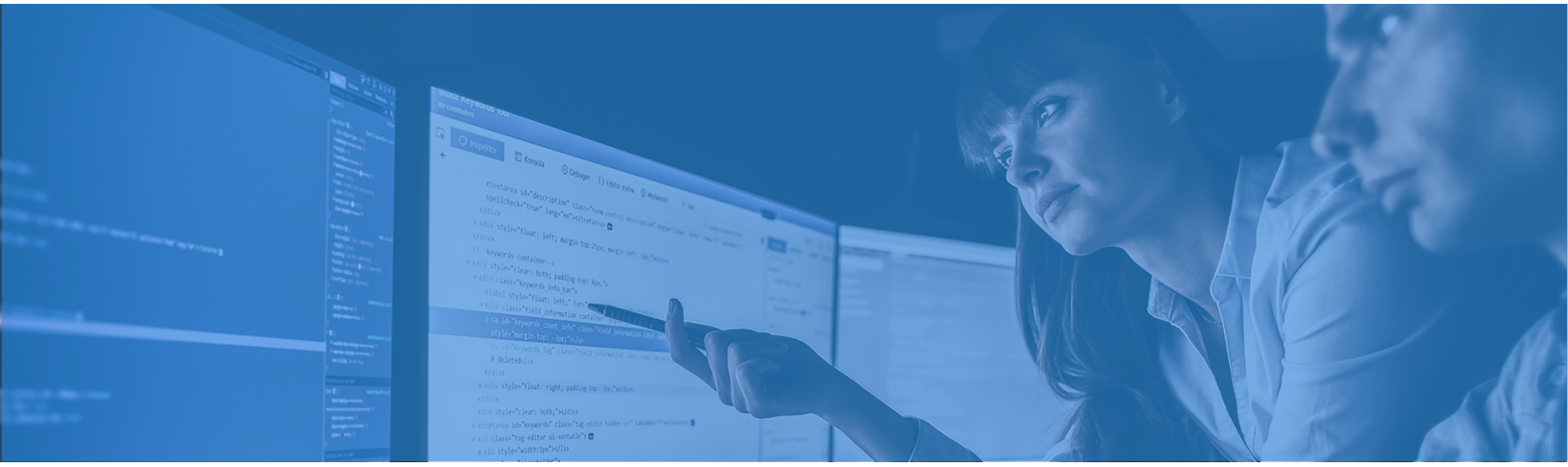
ACTIAN™



A Developer-Centric Approach to Modern Edge Data Management

The case for dumping your file systems and SQLite and moving to Actian Zen

A substantial majority of embedded developers in the IoT and complex instrumentation space use C, C++, or C# to handle data processing and local analytics. That's in part because of how easy it is to handle direct I/O for devices and internal systems components as well as more complex digitally-enhanced machinery through some variations of `inp()` and `outp()` statements. It's also easy to manipulate collected data using familiar file system statements such as `fopen()`, `fclose()`, `fread()`, and `fwrite()`. This is the path of least resistance. Almost anyone who takes a programming class (or just takes the time to learn how) can use these statements to interact with data at the file system level.

The problem is that file systems are very simple. They don't do much by themselves. When it comes down to document and record management, indexing, sorting, creating and managing tables, and so on, there's only one operative statement: `DoItYourself()`. And we're not even talking about rare or rocket science-level activities, here. These are everyday activities that that you'd find in any database system.

Wait! It's the D-word! May as well be the increment of the ASCII character pointer by two to the … you know what word.

We're among friends, here, so let's not mince words: Developers of embedded applications *hate* databases, and it's not hard to appreciate why. Traditional databases have always been far too large to deploy into IoT and mobile environments. They're can

be expensive and difficult to provision; they tend to need constant attention of the kind only an expensive DBA can administer. Sure, some newer open source databases are somewhat less costly than an Oracle or Microsoft database (though you'll still paying for service and support), but most of these are built for the Cloud. Developers building for the IoT or mobile need something designed for the Edge, for a gateway, that can be embedded in a device itself.

Out of the primal ooze

One open source database has stood out as an exception among all cybersaurs described above—a very compact database that is serverless, portable, embeddable, and accessible via most popular programming languages: SQLite.

SQLite has been around for more than 20 years, and it is nearly ubiquitous as a data cache in web and mobile applications. As the name implies, SQLite makes use of standard query language (SQL) and, unlike other popular open-source SQL databases (MySQL, Postgres, MariaDB, etc.), it *can* operate in a serverless, zero database administration (Zero-DBA) manner. In deploying SQLite, developers discovered that they could rely on SQL for transaction-oriented data management and forego all the DIY file system interactions. Developers also discovered a performance advantage when deploying SQLite, as the database could perform block reads and writes of larger data sets around 35% faster than a native file management system.

SQLite provided a step up the evolutionary ladder from simple file management systems, and application developers could use it to address a variety of early Edge use cases more effectively than they could with flat files. But like many creatures on the lower rungs of the evolutionary ladder, SQLite has its limits. The vast majority of tomorrow's edge use cases will involve multiple channels and I/O data rates; multiple applications, processes, and threads; the need to manipulate large data sets at speed; and, the need to secure data both rest and in transit. SQLite was simply never designed to meet these needs.

Developers who found SQLite to be perfect for single user embedded mobile applications and who have tried to use it to build more sophisticated edge applications have encountered challenges when trying to accommodate all these needs. Today's response to these challenges is not fundamentally different than it was when addressing the shortcomings of simple file management systems: `DoItYourself()`. To stretch SQLite into something that can handle multi-channel, multi-process, high-speed Edge data environments, the DIY now involves a mash-up of self-developed code, GitHub downloads, and purchases from very small development shops providing add-on components for server support, security, and connectivity functionality. It's a more sophisticated level of DIY, but it's still DIY.

Stepping up the evolutionary ladder

A banana slug is more sophisticated than an amoeba, but it's far less sophisticated than, say, a peregrine falcon. When it comes to meeting tomorrow's Edge and embedded app needs, developers need something that can fly, not crawl.

That's where Actian Zen family of databases comes in. Zen is to SQLite what SQLite was to flat file management systems. It provides the full set of capabilities needed to meet the needs of modern edge data management.

Actian Zen Core edition provides developers with the same characteristics that made SQLite appealing as a database: Zero-DBA, a serverless option, portability, and embedded support for most popular programming languages, including C, C++, and C#. Actian Zen extends the access options by including a NoSQL as well as a SQL API and extends access even further through SWIG (simplified wrapper and interface generator) support, which makes the SQL and NoSQL APIs accessible to developers using Python, Perl, and PHP on Windows and Linux systems.

The evolutionary leap embodied in Actian Zen, though, is that Actian Zen Core edition (with a footprint of <5MB) has an analogous Edge edition that runs on a server or gateway (with a footprint of less than 50MB), as well as Enterprise and Cloud brethren that can run on even more powerful devices and containers (and still take up less than 200MB). Every edition is based on the same architecture, so they complement and interact with one another like nothing in the world of SQLite.

Building with Actian Zen

Incorporating Actian Zen into a code base requires nothing more than adding the 32-/64-bit C/C++ libraries (btrieveC.h, btrieveCpp.h, btrieveC.lib, btrieveCpp.lib) into a project. If you want to use Perl, Python, or PHP, you can also need to add the interface libraries for SWIG. SQL calls to Zen are largely the same SQL calls you would use to interact with SQLite. But what will be more exciting for IoT and mobile developers striving to overcome the performance limitations of SQLite is the option to use the NoSQL API (Btrieve 2). The Btrieve 2 API is as simple to use as a file management system yet provides access to all the functionality of the underlying database but at 100X the performance of SQLite.

How easy is it to interact with a Zen database engine via the Btrieve2 API? Let's look.

To start, we must first import the btrieveCpp.h header in the application code:

```
#include "btrieveCpp.h"
```

Then to connect to the engine, you must create an instance of the `btrieveClient` object:

```
BtrieveClient btrieveClient (0x4232, 0);
```

That's all the setup required. The first parameter is a `serviceAgentIdentifier`, which identifies each instance of your application to the engine. It can be any two-byte value larger than "AA" (0x4141); in this instance, 0x4232 = "B2." The second parameter is the `clientIdentifier`, which must be a 2-byte integer. If you are writing a multithreaded application, each thread needs to provide a unique client identifier to the engine.

Now, let's do some data collection.

Zen data is stored in a file. For example, a file might contain sensor data from a blood pressure (BP) monitor. Each record would consist of the follow data:

- 8-byte time stamp indicating when the BP reading was taken
- 2-byte integer for a systolic value
- 2-byte integer for a diastolic value
- 1-byte character for an evaluation code

Given the characteristics of the data, we would need to create a file to hold 13-byte records. The snippet below defines a `BPrecord_t` structure that accommodates the blood pressure record. The `#pragma pack` statements ensure that the record is actually 13 bytes long, with no extra alignment bytes added by the compiler.

```
#pragma pack(1)

typedef struct {

  uint64_t timeTaken;

  uint16_t systolic;

  uint16_t diastolic;

  char EvalCode;

} BPrecord_t;

#pragma pack()
```

To create a file for these records, we need to allocate a `btrieveFileAttributes` object and use the `SetFixedRecordLength` property to specify our record size.

```
Btrieve::StatusCode status;

BtrieveFileAttributes btrieveFileAttributes;

status = btrieveFileAttributes.SetFixedRecordLength(sizeof(BPrecord_t));
```

You can add other file attributes (that the file be compressed before written to disk, for example), but record size is the only required attribute.

Next you must instruct the Zen engine to create the data file using the `FileCreate()` method on our `btrieveClient` session object. We can also specify a creation mode to indicate if the file should be overwritten if it already exists.

```
static char* btrieveFileName = (char*)"Pressures.btr";

status = btrieveClient->FileCreate(&btrieveFileAttributes,
btrieveFileName, Btrieve::CREATE_MODE_NO_OVERWRITE);

if ((status != Btrieve::STATUS_CODE_NO_ERROR) &

   (status != Btrieve::STATUS_CODE_FILE_ALREADY_EXISTS))

{

printf("Error: BtrieveClient::FileCreate():%d:%s.\n", status,
```

```
    Btrieve::StatusCodeToString(status));
}
```

The example above includes error checking using the built in `StatusCode` enumerations in the Btrieve class. For simplicity, we will not show error checking in subsequent code samples.

To insert data into a file you would first open the file:

```
BtrieveFile btrieveFile;
status = btrieveClient->FileOpen(btrieveFile, btrieveFileName, NULL,
   Btrieve::OPEN_MODE_NORMAL);
```

Here, we allocate a `btrieveFile` object and use our `btrieveClient` session to open the file we previously created. The third `status` parameter ("NULL" in this example) can be used to pass in a Btrieve owner name, which acts a password for securing (and optionally encrypting) a data file. Our file does not have an owner name, so we pass in NULL.

To insert a record, we allocate a record buffer for the `BPrecord_t` structure created earlier and then populate the record structure with our data values. Then we use the `RecordCreate` method to insert it:

```
Btrieve::StatusCode status = Btrieve::STATUS_CODE_NO_ERROR;
BPrecord_t record;
  // Get current system time and convert to microseconds
time_t now = time(0) * 1000000;
  //Convert time to Btrieve2 Timestamp format
record.timeTaken = Btrieve::UnixEpochMicrosecondsToTimestamp(now);
  //sysdata and diasdata are provided at runtime
record.systolic = sysdata;
record.diastolic = diasdata;
  //Determine the EvalCode from the systolic & diastolic
record.EvalCode = 'N'; // Default is Normal
if ((sysdata >= 120 and sysdata < 130) and (diasdata < 80))
    record.EvalCode = 'E'; //Elevated blood pressure
if ((sysdata >= 130 and sysdata < 140) or
  (diasdata >= 80 and diasdata < 90))
    record.EvalCode = 'H'; //High blood pressure
if ((sysdata >= 140 and sysdata <= 180) or
```

```
 (diasdata >= 90 and diasdata <= 120))
    record.EvalCode = 'V'; //Very high blood pressure
if ((sysdata > 180) or (diasdata > 120))
    record.EvalCode = 'C'; //Hypertensive Crisis
  // Insert the record
status = btrieveFile->RecordCreate((char*)& record, sizeof(record));
```

All these activities execute far faster using the Btrieve 2 API than a standard SQL API, delivering a level of performance in an IoT or mobile scenario that SQLite could not begin to achieve.

For modern edge data management, a database holds huge advantages over a file management system because of its ability to index and retrieve data quickly. By adding indexes to existing Zen data files, you can quickly retrieve records by a particular value or in a particular order. You can also add indexes to newly created files before any records have been inserted.

Index creation involves three easy steps:

1. Set up an index segment
2. Define index attributes
3. Add the index to the data file

Continuing with our example file, we will add an index on the time stamp portion of the record. Note that you must open a file before you can add an index to it.

```
BtrieveKeySegment btrieveKeySegment;
BtrieveIndexAttributes btrieveIndexAttributes;
  // Create a time stamp index segment on the first 8 bytes of the record
status = btrieveKeySegment.SetField(0, 8, Btrieve::DATA_TYPE_TIMESTAMP);
  // Add the segment to the Index object
status = btrieveIndexAttributes.AddKeySegment(&btrieveKeySegment);
  // Specify the nonmodifiable index attribute
status = btrieveIndexAttributes.SetModifiable(false);
  // Create the index
status = btrieveFile->IndexCreate(&btrieveIndexAttributes);
```

That's it. You've just defined an index segment as an 8-byte field starting at record offset 0. The data in those 8 bytes will be interpreted as a Btrieve TIMESTAMP. Here's what's happening above:

- The `AddKeySegment()` method appends the `btrieveKeySegment` instance to a `btrieveIndexAttributes` object to define a single-segment key.

- The `SetModifiable()` method designates the index as either modifiable (true) or nonmodifiable (false). Other attributes can be used to make the index unique or specify a particular index number.
- The `IndexCreate()` method will add index 0 to the previously opened data file associated with the `BtrieveFile` object. The index will be populated with all values currently in the file and will be updated automatically on all subsequent insert/update/delete operations.

After creating indexes, retrieving a record at speed is a straightforward process. There are methods for retrieving the first or last record according to the index definition or retrieving a particular record by comparing to a provided value. In our example, we are going to retrieve the record with the highest time stamp value, which should correspond to the record most recently inserted:

```
Btrieve::StatusCode status = Btrieve::STATUS_CODE_NO_ERROR;
BPrecord_t record;
  // Retrieve last inserted record
if (btrieveFile->RecordRetrieveLast(Btrieve::INDEX_1,
  (char*)& record, sizeof(record),
  Btrieve::LOCK_MODE_NONE) != sizeof(record))
{
  status = btrieveFile->GetLastStatusCode();
  printf("Error: BtrieveFile::RecordRetrieve():%d:%s.\n", status,
      Btrieve::StatusCodeToString(status));
}
```

Note: Record retrieval methods do not return a status code like other calls we have seen. Instead, the *size* of the retrieved record is returned by the function call. If the call does not return the size as expected, then the `GetLastStatusCode()` method can be used to find out what happened.

The last argument to the record retrieval methods provides an option to lock the record while retrieving it.

Once you have retrieved a record, you may decide to update or delete it. You cannot update/delete a record without retrieving it first. The `btrieveFile->RecordUpdate()` and `btrieveFile->RecordDelete()` methods are used for these operations.

When you are finished with a file, close it by calling the `FileClose()` method on your `btrieveClient` object:

```
status = btrieveClient->FileClose(btrieveFile));
```

Though you probably won't need it very often, Btrieve2 even provides a method for deleting a data file:

```
status = btrieveClient->FileDelete(btrieveFileName);
```

Before closing your application, you should release your session with the engine by calling the `Reset()` method.

```
status = btrieveClient->Reset();
```

So that's it for basic Create, Read, Update, Delete functionality and, as you can see, it's very simple and straightforward.

ACTIAN™

2300 Geng Rd, Suite 150, Palo Alto, CA 94303
+1 888 446 4737 [Toll Free] | +1 650 587 5500 [Tel]