file is uniquely identified by an internally assigned key called the *Row ID*. The Row ID is a column (field or attribute) that is created automatically when a table is created with the column name `rowid`.

The `rowid` column *does not display* when a query is executed that references the `SELECT *` column wildcard. Only user defined columns will display. However, a user may explicity add `rowid` to the list of attributes in the `SELECT` clause to force it to display. For example,

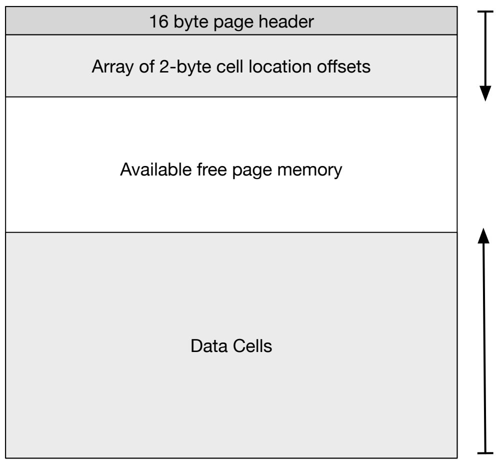
```
SELECT rowid, * FROM table_name;
```

Rowid is an integer. It is automatically generated. It is always monotonically increasing. By nature `rowid` is unique. If a record is deleted, its `rowid` is never re-used or re-purposed.

## 3.

## 4. Page Formats

All pages use the basic format of a fixed size header of 16-bytes followed immediately by an array of 2-byte integers that indicate the location of cells within the page (i.e. page offsets). Cells contained in the cell area are written from highest-to-lowest page address upward.




### 4.1. Page Header Format

Page headers are located at the top—page offset zero (0)—of each page within a file. A page header is 16 bytes size. Not all bytes within the header are currently used. By convention, unused bytes are initialized to byte value zero (0x00), however the value is irrelevant since they are never referenced.

| Page Offset | Element Size | Description |
|---|---|---|
| 0x00 | 1 | The one-byte flag at page offset 0x00 indicates the type of b-tree page.<br>• A value of 2 (0x02) means the page is an ***index*** b-tree ***interior*** page.<br>• A value of 5 (0x05) means the page is an ***table*** b-tree ***interior*** page.<br>• A value of 10 (0x0a) means the page is an ***index*** b-tree ***leaf*** page.<br>• A value of 13 (0x0d) means the page is a ***table*** b-tree ***leaf*** page.<br>Any other value for the b-tree page type is an error.<br>These are the same page type values used by SQLite. |
| 0x01 | 1 | Unused |
| 0x02 | 2 | The two-byte integer at offset 2 designates the number of cells on the page. The value of this two-byte integer is also the number of cell offsets in the array that begin at location 0x10 |
| 0x04 | 2 | The two-byte integer at offset 4 designates the page offset for the start of the cell content area. |
| 0x06 | 2 | The two-byte integer at page offset 0x06 indicates the root page of the file. All pages in the file will have the same value for this two bytes. |
| 0x08 | 2 | The two-byte integer page pointer at page offset 0x08 has a different role depending on the b-tree page type:<br>• For a Table or Index ***interior page*** - value is the page number of rightmost child<br>• For a Table or Index ***leaf page*** - value is the page number of sibling to the right |
| 0x0A | 2 | The two-byte integer page pointer at offset 0x0A references the page's parent. If this is a root page, then it has no parent and the special value 0xFFFF is used. |
| 0x0C | 4 | Unused |
| 0x10 | 2 x *cells on page* | An array of 2-byte integers that indicate the page offset location of each data cell. The array size is 2*n*, where *n* is the number of cells on the page. The array is maintained in key-sorted order. |

*Table 1 - Page Header Format*

The number of cells on any type of page is represented by a 2-byte two's complement integer. Therefore, no page may contain more than $2^{15}–1$ (32767) cells.

Page numbers for both types of file are represented by a 4-byte two's complement integer. Therefore, no file may contain more than $2^{31}–1$ (2147483647) pages.

### 4.2. Cell Formats

The **cell** is the basic unit of content for all types of pages. The cell content area of each page is located at the high offset values, i.e. cell content grows from the end of each page upwards.

The array of cell locations immediately following the page header is an array whose *order is maintained to be sorted according to criteria within the cells*, depending upon file type. Table file cells are maintained in rowid sorted order.

The format of a cell depends on which kind of page the cell appears on. The following shows the elements of a cell, in order of appearance, for the various page types.

Table File Leaf Cell (page header 0x0d):

- A 1-byte integer which is the total number of bytes of payload
- A 4-byte big-endian integer which is the integer key, a.k.a. “rowid"
- The payload bytes

Table B-Tree Interior Cell (page header 0x05):

- A 2-byte big-endian page number which is the left child pointer.
- A 4-byte big-endian integer which is the `rowid`

Index B-Tree Leaf Cell (page header 0x0a):

- A 1-byte integer which is the total number of bytes of key payload
- The payload bytes

Index B-Tree Interior Cell (page header 0x02):

- A 2-byte big-endian page number which is the left child pointer.
- A 1-byte integer which is the total number of bytes of key payload
- The payload bytes

| File Type | Page Type | Cell Header | | | Cell Body |
|---|---|---|---|---|---|
| | | 2-byte int | 2-byte int | 4-byte int | N-byte array |
| | | Left Child Page# | Bytes of Cell Payload | Rowid | Payload |
| Table | Leaf | | ✔ | ✔ | ✔ |
| | Interior | ✔ | | ✔ | |
| Index | Leaf | | ✔ | | ✔ |
| | Interior | ✔ | ✔ | | ✔ |

*Figure 1: Cell Formats*

Note that only interior pages have child pointers, since leaf pages don’t have childeren.

The following two tables indicate the contents of the cell body (i.e. the "payload") of record cells and index cells, respectively.

| Record Header | | Record Body |
|---|---|---|
| **Number of Columns** | **List of Column Data Types** | **List of Column Data Values** |
| **1-byte INT** | **Array of 1-byte INTs** | ***N*-bytes** |

*Figure 2: Record Format (i.e. Cell Body of Table Leaf Cell)*

### 4.3. Data Types

All data types represented by DavisBase by a 1-byte integer that designates their type. Notice that TEXT (i.e. string) data types are represented by at *set* of 1-byte values. Each particular length of string is represented as its own unique data type. For example, a five character is string is data type 0x11 (i.e. 0x0C + 0x05) and a 43 (0x2B) character string is data type 0x37 (i.e. 0x0C + 0x2B).

Note that data types 0x00 and 0x0C are the *NULL* value and *empty string*, respectively. Neither would require any bytes in the record body.

| Date Type Code | Data Type Name | C | Description |
|---|---|---|---|
| 0x00 | NULL | 0 | Data type is a NULL (requires no bytes in the record body) |
| 0x01 | TINYINT | 1 | Data type is an 8-bit twos-complement integer. |
| 0x02 | SMALLINT | 2 | Data type is a big-endian 16-bit twos-complement integer. |
| 0x03 | INT | 4 | Data type is a big-endian 32-bit twos-complement integer. |
| 0x04 | BIGINT, LONG | 8 | Data type is a big-endian 64-bit twos-complement integer. |
| 0x05 | FLOAT | 4 | Data type is a big-endian IEEE 754-2008 32-bit floating point number. |
| 0x06 | DOUBLE | 8 | Data type is a big-endian IEEE 754-2008 64-bit floating point number. |
| 0x08 | YEAR | 1 | Data type is an 8-bit twos-complement integer. Both positive and negative numbers are supported in the range -128 to 127. This indicates a year with respect to the epoch year 2000. |
| 0x09 | TIME | 4 | Data type is a big-endian 32-bit twos-complement integer. Indicates time of day in milliseconds since midnight, i.e. "millis". Note that only values of 0-86400000 (0x00-0x05265c00) are valid. |
| 0x0A | DATETIME | 8 | Data type is a big-endian unsigned LONG integer that represents the specified number of milliseconds since the standard base time known as "the epoch". It should display as a formatted string string: YYYY-MM-DD_hh:mm:ss, e.g. `2016-03-23_13:52:23`. |
| 0x0B | DATE | 8 | A datetime whose time component is 00:00:00, but does not display. |
| 0x0C + *n* | TEXT | 0+ | Value is data type of "ASCII string of length *n* ". C-style string null terminators are not used or needed. A value of 0x0C is the empty string and would have no bytes in the record body. |

*Table 2 - Data Types and Their Implementation*

Note that only strings of size 0-115 ASCII characters are supported. Any value 0x0C or above represents an ASCII string data type. Each length string 0-115 has its own unique data type. For example, 0x41 data type is a string of length 53 characters.

DavisBase 目录由两个表组成，其中包含有关每个用户表的元数据。您可以选择在目录本身中包含有关两个目录文件的元数据。这两个表（及其关联的实现文件）具有以下表架构，就好像它们是通过常规 CREATE 命令创建的一样。

## 5. Database Catalog (meta-data)

The DavisBase Catalog consists of two tables containing meta-data about each of the user table. You may optionally choose to include meta-data about the two catalog files in the catalog itself. These two tables (and their assocated implmentation files) have the following table schema, as if they had been created via the normal **CREATE** command.

```
CREATE davisbase_tables (
   rowid INT,
   table_name TEXT,
   record_count INT,   -- optional field, may help your implementation
   avg_length SMALLINT -- optional field, may help your implementation
   root_page SMALLINT  -- optional field, may help your implementation
);
```

```
CREATE davisbase_columns (
   rowid            INT,
   table_name       TEXT, -- optionally table_rowid INT
   column_name      TEXT,
   data_type        TEXT,
   ordinal_position TINYINT,
   is_nullable      TEXT
);
```

If you choose to include these two tables in the catalog itself, their content would intially be:

```
SELECT * FROM davisbase_tables;

rowid   table_name
--------------------------
1       davisbase_tables
2       davisbase_columns
```

```
SELECT * FROM davisbase_columns;

rowid   table_name         column_name      data_type  ordinal_position is_nullable
---------------------------------------------------------------------------------
1       davisbase_tables  rowid            INT        1                NO
2       davisbase_tables  table_name       TEXT       2                NO
3       davisbase_columns rowid            INT        1                NO
4       davisbase_columns table_name       TEXT       2                NO
5       davisbase_columns column_name      TEXT       3                NO
6       davisbase_columns data_type        TEXT       4                NO
7       davisbase_columns ordinal_position TINYINT    5                NO
8       davisbase_columns is_nullable      TEXT       6                NO
9       davisbase_columns column_key       TEXT       7                YES
```

## 5.1. Database Catalog

The database catalog (i.e. meta-data) for DavisBase is stored in specially designated tables. These are also referred to as **system tables**.[5] The Table Files associated with these are structured exactly the same as Table Files generated with user-defined tables.

DavisBase 的数据库目录（即元数据）存储在专门指定的表中。这些表也称为系统表。与这些表相关的表文件的结构与使用用户定义表生成的 5 个表文件完全相同。

| Index Header | | | Index Body |
|---|---|---|---|
| Number of rowids associated with index value | Index Data Type | Index Value | List of rowids associated with index value |
| 1-byte INT | 1-byte INT | *N*-bytes | Number of rowids x 4-bytes |

*Figure 3: Index Format (i.e. Cell Body of All Index Cells)*

[5] The term “system tables” distinguishes them from “user tables”, those createMySQL groups these system tables together in what it calls the INFORMATION_SCHEMA.