STSdb 4.0
Developer's Guide

Overview

The current document represents a developer's guide for using STSdb 4.0. All of the main concepts behind the STSdb 4.0 engine are explained with the help of additional examples that demonstrate the theory in practice. We will update and complete this document in the future.

Contents

Storage Engine

The STSdb 4.0 storage engine is a WaterfallTree™ implementation. The storage engine provides two data structures – XTable and XFile. One storage engine can contain many XTable tables and many XFile files.
In STSdb 4.0 one storage engine instance is one database.

The IStorageEngine interface

The StorageEngine class implements the IStorageEngine interface, which has the following structure:

```
public interface IStorageEngine : IEnumerable<IDescriptor>, IDisposable
{
    ITable<IData, IData> OpenXTable(string name, DataType keyDataType, DataType
recordDataType);

    ITable <TKey, TRecord> OpenXTable <TKey, TRecord>(string name);

    ITable <TKey, TRecord> OpenXTablePortable<TKey, TRecord>(string name,
ITransformer<TKey, IData> keyTransformer = null, ITransformer<TRecord, IData>
recordTransformer = null);

    XFile OpenXFile(string name);

    IDescriptor this[string name] { get; }
    IDescriptor Find(long id);

    void Delete(string name);
    void Rename(string name, string newName);
    bool Exists(string name);

    int Count { get; }
    long DatabaseSize { get; }
    int CacheSize { get; }

    void Commit();
    void Close();
```

Opening a XTable

Opening an XTable instance can be done with one of the following methods:

1.
```
ITable<TKey, TRecord> OpenXTable<TKey, TRecord>(string name);
```

   The native way of using the database and also the fastest of the three. By using this method, the engine works directly with the specified TKey/TRecord and generates the appropriate persist and comparer logic for the specified types. This makes the database dependent from the user types and opening a table with different TKey/TRecord cannot be done directly. We recommended this way if the user types can be distributed with the database file or if the application uses the database as embedded.

   When the database is opening, the engine loads its scheme and scans all the assemblies in the application domain to search for the specified TKey/TRecord types for each table in it. If they are not present, an anonymous type will be generated in order for the tables to be opened. Consider the following examples:

Opening a table with the following key/record types:

```
ITable <int, Tick> table = engine.OpenXTable<int, Tick>("table");
```

Now, if we want to open the same XTable later, but we no longer have the Tick type, we can do the following:

```
using(IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
{
    engine["table"].RecordType = typeof(Tick2);

    var table = engine.OpenXTable<int, Tick2>("table");

    foreach(var item in table)
          Console.WriteLine("Key:{0},Record:{1}", item.Key.ToString(),
item.Value.ToString());

}
```

Setting the *RecordType* property of the table to type Tick2, which is identical with Tick, will make the engine generate persist and comparer logic that will allow direct usage of the type Tick2.

```
public class Tick
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
   public double Bid { get; set; }
   public double Ask { get; set; }
   public long Volume { get; set; }
   public string Provider { get; set; }
}

public class Tick2
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
   public double Bid { get; set; }
   public double Ask { get; set; }
   public long Volume { get; set; }
   public string Provider { get; set; }
}
```

If the user does not provide an equivalent type, the engine will generate an anonymous type Slots to represent the original type:

```
public interface ISlots
{
}

[Serializable]
public class Slots<TSlot0> : ISlots
{
    public TSlot0 Slot0;
}

[Serializable]
public class Slots<TSlot0, TSlot1> : ISlots
{
    public TSlot0 Slot0;
    public TSlot1 Slot1;
}

[Serializable]
public class Slots<TSlot0, TSlot1, TSlot2> : ISlots
{
    public TSlot0 Slot0;
    public TSlot1 Slot1;
    public TSlot2 Slot2;
}
…
```

In case of the Tick type, the Slots class will be generated in the following way:

```
Slots<string, DateTime, double, double, long, string>
```

And the table will be opened with the following record type:

```
Data<Slots<string, DateTime, double, double, long, string>>
```

2.
```
ITable<IData, IData> OpenXTable(string name, DataType keyDataType, DataType recordDataType);
```

The DataType class is used to describe the type and create a universal type description that is used by the engine to create the persist and comparer logic. For example, creating a DataType for the class Tick looks like this:

```
public class Tick
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
   public double Bid { get; set; }
   public double Ask { get; set; }
   public long Volume { get; set; }
   public string Provider { get; set; }

}
```

```
DataType tickType = DataType.Slotes
                (
                    DataType.String,
                    DataType.DateTime,
                    DataType.Double,
                    DataType.Double,
                    DataType.Int64,
                    DataType.String
                 );
```

As we can see, the created DataType description looks almost identical to the original definition of Tick. For primitive types the DataType is straightforward:

```
DataType primitiveType = DataType.Int32;
```

3.
```
ITable<TKey, TRecord> OpenXTablePortable<TKey, TRecord>(string name, ITransformer<TKey, IData> keyTransformer = null, ITransformer<TRecord, IData> recordTransformer = null);
```

The following method also works with the user types, but because this time they are transformed to IData instances via specially generated transformer logic, the database remains portable and the tables can be opened with TKey/TRecord types that has an equivalent DataType descriptions as the original ones.

For more sophisticated usage, the user can specify custom ITransfomer implementations.

Manipulating tables

Scheme Capabilities

The StorageEngine implementation provides scheme functionality, which can be used to obtain meta-information about the database and the tables & virtual files inside the current storage engine instance.

This capability is exposed to the user through the following methods:

```
IDescriptor this[string name] { get; }
IDescriptor Find(long id);

void Delete(string name);
void Rename(string name, string newName);
bool Exists(string name);

int Count { get; }
long DatabaseSize { get; }
int CacheSize { get; }
```

The usage of the scheme is shown in the following examples.

If we have a table:

```
ITable<int, Tick> table = engine.OpenXTable<int, Tick>("table");
```

Obtaining a table's IDescriptor can be done in the following ways:

1. Via the *this* property of the storage engine instance:

```
IDescriptor descriptor1 = engine["table"];
```

2. Via the method `IDescriptor Find(long id)`

```
IDescriptor descriptor2 = Find(id);
```

The engine also provides the following scheme methods and properties:

1. Delete a table:

```
engine.Delete("table");
```

Be aware that after deleting a table, inserting records in it is still possible. When the storage engine is disposed, the table will be deleted permanently and all data will be lost.

2. Rename a table

```
engine.Rename("table", "table_NewName");
```

3. Check if a table exists:

```
bool exists = engine.Exists("table");
```

4. Obtain the number of tables & virtual files the storage engine holds:

```
int tablesCount = engine.Count;
```

5. Obtain the size of the database:

```
long databaseSize = engine.Size;
```

XTable

The XTable class is an ITable implementation. XTable is an ordered key/value store map (table). We can open many generic tables with different key/record types in one storage engine. There are no practical limitations for the number of rows in each table.

XTable<TKey, TRecord> supported types

Supported types for TKey:

1. Primitive STSdb types – `Boolean`, `Char`, `SByte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Decimal`, `DateTime`, `String`, `byte[]`;
2. Enums;
3. Guid;
4. Classes (with public default constructor) and structures, containing public read/write properties or fields with types from [1-3];

Supported types for TRecord:

1. Primitive STSdb types – `Boolean`, `Char`, `SByte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Decimal`, `DateTime`, `String`, `byte[]`;
2. Enums;
3. Guid;
4. T[], List<T>, KeyValuePair<K, V> , Dictionary<K, V> and Nullable<T>, where T, K and V are types from [1-4];
5. Classes (with public default constructor) and structures, containing public read/write properties or fields with types from [1-5].

For example, if we have the following two types:

```
public class Key
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
}

    public class Tick
    {
        public double Bid { get; set; }
        public double Ask { get; set; }
        public long Volume { get; set; }
        public string Provider { get; set; }
    }
```

We can open different table types:

```
    ITable<long, Tick> table1 = engine.OpenXTable<long, Tick>("table1");
```

```
    ITable<DateTime, Tick> table2 = engine.OpenXTable<DateTime, Tick>("table2");
```

```
    ITable<Key, Tick> table3 = engine.OpenXTable<Key, Tick>("table3");
```

Or even:

```
    ITable<Tick, Tick> table3 = engine.OpenXTable<Tick, Tick>("table4");
```

In the case of composite keys, the engine compares sub-keys in the order in which they are declared as fields or properties. If both are present, the engine compares the fields first and then the properties, but not in the way they are declared.

Let us consider the following examples related to opening an ITable:

1.
```
    ITable<IData, IData> table1 = engine.OpenXTable<IData, IData>("table1",
DataType.Int32, DataType.String);
```

2.
```
    ITable int, string> table2 = engine.OpenXTablePortable<int, string>("table2");
```

3.
```
    ITable<int, string> table3 = engine.OpenXTable<int, string>("table3");
```

XTable <TKey, TRecord> methods

`XTable<TKey, TRecord>` implements `IXTable<TKey, TRecord>` interface which provides the following basic user methods:

```
    public class XTable<TKey, TRecord> : ITable <TKey, TRecord>
```

```
  public interface ITable
  {
  }

  public interface ITable<TKey, TRecord> : ITable, IEnumerable<KeyValuePair<TKey,TRecord>>
  {
     TRecord this[TKey key] { get; set; }

      void Replace(TKey key, TRecord record);
      void InsertOrIgnore(TKey key, TRecord record);
      void Delete(TKey key);
      void Delete(TKey fromKey, TKey toKey);
      void Clear();

      bool Exists(TKey key);
      bool TryGet(TKey key, out TRecord record);
      TRecord Find(TKey key);
      TRecord TryGetOrDefault(TKey key, TRecord defaultRecord);

      KeyValuePair<TKey, TRecord>? FindNext(TKey key);
      KeyValuePair<TKey, TRecord>? FindAfter(TKey key);
      KeyValuePair<TKey, TRecord>? FindPrev(TKey key);
      KeyValuePair<TKey, TRecord>? FindBefore(TKey key);

      IEnumerable<KeyValuePair<TKey, TRecord>> Forward();
      IEnumerable<KeyValuePair<TKey, TRecord>> Forward(TKey from, bool hasFrom, TKey to,
bool hasTo);
      IEnumerable<KeyValuePair<TKey, TRecord>> Backward();
      IEnumerable<KeyValuePair<TKey, TRecord>> Backward(TKey to, bool hasTo, TKey from,
bool hasFrom);

      KeyValuePair<TKey, TRecord> FirstRow { get; }
      KeyValuePair<TKey, TRecord> LastRow { get; }

      IDescriptor Descriptor { get; }

      long Count();
    }
```

The default XTable enumerator enumerates the table rows in ascending order (forward).

- Forward() method enumerates the table rows in ascending order.
- Backward() method enumerates the table rows in descending order.
- FindNext() returns the first row (if exists) with key greater than or equal to the specified.
- FindAfter() returns the first row (if exists) with key greater than the specified.
- FindPrev() returns the first row (if exists) with key less than or equal to the specified.
- FindBefore() returns the first row (if exists) with key less than the specified.
- FirstRow returns the row with the smallest key.
- LastRow returns the row with the greatest key.

Methods that change XTable content are asynchronous - they just put the appropriate operation in the WaterfallTree:

- this[TKey key] set;
- `Replace(TKey key, TRecord record);`
- `InsertOrIgnore(TKey key, TRecord record);`
- `Delete(TKey key);`
- `Clear().`

XTable <TKey, TRecord> basics

Each storage engine can handle many generic XTable<TKey, TRecord> tables with different types. Each TKey and TRecord can be from any primitive type or complex structure and class. However, in all cases each generic XTable<TKey, TRecord> uses a special non-generic XTable table to store its data.

```
public class XTable : ITable<IData, IData>
```

When an XTable<TKey, TRecord> is being created, it uses .NET expressions to automatically generate and compile the persist, comparer and equality comparer logic for each TKey and TRecord.

The code below shows an example of working with the non-generic XTable class:

```
// writing
using (IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
{
    ITable<IData, IData> table = engine.OpenXTable(DataType.Int32, DataType.String,
"table");

    for (int i = 0; i < 1000000; i++)
    {
        table[new Data<int>(i)] = new Data<string>(i.ToString());
    }

    engine.Commit();
}
```

```
    // reading
    using (IStorageEngine engine = STSdb.FromFile("test.stsdb4 "))
    {
        IIndex<IData, IData> table = engine.OpenXTable(DataType.Int32, DataType.String,
"table");

        foreach (var row in table) // table.Forward(), table.Backward()
        {
            Data<int> key = (Data<int>)row.Key;
            Data<string> record = (Data<string>)row.Value;

            Console.WriteLine("{0} {1}", key.Value, record.Value);
        }
    }
```

IData technology

Summary

IData is an API designed for binary serialization of user data that is integrated into STSdb 4.0. It uses reflection and .NET expressions to generate the appropriate persist and comparer logic. The main focus of the technology is blazing speed and ease of use.

Type reflection and .NET expressions

The two key components of the technology are the integrated type reflection and .NET expressions.

Using type reflection and .NET expressions, for every user type can be generated custom persist logic. Unlike most binary persist protocols, which obtain the type description for every object that is serialized, the IData engine uses type reflection only once for every user type. Without any performance dropdown, persists code for every type is compiled once on the fly without any additional effort from the user. The compiled code looks like an ordinary code that a developer could have wrote and also executes with the same speed, as if it was actually there.

IData interface and Data class

The two main structures that the IData technology uses are the IData interface and Data class:

```
[Serializable]
public class Data<T> : IData
{
    public T Value;
}
```

All tables works with Data<T> instances:

- OpenXTable<TKey, TRecord> works directly with the user types (for example, it uses Data<int> and Data<Tick> in its internal structures)
- OpenXTablePortable <TKey, TRecord> works with anonymous types (for example Data<int> and Data<Slots<string, DateTime, double, double, long, string>>)

ITransformer<T1, T2> and Transformer<T1, T2>

The interface ITransformer<T1, T2> is the basic interface which all transformer classes inherit. It has the following structure:

```
public interface ITransformer<T1, T2>
{
   T2 To(T1 value1);
   T1 From(T2 value2);
}
```

The Transformer<T1, T2> class can transform T1 instances to T2 and T2 to T1 respectively. The provided types must be compatible (i.e. their internal structure must match), otherwise transformation is impossible.

Consider the following two types:

```
public class Tick
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
   public double Bid { get; set; }
   public double Ask { get; set; }
   public long Volume { get; set; }
   public string Provider { get; set; }
}

public class Bar
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
   public double Bid { get; set; }
   public double Ask { get; set; }
   public long Volume { get; set; }
   public string Provider { get; set; }
}
```

Now to create a transformer with *Tick* and *Tick2* we can do the following:

```
Transformer<Tick, Tick2> transformer = new Transformer<Tick, Tick2>();

Tick2 tick2 = transformer.To(new Tick("EUR", DateTime.Now, 2.34, 2.56, 10000,
"ForexTrader"));
Tick tick = transformer.From(tick2);
```

Supported types

The following types are supported:

1. Primitive types – `Boolean`, `Char`, `SByte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Decimal`, `DateTime`, `String`, `byte[]`;
2. Enums;
3. Guid;
4. T[], List<T>, KeyValuePair<K, V> , Dictionary<K, V> and Nullable<T>, where T, K and V are types from [1-4];
5. Classes (with public default constructor) and structures, containing public read/write properties or fields with types from [1-5].

XFile

STSdb W4.0 supports sparse files called XFile. We can work with XFile as we are with a standard .NET stream.

```
    using (IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
    {
        XFile file = engine.OpenXFile("file");

        Random random = new Random();
        byte[] buffer = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        for (int i = 0; i < 100; i++)
        {
            long position = random.Next();

            //writes some data on random positions
            file.Seek(position, SeekOrigin.Begin);
            file.Write(buffer, 0, buffer.Length);
        }

        file.Flush();
        engine.Commit();
    }
```

XFile uses special XTable<long, byte[]> implementation and provides effective sparse file functionality. Thus, in one storage engine developers can combine using of tables and files.

Multi-threading

Storage engine instance is thread-safe. Creating (opening) XTable and XFile instances in one storage engine from different threads is thread-safe.
XTable and XFile instances are also thread-safe. Manipulating different XTable/XFile instances from different threads is thread-safe.

Transactions

STSdb 4.0 supports atomic commit at storage engine level – engine.Commit() commits all changes in all opened tables and files.

Client/Server

From the client side, creating a client connection:

```
using (IStorageEngine engine = STSdb.FromNetwork("localhost", 7182))
{
      ITable<int, string> table = engine.OpenXTable<int, string>("table");

      for (int i = 0; i < 100000; i++)
      {
          table[i] = i.ToString();
      }

      engine.Commit();
 }
```

From the server side, starting the server:

```
using (IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
{
      var server = STSdb.CreateServer(engine, 7182);

      server.Start();

      //server is ready for connections

      server.Stop();
}
```

The created server instance will listen on the specified port and receive/send data from/to the clients.