STSdb 4.0

DEVELOPER'S GUIDE

4.0.7 API

This document is a developer's guide for using STSdb 4.0.

All main concepts behind the database are explained with the help of additional examples.

We will keep updating it.

Contents

Overview

What is STSdb 4.0?

STSdb 4.0 is a NoSQL key-value store open-source database. The entire database is written in pure managed .NET language without using unsafe code. The STSdb 4.0 engine is based on WaterfallTree™ technology. WaterfallTree provides blazing performance in real-time indexing of both sequential and random keys.

At the present moment STSdb 4.0 is a universal key/value store with the following base characteristics:

- Ultra-fast embedded .NET database
- WaterfallTree™ storage engine
- Real-time indexing of random and monotonous keys
- Real-time support of multiple tables
- Wide set of natively supported .NET types
- Support of complex keys and records
- IData technology for direct work with user types without performance penalty
- Parallel vertical compressions
- Ability to work with provided user logic (custom comparers, custom persist etc.)
- Can work as an in-memory database
- Modular architecture, replaceable layers
- Base client/server functionality
- 100% managed code
- Intuitive use

About WaterfallTree™

WaterfallTree is discovered by Iliya Tronkov and Atanas Todorov in the summer of 2010 while they started working on the fourth version of STSdb. The discovered algorithm effectively solves one of the most fundamental problems in the database world – speed degradation in random keys indexing. Two years later the company STS Soft SC patented the technology. Since then the technology is theoretically justified, patented and implemented.

WaterfallTree achieves significant improvement in the speed of indexing by minimizing the number of random write operations. The created algorithm allows tangible increase in the overall speed of data systems, regardless of their size.

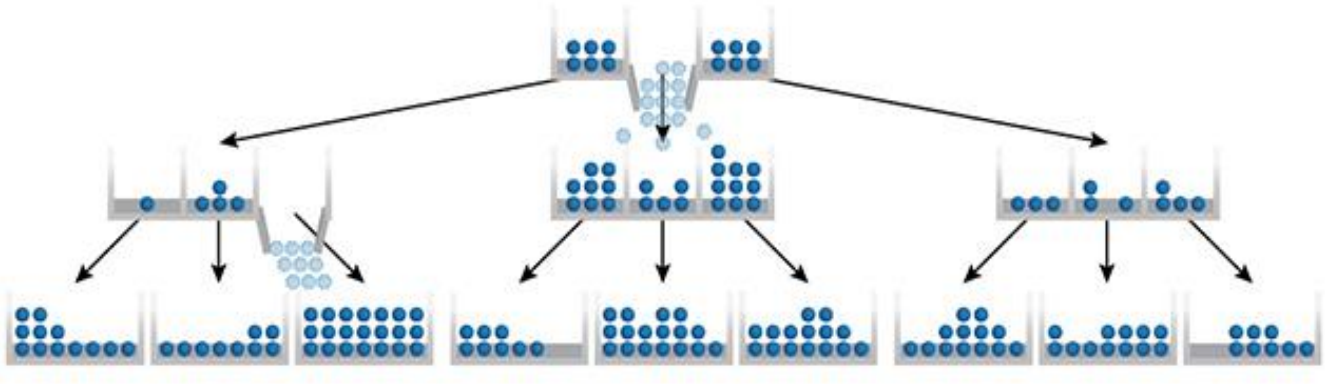
From algorithmic point of view WaterfallTree (W-tree) is a natural generalization of B^+-tree. In B^+-tree the logic operations with records and keys are executed synchronously (one by one). In WaterfallTree operations are executed asynchronously (in groups). Upon receipt of an operation in B^+-tree, it sinks down to its relevant leaf. Upon receipt of an operation in W-tree, it stops in the first non-overloaded node.

Logical operations in W-tree are accumulated within the internal nodes, grouped in branches. When a node is overloaded, an appropriate branch is chosen and its adjacent operations are poured down the tree. The operations flow down and are arranged by their keys, and the process subsides with the progress down the tree.

Unlike B-tree, the keys in W-tree can be duplicated. On its way from the root to the leaf certain key may appear more than once. These are the asynchronous operations with certain key - insert, replace, delete etc. Along their way they reach, replace and annihilate each other, until they reach the leaves. In the leaves the operations are cleared to records.

Since all movements in W-tree are massive, this allows the execution of maximum workload for one seek of the external device.

The WaterfallTree technology is naturally susceptible to parallelization. Operations along the tree are poured down independently on the different branches, thus utilizing multicore machines. STSdb 4.0 uses such parallelization in its engine and achieves insanitive performance.



Quick Start

One of the key design goals of STSdb is ease of use. Its API is simple and intuitive and provides advanced functionality out from the box.

The following examples give a practical demonstration of using STSdb with defined user key and record type.

1. Open the database and create a simple table:

```
using (IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
{
    var table = engine.OpenXTable<int, Tick>("table");

    for (int i = 0; i < 1000000; i++)
    {
        table[i] = new Tick();
    }

    engine.Commit();
}
```

2. Read the inserted records:

```
using (IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
{
    var table = engine.OpenXTable<int, Tick>("table");

    foreach (var row in table) //table.Forward(), table.Backward()
    {
        Console.WriteLine("{0} {1}", row.Key, row.Value);
    }
}
```

The record type *Tick* has the following structure:

```
public class Tick
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
   public double Bid { get; set; }
   public double Ask { get; set; }
   public int BidSize { get; set; }
   public int AskSize { get; set; }
   public string Provider { get; set; }
}
```

We can also use custom user type for the keys:

```
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 int BidSize { get; set; }
    public int AskSize { get; set; }
    public string Provider { get; set; }
}
```

And easily open a table with a composite key:

```
    var table2 = engine.OpenXTable <Key, Tick>("table2");
```

In case of composite keys, the engine compares sub-keys in the order in which they are declared as fields or properties. The engine will automatically generate the comparer logic for the keys and serialization/deserialization logic for both keys and records.

Storage Engine

The STSdb 4.0 storage engine is a WaterfallTree™ implementation. The storage engine provides two data structures – XTable and XFile. XTable is an ordered key/value table. XFile is a sparse file. One storage engine can contain many tables and many files.

In STSdb 4.0 one storage engine instance is one database.

We can open/create a database using the STSdb factory:

```
    IStorageEngine engine = STSdb.FromFile("test.stsdb4");
```

There are also several overloads:

```
    IStorageEngine engine = STSdb.FromMemory();
    IStorageEngine engine = STSdb.FromStream(stream);
    IStorageEngine engine = STSdb.FromHeap(heap);
    IStorageEngine engine = STSdb.FromNetwork(host, port);
```

The IStorageEngine interface

The returned instance from the STSdb factory is an IStorageEngine implementation. The interface provides the base functionality for managing tables and files within the database. The IStorageEngine has the following definition:

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

    ITable<TKey, TRecord> OpenXTablePortable<TKey, TRecord>(string name, DataType keyDataType,
DataType recordDataType, ITransformer<TKey, IData> keyTransformer, ITransformer<TRecord, IData>
recordTransformer);

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

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

    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; }

    int CacheSize { get; set; }

    IHeap Heap { get; }

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

Concept for the types

One database instance can work with tables from different types. Each table in the database is generic and can be used as such with provided TKey/TRecord user types. There is only one important rule: ***all used types and custom logic in the database must be available before the engine reads any record from any table***.

If the database is used as embedded and the user does not use any custom logic, the engine resolves the used types by their names and automatically generates the needed code environment. The user does not have to do anything.

But if the user has some custom logic in any of the tables (custom comparers, custom persist etc.) or if he accesses an external database file that requires external types unavailable in the application, he has to provide the entire logic and types in order to be able to work properly with the database. The custom logic and types must be provided via the table descriptors before the user reads any record from any of the database tables. This rule applies for all tables that have custom logic. In short: all custom logic and all external types must be available (or provided) before any record read from any table, but if the user uses the database as embedded and does not have any custom logic, there is nothing to worry about.

For performance reasons all tables and all virtual files in STSdb 4.0 use *one* WaterfallTree instance. This is a **main concept: *one physical device - one WaterfallTree***. Therefore **the WaterfallTree implementation** (the StorageEngine instance) is heterogeneous - it **is actually an IData->IData key-value tree** that holds instances of various IData successors!

```
public interface IData
{
}

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

That's why we need all the logic for all tables before opening any of them - because different records from different tables share the same internal nodes in the tree. A W-tree node may contain data from different tables and we need all the types and logic for it to know how to load it.

In STSdb 4.0, when the database is opened, the engine does not load any user data until some table is not opened *and* some data is not read from it. After the database is opened only the *database scheme* is loaded. This allows the users to make some preliminary preparations to the tables before start working with the data. Via the table descriptors the users can optionally provide their custom logic – comparers, persists, external types etc., to be sure that the engine will properly load the W-tree nodes with the mixes of data during operations with different tables.

NOTE: In some of the early 4.0 RC versions there was no such principle. The database worked only with anonymous data and always transformed all user records to anonymous records. In that way we kept the database independent from the user types but sacrificed part of the performance. The users also had no options to have any custom logic (for the same reason). In STSdb 4.0 we decided to provide a direct way for working with domain user types, reaching even higher speeds. We also decided to open the database and allow users to provide their custom comparers, custom persist etc. Thus, we keep the portable mode as an option and provide the new fastest direct mode. The user can now choose between the two modes for each of his tables.

XTable Modes

In STSdb 4.0 there are two basic concepts for working with tables:

1. **Working with *anonymous types* (portable mode)** – there are two methods:
   - Native way – the user and the database work with anonymous types according to their DataType descriptions. This is done by the *OpenXTablePortable* method:

```
ITable<IData, IData> OpenXTablePortable(string name, DataType keyDataType, DataType recordDataType);
```

   - Via transformers – the user works with his domain user types, while the engine transforms the data to anonymous types. This is done by the *OpenXTablePortable* methods:

```
ITable<TKey, TRecord> OpenXTablePortable<TKey, TRecord>(string name, DataType keyDataType, DataType
recordDataType, ITransformer<TKey, IData> keyTransformer, ITransformer<TRecord, IData>
recordTransformer);
```

```
ITable<TKey, TRecord> OpenXTablePortable<TKey, TRecord>(string name);
```

2. **Working with *named types* (direct mode)**:
   - The user and the database work directly with the domain user types - all runtime generated logic is created to work directly with the provided objects. This is done by the *OpenXTable<TKey, TRecord>* method:

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

Within a storage engine instance, each table can be created either in portable mode or in direct mode. A storage engine instance is portable if all tables within it are created in portable mode.

Portable mode allows the database to be independent from the application types. This mode is recommended for shared databases (used by or distributed to other applications).

Direct mode can make the database dependent from the application types. This mode is recommended for embedded databases (used only by the host application).

In the common case **the direct mode** for a table **is faster than or equal to the portable mode**. The portable mode is slightly slower only when it is used via transformers. Sometimes, in portable mode via transformers the *named* user *type* matches with his equivalent *anonymous type*. In this case there is no performance remark - the generated transformers are empty stubs and they just pass the objects.

As a general rule, once a table is being created in some of the two modes, it is not possible to open the table lately in another mode. (However, still there are advanced techniques to do that by playing with the table descriptor.)

OpenXTable<TKey, TRecord>()

Opening a table in direct mode can be done with the method:

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

This is the recommended way of using the database. By using this method the engine works directly with the specified TKey/TRecord user types and generates the appropriate persist and comparer logic for the keys and records. This can make the database dependent from the application types, but is the fastest and easiest way of using the database.

Because the engine works directly with the user types it needs them to be able to open the table - to reconstruct the environment table logic (for comparing the keys, persisting the records etc.)

NOTE: When a record from a direct user table has to be read, and this in turn requires the reading of a new W-tree node from the disk, the engine first ensures that all needed types are available to properly load the mixed W-tree node content - it scans all the assemblies in the application domain and searches for the unresolved TKey/TRecord types by their names. If the types are found the engine uses them to rebuild the expression trees with the entire environment table logic (of course, the generated logic is cached for further use). If a type is not present in the domain, the engine tries to assign equivalent anonymous types to be able to properly load the node content.

For example, if we have the following record type in the application.

```
public class Tick
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
   public double Bid { get; set; }
   public double Ask { get; set; }
   public int BidSize { get; set; }
   public int AskSize { get; set; }
   public string Provider { get; set; }
}
```

We can simply open XTable in a direct way:

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

The engine knows these two types – *long* and *Tick* and will generate the entire table logic. **The user can work with the table without providing any other functionality** (and avoiding any cast or any object transformations).

Once the user table is opening, we can simply work with it as if the table is a simple .NET generic collection, where the keys and records are from our domain types:

```
    table[key] = record;
```

```
    table.Delete(key);
```

```
    if (table.Exists(key))
    {
    }
```

```
    foreach (var kv in table)
    {
    }
```

```
    foreach (var kv in table.Forward(fromKey, toKey)) //table.Backward()
    {
    }
```

OpenXTablePortable()

```
ITable<IData, IData> OpenXTablePortable(string name, DataType keyDataType, DataType recordDataType);
```

This is the raw way for opening a portable table. With this method the user does not work with his types, instead he uses anonymous types and anonymous data.

The provided DataType descriptions in the method are used to describe the anonymous table types. The descriptions are used by the engine to prepare the table logic (comparers and persists) for the expected anonymous data. After the table is opened it will expect wrapped anonymous data matched with the exactly declared descriptions.

For example, we can create a portable table and insert records of anonymous type without engaging the table with the user type Tick:

```
    //description of the key type
    DataType keyDataType = DataType.Int64;

    //description of the record type
    DataType recordDataType = DataType.Slots(
        DataType.String,
        DataType.DateTime,
        DataType.Double,
        DataType.Double,
        DataType.Int32,
        DataType.Int32,
        DataType.String);

    //open(create) the table
    ITable<IData, IData> table = engine.OpenXTablePortable("table", keyDataType, recordDataType);

    //the key type is: long
    IData key = new Data<long>(4);

    //the record type is: Slots<string,DateTime,double,double,int,int,string>
    Tick tick = new Tick();
    var slot = new Slots<string,DateTime,double,double,int,int,string>();
    slot.Slot0 = tick.Symbol;
    slot.Slot1 = tick.Timestamp;
    slot.Slot2 = tick.Bid;
    slot.Slot3 = tick.Ask;
    slot.Slot4 = tick.BidSize;
    slot.Slot5 = tick.AskSize;
    slot.Slot6 = tick.Provider;
    IData record = new Data<Slots<string, DateTime, double, double, int, int, string>>(slot);

    //set the record for that key in the table
    table[key] = record;
```

As it is clear from the code above, the method always works with `IData` instances – for both keys and records. The IData instances are concrete Data<T> instances that wrap different anonymous data no matter keys or records. For the above table the used anonymous type for the keys is `typeof(long)`; the used anonymous type for the records is `typeof(Slots<string, DateTime, double, double, int, int, string>)`. The Slot classes here are internally available in the STSdb 4.0.

NOTE: For details about anonymous types, see the Anonymous Type topic.

Working with this method is laborious. We recommended it only if the key and record types are simple and mostly primitive. This method can be appropriate for usage in GUI clients – grids, charts etc., for uniform data access and visualizing.

OpenXTablePortable<TKey, TRecord>()

```
ITable<TKey, TRecord> OpenXTablePortable<TKey, TRecord>(string name);
```

This is the most suitable way for working with portable tables. The user works with his types and objects, while the engine transforms them to anonymous objects with automatically generated transformers.

For example, we can open a portable table and continue to work with our user type Tick:

```
    ITable<long, Tick> table = engine.OpenXTablePortable<long, Tick>("table");
```

```
    table[4] = new Tick();
```

Instead of with the Tick type, behind the scene the engine works with `typeof(Slots<string, DateTime, double, double, int, int, string>)` and with the generated transformers transforms each input tick to anonymous object with 7 slots.

Working with portable tables via transformers is identical as working with direct user tables.

For advanced users there is an OpenXTablePortable overload where custom transformers can be provided:

```
ITable<TKey, TRecord> OpenXTablePortable<TKey, TRecord>(string name, DataType keyDataType, DataType
recordDataType, ITransformer<TKey, IData> keyTransformer, ITransformer<TRecord, IData>
recordTransformer);
```

For details about how transformers work, see the Transformers topic.

Database Scheme

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

There are several IStorageEngine methods and properties:

```
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; }
int CacheSize { get; set; }
```

The usage of the scheme is shown in the following short examples. If we have a user 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 specified *id* here is a unique number internally assigned on table creation. Each table and file in STSdb 4.0 has its unique id which is never changed. Even if a table is deleted and recreated again with the same name, key and record types, it will receive a brand new (incremental) id. These IDs are used by the engine, but are visible to the users. They can be used as unique table identifiers.

The engine also provides the following scheme methods and properties:

1. Delete a table:

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

NOTE: After deleting a table, inserting records in it is still possible (while the user has the table reference). But when the storage engine is committed and disposed, the table will be deleted and it used space will be gradually recycled.

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 and virtual files the storage engine holds:

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

5. Obtain the size of the database:

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

6. Obtain the cache capacity of the engine. The returned value represents the number of tree nodes that are kept in RAM:

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

IDescriptor

Each table/file has its own IDescriptor. The descriptors are provided by the database scheme and contain meta-information about the table/file – name, id, key type, record type etc. Each descriptor also contains the entire environment logic needed for the table/file – for how to compare keys, how to store and read the records etc. In most cases this logic is auto-generated by the engine, but it can also be customized by the user.

```
public interface IDescriptor
{
    long ID { get; }
    string Name { get; }
    int StructureType { get; }

    DataType KeyDataType { get; }
    DataType RecordDataType { get; }

    Type KeyType { get; set; }
    Type RecordType { get; set; }

    IComparer<IData> KeyComparer { get; set; }
    IEqualityComparer<IData> KeyEqualityComparer { get; set; }
    IPersist<IData> KeyPersist { get; set; }
    IPersist<IData> RecordPersist { get; set; }
    IIndexerPersist<IData> KeyIndexerPersist { get; set; }
    IIndexerPersist<IData> RecordIndexerPersist { get; set; }

    DateTime CreateTime { get; }
    DateTime ModifiedTime { get; }
    DateTime AccessTime { get; }

    byte[] Tag { get; set; }
}
```

Note that every table, no matter direct or portable has key type and record type descriptions (the KeyDataType and RecordDataType properties). These descriptors are used for internal checks but are also used for anonymous type generation in case that original user types are removed or unresolved by the engine. (In the last case, based on these descriptions, the engine tries to build KeyType and RecordType anonymous types equivalent to the original ones and then tries to recreate the entire table logic (comparers and persists).

Understanding Data Types

This topic covers the basic concepts and rules needed to understand the types used in STSdb.

Anonymous Type

Definition: A .NET type in STSdb 4.0 is *anonymous*, if it is one of the following types:

1. Primitive type - `Boolean`, `Char`, `SByte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Decimal`, `DateTime`, `TimeSpan`, `String`, `byte[]`;
2. Collection type - T[], List<T>, Dictionary<K, V>, where T, K and V are types from [1-2];
3. Slot type - `Slots`<T_0>, `Slots`<T_0, T_1>, `Slots`<T_0, T_1, … >, where T_i are types from [1-3];

For example, the following .NET types are anonymous (in the terms of STSdb 4.0 concepts):

1. int, string, double, string[], List<int>, Dictionary<int, string>
2. List<Dictionary<int, string>>
3. Slots<double>, Slots<int, string>, Slots<int[], List<string>>
4. Slots<int, string, Slots<int[], List<string>>, double>
5. List<Slots<int, string>>
6. etc.

The above definition is recursive. In other words, a type is anonymous if it is primitive *or* if it is a collection or Slot type containing anonymous sub-types. The anonymous type does not contain concrete domain user classes.

Named Types

Definition: in STSdb 4.0 *named types* are all .NET types that are not *anonymous*.

For example the following type is *named*:

```
public class Tick
{
    public string Symbol { get; set; }
    public DateTime Timestamp { get; set; }
    public double Price { get; set; }
}
```

The following type is also named, because it contains sub-type that is *named*:

```
    List<Tick>
```

Named types are all types than are not *anonymous* – all types in the application domain containing named properties or fields.

Linear Types

Definition: A .NET type in STSdb 4.0 is *linear*, if it is one of the following types:

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

For example, all available Slot types in STSdb 4.0 - `Slots`<T_0>, `Slots`<T_0, T_1>, `Slots`<T_0, T_1,… > (when all T_i are *primitive types*) are also *linear types*.

Following the above definition, the *named type* Tick is also *linear*:

```
public class Tick
{
    public string Symbol { get; set; }
    public DateTime Timestamp { get; set; }
    public double Price { get; set; }
}
```

It is not primitive but it contains properties that are all primitive.

Linear types are frequently used as composed keys in STSdb 4.0 tables. The linear types can be *anonymous* and *named*. The linear type set includes all primitive types and all types that not have complex sub-types.

Supported Database Types

STSdb 4.0 supports wide set of .NET types. The supported types for a table depend on the used table mode. After defining some database terms we can provide strict definition of all supported database types:

Definition: The STSdb 4.0 *supported types* are:

In portable table mode:

1. for keys:
   - all linear anonymous types
2. for records:
   - all anonymous types

NOTE: Via default transformers portable tables can also support all types supported by the tables in direct mode. Via custom transformers the portable tables can support practically all .NET types.

In direct table mode:

1. For keys:
   1.1 all linear types
   1.2 Enums
   1.3 Guid
2. For records:
   2.1 all anonymous types
   2.2 Enums
   2.3 Guid
   2.4 Classes (with public default constructor) and structures, containing public read/write properties or fields with types from the 3 groups
   2.5 T[], List<T>, Dictionary<K, V> , KeyValuePair<K, V> and Nullable<T>, where T, K and V are types from the above 4 groups
   2.6 Classes (with public default constructor) and structures, containing public read/write properties or fields with types from the 5 groups

The definition is recursive, so the engine can handle very complex types. The supported types however do not include self-defined types – types that contain recursive definition. For example the following type Node *is not* supported by the engine (unless the user provides custom logic):

```
public class Node
{
    public long Key { get; set; }
    public string Value { get; set; }

    public Node Left { get; set; }
    public Node Right { get; set; }
}
```

Graph objects are also not supported by the database.

Data Type Description

Each anonymous type can be described with a DataType description. A DataType description is an instance, containing strict (and compact) definition of the anonymous type. **An anonymous type and its DataType description are isomorphic** - each anonymous type has one description and each description has one anonymous type. An anonymous type can be restored from its description; a description can be restored from its anonymous type.

Examples:

ANONYMOUS TYPE	DATATYPE DESCRIPTION
int	DataType.Int32
string	DataType.String
List<string>	DataType.List(DataType.String)
double[]	DataType.Array(DataType.Double)
Dictionary<int, string>	DataType.Dictionary(DataType.Int32, DataType.String)
Slots<int, string, double>	DataType.Slots(DataType.Int32, DataType.String, DataType.Double)
Slots<Slots<int, string>, double>	DataType.Slots(DataType.Slots(DataType.Int32, DataType.String), DataType.Double)
Dictionary<int[], Slots<DateTime, String>	DataType.Dictionary(DataType.Array(DataType.Int32), DataType.Slots(DataType.DateTime, DataType.String))

Note that not only the anonymous types can have DataType descriptions – *each STSdb 4.0 supported type can be described with DataType description* (not all types - only supported by the engine). Unlike the anonymous types, the user types and their descriptions are not isomorphic - if we have a description we cannot recover the original user type (unless if it is anonymous).

Each table, no matter portable or direct, stores two descriptions - for the key and for the record. The descriptions are stored in the table descriptor (table meta-info). The DataType descriptions make possible opening a table with different but equivalent user types. The descriptions are used for internal checks, for fast and compact serialization and deserialization of the user types, for easy describing and querying the sub-types etc. In short, **the DataType description is an anonymous definition of the supported database .NET type**.

When the user opens a table with anonymous types he must provide key and record DataType descriptions:

```
    ITable<IData, IData> table = engine.OpenXTablePortable("table1", DataType.Int64,
DataType.Slots(DataType.String, DataType.DateTime, DataType.Double));
```

The above line will create a non-generic portable table with key type *long* and record type *Slots<string, DateTime, double>* (a record with 3 primitive fields). Adding a record will look like:

```
    IData key = new Data<long>(7);
    IData record = new Data<Slots<string, DateTime, double>(new
Slots<string,DateTime,double>("EURUSD", DateTime.Now, 1.3615));

    table[key] = record;
```

When the user opens a table directly with his types, he does not need to provide key and record descriptions – they are internally created from the provided user types and are kept in the table's descriptor.

For example, if we create a direct table with user type Tick:

```
public class Tick
{
    public string Symbol { get; set; }
    public DateTime Timestamp { get; set; }
    public double Price { get; set; }
}
```

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

Along with the original types – *long* and *Tick*, the table's descriptor will keep the auto created DataType descriptors for the keys and for the records:

```
    DataType.Int64
```

```
    DataType.Slots(DataType.String, DataType.DateTime, DataType.Double)
```

Equivalent Types

While each anonymous type has one DataType description and can be restored from it, this is not true for the *named types* supported by the engine - **many different *named types* can have identical DataType descriptions.**

Definition: Two types are *equivalent* if they have identical DataType descriptions.

For example, the following Tick and Bar types have different definitions (and purposes), but from the engine point of view they are *equivalent* types, because they have identical DataType descriptions:

```
public class Tick
{
    public string Symbol { get; set; }
    public DateTime Timestamp { get; set; }
    public double Bid { get; set; }
    public double Ask { get; set; }
    public double BidSize { get; set; }
    public double AskSize { get; set; }
}

public class Bar
{
    public string Symbol { get; set; }
    public DateTime Timestamp { get; set; }
    public double Open { get; set; }
    public double High { get; set; }
    public double Low { get; set; }
    public double Close { get; set; }
}
```

```
    DataType.Slots(DataType.String, DataType.DateTime, DataType.Double, DataType.Double,
DataType.Double, DataType.Double);
```

The anonymous type

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

Also has the same description and therefore it is *equivalent* to the above two types.

Equivalent types can be used interchangeable within one database session.

For example if we have a database with created Tick table:

```
    using (IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
    {
        var table = engine.OpenXTable<int, Tick>("table");

        //add some records
        table[4] = new Tick();
        table[5] = new Tick();
        table[6] = new Tick();

        //commit all changes
        engine.Commit();
    }
```

With appropriate preparation we can lately open the same database file and the same table, but with type Bar for the records:

```
    using (IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
    {
        //before read any data from the database we tell the table to use the Bar type
        //for the records. The engine will use the new type to regenerate the entire
        //enviorment code logic for that table.
        engine["table"].RecordType = typeof(Bar);

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

        foreach (var kv in table)
        {
            Bar bar = kv.Value;
        }
    }
```

Note that this change must be done before the user reads any data from any table in the database. As a general rule all alter table changes and table customizations must be made immediately after the database is opened and before any record read.

XTable

XTable is a term used for the tables created by the database. Each table is an ordered key/value store map. Each storage engine can handle many XTable instances. Based on how they are created, there are two types of tables:

1. Portable tables – that work with anonymous types and data;
2. Direct user tables – that work with domain user types.

XTable implementations

I. Portable tables are two types:

1. Created by `ITable<IData, IData> OpenXTablePortable()` method (that work with anonymous type and anonymous data only):

```
public class XTablePortable : ITable<IData, IData>
{
}
```

2. Created by `ITable<TKey, TRecord> OpenXTablePortable<TKey, TRecord>()` method (that works with user types via transformers):

```
public class XTablePortable<TKey, TRecord> : ITable<TKey, TRecord>
{
}
```

NOTE: The last class is just a wrapper. It receives an ITable<IData, IData> instance on its constructor and via the provided (or generated) transformers converts each user key and record from/to anonymous data.

II. Direct user tables are created by `ITable<TKey, TRecord> OpenXTable<TKey, TRecord>()` method (and work directly with the provided user types):

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

Examples

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 int BidSize { get; set; }
    public int AskSize { get; set; }
    public string Provider { get; set; }
}
```

We can open different direct user tables:

```
    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");
```

NOTE: In case of composite keys, the engine compares sub-keys in the order in which they are declared as fields or properties. If the key type contains only public fields OR if it contains only public properties, there will be no problem with the comparing order. But if both public fields and properties are presented, the engine compares the fields first and then the properties, i.e. not in the way that they are declared (because the .NET reflection mechanism does not return them in the declared order).

We can also open different portable tables (working with transformers):

```
    ITable<long, Tick> table5 = engine.OpenXTablePortable<long, Tick>("table5");
```

```
    ITable<DateTime, Tick> table6 = engine.OpenXTablePortable<DateTime, Tick>("table6");
```

```
    ITable<Key, Tick> table7 = engine.OpenXTablePortable<Key, Tick>("table7");
```

```
    ITable<Tick, Tick> table8 = engine.OpenXTablePortable<Tick, Tick>("table8");
```

We can also open different portable tables (working with anonymous types directly):

```
    ITable<IData, IData> table9 = engine.OpenXTablePortable("table9", DataType.Int64,
DataType.Slots(DataType.Double, DataType.Double, DataType.Int32, DataType.Int32, DataType.String));
```

```
    ITable<IData, IData> table10 = engine.OpenXTablePortable("table10", DataType.DateTime,
DataType.Slots(DataType.Double, DataType.Double, DataType.Int32, DataType.Int32, DataType.String));
```

```
    ITable<IData, IData> table11 = engine.OpenXTablePortable("table11",
DataType.Slots(DataType.String, DataType.DateTime), DataType.Slots(DataType.Double, DataType.Double,
DataType.Int32, DataType.Int32, DataType.String));
```

The ITable interface

All XTable classes implement the ITable interface. The interface provides the following methods:

```
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();
}
```

In all XTable implementations the default enumerator enumerates the table rows in ascending (forward) order.

- 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:

- this[TKey key] – get & set
- Replace(TKey key, TRecord record)
- InsertOrIgnore(TKey key, TRecord record)
- Delete(TKey key)
- Delete(TKey fromKey, TKey toKey)
- Clear()

NOTE: You may note that in the ITable interface there is no Count property. Instead there is a Count() method. This is no accident. In W-tree the only slow operation is taking the number of records for a table. W-tree works with asynchronous operations which are accumulated in its internal nodes. If there are such operations, they must be flushed down the tree in order to obtain a proper record count. After all operations reach their leaf nodes a proper count value can be returned. The accumulation of operations in the nodes (which is one of the reasons for the blazing WaterfallTree speed) is actually what slows down the count calculation. That's why we changed it from property to method – to remind the users, that this is not a fast operation. In the future versions we will improve this calculation with intelligent tree monitoring system.

IData technology

IData is a summary name of a developed standalone API designed for binary serialization and comparison of user data. The IData technology is deeply integrated into the STSdb 4.0 engine. It uses reflection and .NET expressions to generate the appropriate persist and comparer logic for each user type. The main focus of the technology is speed and intuitive use.

Using .NET reflection and expression trees, a concrete persist and comparer logic can be generated for every user data. Unlike old binary serialization classes, which always query the type of every object and use lots of casts and switches, the IData engine uses type reflection once to generate the exact needed .NET expressions. Without any performance drop-off or additional user efforts, code for persisting is generated and compiled on the fly. The compiled code looks like an ordinary code that the developer could have written and is executed with the same speed (as if it was actually there). This technology is the second main reason for the STSdb 4.0 speed (the first one is the WaterfallTree technology).

IData interface

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

```
public interface IData
{
}

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

No matter portable or direct all user tables in STSdb 4.0 work with Data<T> instances, where T is *named* or *anonymous* supported type.

IData Tools

The IData technology includes suite of tools for automatic code generation:

1. `class DataComparer : IComparer<IData>` - comparer code generation for objects of any anonymous linear type
2. `class DataEqualityComparer : IEqualityComparer<IData>` - equality comparer code generation for data of any anonymous linear type
3. `class DataPersist : IPersist<IData>` - persist code generation for data of any anonymous type
4. `class DataTransformer<T> : ITransformer<T, IData>` - transformer code generation that converts user data to/from anonymous data
5. `class DataToString` – parsing/ToString code generation for data of any anonymous linear type
6. `class DataIndexerPersist : IIndexerPersist<IData>` - vertical compression code generation for sequence of IData objects
7. etc.

All these classes have their independent equivalents that can be used in any other application or project not engaged with the database:

- `class Comparer<T> : IComparer<T>` - comparer code generation for objects of any linear type T
- `class EqualityComparer<T> : IEqualityComparer<T>` - equality comparer code generation for data of any linear type T
- `class Persist<T> : IPersist<T>` - persist code generation for data of any type T
- `class Transformer<T1, T2> : ITransformer<T1, T2>` - transformer code generation that converts user data to/from other user data
- `class ValueToString<T>` – parsing/ToString code generation for data of any linear type T
- `class IndexerPersist<T> : IIndexerPersist<T>` - vertical compression code generation for sequence of T objects
- etc.

For example, for our previously defined class Tick, we can easily create a persist logic for ultra-fast serialization and deserialization of sequence of Tick objects:

```
public class Tick
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
   public double Bid { get; set; }
   public double Ask { get; set; }
   public int BidSize { get; set; }
   public int AskSize { get; set; }
   public string Provider { get; set; }
}
```

We only have to create one persist instance:

```
    Persist<Tick> persist = new Persist<Tick>();
```

Then just write (or read) the ticks directly in/from some stream:

```
    using (MemoryStream ms = new MemoryStream())
    {
        BinaryWriter writer = new BinaryWriter(ms);

        for (int i = 0; i < 10000; i++)
        {
            Tick tick = new Tick();

            persist.Write(writer, tick);
        }
    }
```

The generated logic behind the Persist<Tick> is close to the manually written logic, it looks like:

```
public void Write(BinaryWriter writer, Tick tick)
{
    writer.Write(tick.Symbol);
    writer.Write(tick.Timestamp.Ticks);
    writer.Write(tick.Bid);
    writer.Write(tick.Ask);
    writer.Write(tick.BidSize);
    writer.Write(tick.AskSize);
    writer.Write(tick.Provider);
}
```

The generated read method looks like:

```
public Tick Read(BinaryReader reader)
{
    Tick tick = new Tick();

    tick.Symbol = reader.ReadString;
    tick.Timestamp = new DateTime(reader.ReadInt64());
    tick.Bid = reader.ReadDouble();
    tick.Ask = reader.ReadDouble();
    tick.BidSize = reader.ReadInt32();
    tick.AskSize = reader.ReadInt32();
    tick.Provider = reader.ReadString();

    return tick;
}
```

Pretty.

The Persist<T> is away faster than Microsoft's BinaryFormatter and the .NET port of the Google's Protocol Buffers. The Persist<T> cannot be used for serializing graph objects or objects of recursive types, but it can be used for all supported database type – primitive types, nested types, arrays, lists, dictionaries etc. - it will generate and compile all the needed .NET expressions and will save the programmer the effort of writing formal code.

For details about how to use all IData tools in your projects see the STSLabs articles on http://stsdb.com/.

Transformers

When we use the OpenXTablePortable<TKey, TRecord>() methods, the engine transforms all input TKey and TRecord data to anonymous (portable) data. The transformation logic can be either default (via the run-time generated expressions) or custom (via the provided implementations).

To explain the transformers logic lets consider an example with user data, where the key is of type *long* and the record is our well known type *Tick*:

```
public class Tick
{
    public string Symbol { get; set; }
    public DateTime Timestamp { get; set; }
    public double Bid { get; set; }
    public double Ask { get; set; }
    public int BidSize { get; set; }
    public int AskSize { get; set; }
    public string Provider { get; set; }
}
```

Default Transformers

Let us see what happens behind the scenes when we create a portable table with default transformers:

```
    ITable<long, Tick> table = engine.OpenXTablePortable<long, Tick>("table");
```

1. The engine will generates default dataType descriptions for *long* and *Tick* types. The descriptions practically represent the table definition.

The generated description for the key type is:

```
    DataType.Int64
```

The generated description for the record type is:

```
    DataType.Slots(
        DataType.String,
        DataType.DateTime,
        DataType.Double,
        DataType.Double,
        DataType.Int32,
        DataType.Int32,
        DataType.String
    );
```

2. Based on the descriptions the engine will build anonymous types, equivalent to the provided user types.

The anonymous type for the keys matches the original user type:

```
    typeof(long)
```

The anonymous type for the records is:

```
    typeof(Slots<string, DateTime, double, double, int,  int, string>)
```

The Slots class is a generic class, internally available in the database (there are many of them):

```
public class Slots<TSlot0, TSlot1, TSlot2, TSlot3, TSlot4, TSlot5, TSlot6> : ISlots
{
    public TSlot0 Slot0;
    public TSlot1 Slot1;
    public TSlot2 Slot2;
    public TSlot3 Slot3;
    public TSlot4 Slot4;
    public TSlot5 Slot5;
    public TSlot6 Slot6;
}
```

3. After creation of the descriptions and creation of the anonymous types, the engine will generate two transformers – one for the keys and one for the records. The first transformer is responsible to convert all user keys to/from anonymous keys; the second transformer is responsible to convert all user records to/from anonymous records. The transformers are run-time created with .NET expressions, so there is no performance penalty. The generated code will look like:

```
public class KeyTransformer : ITransformer<long, IData>
{
    public IData To(long value1)
    {
        return new Data<long>(value1);
    }

    public long From(IData value2)
    {
        return ((Data<long>)value2).Value;
    }
}
```

```
public class RecordTransformer : ITransformer<Tick, IData>
{
    public IData To(Tick value1)
    {
        var slots = new Slots<string, DateTime, double, double, int, int, string>();

        slots.Slot0 = value1.Symbol;
        slots.Slot1 = value1.Timestamp;
        slots.Slot2 = value1.Bid;
        slots.Slot3 = value1.Ask;
        slots.Slot4 = value1.BidSize;
        slots.Slot5 = value1.AskSize;
        slots.Slot6 = value1.Provider;

        var data = new Data<Slots<string, DateTime, double, double, int, int, string>>(slots);

        return data;
    }

    public Tick From(IData value2)
    {
        var data = (Data<Slots<string, DateTime, double, double, int, int, string>>)value2;
        Slots<string, DateTime, double, double, int, int, string> slots = data.Value;

        Tick tick = new Tick();
        tick.Symbol = slots.Slot0;
        tick.Timestamp = slots.Slot1;
        tick.Bid = slots.Slot2;
        tick.Ask = slots.Slot3;
        tick.BidSize = slots.Slot4;
        tick.AskSize = slots.Slot5;
        tick.Provider = slots.Slot6;

        return tick;
    }
}
```

As the code shows, the RecordTransformer converts *Tick* instances to *Slots<string, DateTime, double, double, int, int, string>* instances. Then it wraps each anonymous instance in Data<Slots<string, DateTime, double, double, int, int, string>> wrapper. Similarly, the KeyTransformer converts all *long* instances to *long* instances (i.e. there is no real transformation) and wraps each of them into Data<long> wrapper.

As we said previously the Data<T> class in the database is a universal data wrapper:

```
public interface IData
{
}

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

The database uses Data<T> wrappers, because all tables in STSdb 4.0 share one WaterfallTree instance. In the Data<T> values, the real T values can be from any .NET type supported by the engine. (From the engine point of view, all user keys and records from all tables are just IData instances.)

Thus, we transform the user values to anonymous values and wrap each of them with IData instances. In the above example the real table behind our created ITable<long, Tick> table is ITable <IData, IData>, where the IData keys are actually Data<long> instances and the IData records are actually Data<Slots<string, DateTime, double, double, int, int, string>> instances.

Let's back to the OpenXTablePortable method for a moment. When a table is being created with:

```
var table = engine.OpenXTablePortable<TKey, TRecord>();
```

the engine does not keep the original key and record user types (your can check the KeyType and RecordType properties - it will not keep the Tick type), Instead it stores only the key and record *descriptions* (the DataType instances). On the bases of them lately the engine rebuilds the anonymous types. On its turn, from the anonymous types it rebuilds the transformers code expressions. With this chain: *user type -> description -> anonymous type -> transformer code*, we allow the users to open the same table with completely different user types - as long as they are compatible with the stored dataType descriptions. Thus, we make the database table independent from the user types, while keeping excellent performance and ease of use.

All these steps are happening behind the OpenXTablePortable<TKey, TRecord> method.

The above logic, however is not true when we work with the user types directly:

```
var table = engine.OpenXTable<TKey, TRecord>();
```

Here the engine works directly with the TKey & TRecord types. There are no transformers - the engine will generate a concrete compare logic, a concrete persist logic etc and along with the descriptions it will store the exact TKey and TRecord *full type names*. And when the user opens the table later, the engine will try to restore the original types from the stored names and will regenerate the entire concrete TKey & TRecord expression trees. The direct generic OpenXTable works only with the specified user types, but provides insanitive performance.

Custom Transformers

Custom transformers can be used with the following IStorageEngine method:

```
ITable<TKey, TRecord> OpenXTablePortable<TKey, TRecord>(string name, DataType keyDataType, DataType
recordDataType, ITransformer<TKey, IData> keyTransformer, ITransformer<TRecord, IData>
recordTransformer);
```

To open a table with custom transformers, along with the user types, we must manually provide 4 things:

- description of the anonymous key type
- transformer implementation that transforms each user key to/from anonymous key
- description of the anonymous record type
- transformer implementation that transforms each user record to/from anonymous record

In the default transformers, the engine generates key & record descriptions that exactly match the key & user types. This however is not mandatory – with the custom transformers the dataType descriptors can be completely different from the original user types. The important thing here is that the provided descriptions should accurately describe the anonymous instances returned by the provided transformers.

Example:

```
    DataType keyDataType = DataType.Int64;
    DataType recordDataType = DataType.Slots(DataType.String, DataType.DateTime, DataType.Decimal,
DataType.Decimal);

    ITable<long, Tick> table = engine.OpenXTablePortable<long, Tick>("table2", keyDataType,
recordDataType, null, new CustomRecordTransformer());
```

With the above code we just tell the engine the *table2* structure, the rest is covered by **the transformers** - they **are responsible for converting the input user data to anonymous objects, that match the specified dataType descriptors**. In the above example we decided to provide only a record transformer (the engine will generate a default key transformer).

```
public class CustomRecordTransformer : ITransformer<Tick, IData>
{
    public IData To(Tick value1)
    {
        Slots<string, DateTime, decimal, decimal> slots = new Slots<string, DateTime, decimal,
decimal>();

        slots.Slot0 = value1.Symbol;
        slots.Slot1 = value1.Timestamp;
        slots.Slot2 = (decimal)value1.Bid;
        slots.Slot3 = (decimal)value1.Ask;

        Data<Slots<string, DateTime, decimal, decimal>> data = new Data<Slots<string, DateTime,
decimal, decimal>>(slots);

        return data;
    }

    public Tick From(IData value2)
    {
        Data<Slots<string, DateTime, decimal, decimal>> data = (Data<Slots<string, DateTime, decimal,
decimal>>)value2;
        Slots<string, DateTime, decimal, decimal> slots = data.Value;

        Tick tick = new Tick();
        tick.Symbol = slots.Slot0;
        tick.Timestamp = slots.Slot1;
        tick.Bid = (double)slots.Slot2;
        tick.Ask = (double)slots.Slot3;

        return tick;
    }
}
```

Custom XTable logic

STSdb 4.0 supports a wide set of user types for the keys and for the records. By default the engine generates all the needed logic. But in some cases the user needs to provide his own custom logic, for example: how to compare and store the keys, how to store the records. This can be done by providing custom comparers and custom persist implementations via the table descriptor property.

To understand how to write custom table logic we have to know how the engine builds the default table logic. For example, let us consider in details what is happening when the user opens a table with key type *long* and (our well known) record type *Tick*:

```
public class Tick
{
   public string Symbol { get; set; }
   public DateTime Timestamp { get; set; }
   public double Bid { get; set; }
   public double Ask { get; set; }
   public int BidSize { get; set; }
   public int AskSize { get; set; }
   public string Provider { get; set; }
}
```

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

When the user opens (creates) a direct user table, based on the provided user types, the following things happen:

1. **DataType descriptions are generated** for the key type and for the record type. In our case for the types *long* and *Tick* the generated descriptions look like this:

```
    DataType.Int64;
```

```
    DataType.Slots(DataType.String, DataType.DateTime, DataType.Double, DataType.Double,
DataType.Double, DataType.Double);
```

These values will be permanently stored respectively in KeyDataType and RecordDataType properties of the table descriptor. (They cannot be changed during the entire table life.)

2. **Comparer and equality comparer logic is generated** for the key type:

```
    public int Compare(IData var1, IData var2)
    {
        long value1 = ((Data<long>)var1).Value;
        long value2 = ((Data<long>)var2).Value;

        if (value1 < value2)
            return -1;
        else if (value1 > value2)
            return 1;
        else
            return 0;
    }
```

The code is invoked from a created IComparer<IData> instance, automatically assigned to the KeyComparer table descriptor property.

```
public bool Equals(IData var1, IData var2)
{
    long var3 = ((Data<long>)var1).Value;
    long var4 = ((Data<long>)var2).Value;

    return var3 == var4;
}

public int GetHashCode(IData var1)
{
    long var2 = ((Data<long>)var1).Value;

    return (int)var2 ^ (int)var2 >> 32;
}
```

The code is invoked from a created IEqualityComparer<IData> instance, automatically assigned to the KeyEqualityComparer table descriptor property.

3. **Persist logic is generated** for the keys and for the records:

For type *long*:

```
public void Write(BinaryWriter writer, IData data)
{
    long dataValue = ((Data<long>)data).Value;
    writer.Write(dataValue);
}

public IData Read(BinaryReader reader)
{
    return new Data<long>(reader.ReadInt64());
}
```

The code is invoked from a created IPersist<IData> instance, automatically assigned to the KeyPersist table descriptor property

For type *Tick*:

```
public void Write(BinaryWriter writer, IData idata)
{
    Tick dataValue = ((Data<Tick>)idata).Value;

    if (dataValue.Symbol != null)
    {
        writer.Write(true);
        writer.Write(dataValue.Symbol);
    }
    else
        writer.Write(false);

    writer.Write(dataValue.Timestamp.Ticks);
    writer.Write(dataValue.Bid);
    writer.Write(dataValue.Ask);
    writer.Write(dataValue.BidSize);
    writer.Write(dataValue.AskSize);

    if (dataValue.Provider != null)
    {
        writer.Write(true);
        writer.Write(dataValue.Provider);
    }
    else
        writer.Write(false);
}

public IData Read(BinaryReader reader)
{
    var var1 = new Tick();

    var1.Symbol = reader.ReadBoolean() ? reader.ReadString() : null;
    var1.Timestamp = new DateTime(reader.ReadInt64());
    var1.Bid = reader.ReadDouble();
    var1.Ask = reader.ReadDouble();
    var1.BidSize = reader.ReadInt32();
    var1.AskSize = reader.ReadInt32();
    var1.Symbol = reader.ReadBoolean() ? reader.ReadString() : null;

    return new Data<Tick>(var1);
}
```

The code is invoked from a created IPersist<IData> instance, automatically assigned to the RecordPersist table descriptor property.

4. **IndexerPersist logic is generated** in case that key type and record type are linear:

For type *long*:

```
    public void Store(BinaryWriter writer, Func<int, IData> values, int count)
    {
        using (MemoryStream ms = new MemoryStream())
        {
            ((Int64IndexerPersist)persists[0]).Store(new BinaryWriter(ms), (idx) =>
((Data<long>)values.Invoke(idx)).Value, count);

            CountCompression.Serialize(writer, (ulong)ms.Length);
            writer.Write(ms.GetBuffer(), 0, (int)ms.Length);
        }
    }

    public void Load(BinaryReader reader, Action<int, IData> func, int count)
    {
        byte[] var1 = reader.ReadBytes((int)CountCompression.Deserialize(reader));

        using (MemoryStream var2 = new MemoryStream(var1))
        {
            ((Int64IndexerPersist)persists[0]).Load(new BinaryReader(var2), (idx, value) => {
func.Invoke(idx, new Data<long>(value)); }, count);
        }
    }
```

For type *Tick*:

```
    public void Store(BinaryWriter writer, Func<int, IData> values, int count)
    {
        Action[] actions = new Action[7];
        MemoryStream[] streams = new MemoryStream[7];

        actions[0] = () =>
        {
            streams[0] = new MemoryStream();
            ((StringIndexerPersist)persists[0]).Store(new BinaryWriter(streams[0]), (idx) =>
((Data<Tick>)values.Invoke(idx)).Value.Symbol, count);
        };

        actions[1] = () =>
        {
            streams[1] = new MemoryStream();
            ((DateTimeIndexerPersist)persists[1]).Store(new BinaryWriter(streams[1]), (idx) =>
((Data<Tick>)values.Invoke(idx)).Value.Timestamp, count);
        };

        actions[2] = () =>
        {
            streams[2] = new MemoryStream();
            ((DoubleIndexerPersist)persists[2]).Store(new BinaryWriter(streams[2]), (idx) =>
((Data<Tick>)values.Invoke(idx)).Value.Ask, count);
        };

        actions[3] = () =>
        {
            streams[3] = new MemoryStream();
            ((DoubleIndexerPersist)persists[3]).Store(new BinaryWriter(streams[3]), (idx) =>
((Data<Tick>)values.Invoke(idx)).Value.Bid, count);
        };

        actions[4] = () =>
        {
            streams[4] = new MemoryStream();
            ((Int32IndexerPersist)persists[4]).Store(new BinaryWriter(streams[4]), (idx) =>
((Data<Tick>)values.Invoke(idx)).Value.BidSize, count);
        };

        actions[5] = () =>
        {
            streams[5] = new MemoryStream();
            ((Int32IndexerPersist)persists[5]).Store(new BinaryWriter(streams[5]), (idx) =>
((Data<Tick>)values.Invoke(idx)).Value.AskSize, count);
        };

        actions[6] = () =>
        {
            streams[6] = new MemoryStream();
            ((StringIndexerPersist)persists[6]).Store(new BinaryWriter(streams[6]), (idx) =>
((Data<Tick>)values.Invoke(idx)).Value.Provider, count);
        };

        Parallel.Invoke(actions);
    }
```

```
        for (int i = 0; i < actions.Length; i++)
        {
            var stream = streams[i];
            using (stream)
            {
                CountCompression.Serialize(writer, (ulong)stream.Length);
                writer.Write(stream.GetBuffer(), 0, (int)stream.Length);
            }
        }
    }
```

```
    public void Load(BinaryReader reader, Action<int, IData> values, int count)
    {
        Data<Tick>[] array = new Data<Tick>[count];
        for (int i = 0; i < count; i++)
        {
            var item = new Data<Tick>();
            item.Value = new Tick();

            array[i] = item;
            values(i, item);
        }

        Action[] actions = new Action[7];
        byte[][] buffers = new byte[7][];

        for (int i = 0; i < 6; i++)
            buffers[i] = reader.ReadBytes((int)CountCompression.Deserialize(reader));

        actions[0] = () =>
        {
            using (MemoryStream ms = new MemoryStream(buffers[0]))
                ((IIndexerPersist<String>)persists[0]).Load(new BinaryReader(ms), (idx, value) => {
((Data<Tick>)array[idx]).Value.Symbol = value; }, count);
        };

        actions[1] = () =>
        {
            using (MemoryStream ms = new MemoryStream(buffers[1]))
                ((IIndexerPersist<DateTime>)persists[1]).Load(new BinaryReader(ms), (idx, value) => {
((Data<Tick>)array[idx]).Value.Timestamp = value; }, count);
        };

        actions[2] = () =>
        {
            using (MemoryStream ms = new MemoryStream(buffers[2]))
                ((IIndexerPersist<Double>)persists[2]).Load(new BinaryReader(ms), (idx, value) => {
((Data<Tick>)array[idx]).Value.Bid = value; }, count);
        };

        actions[3] = () =>
        {
            using (MemoryStream ms = new MemoryStream(buffers[3]))
                ((IIndexerPersist<Double>)persists[3]).Load(new BinaryReader(ms), (idx, value) => {
((Data<Tick>)array[idx]).Value.Ask = value; }, count);
        };
```

→

```
        actions[4] = () =>
        {
            using (MemoryStream ms = new MemoryStream(buffers[4]))
                ((IIndexerPersist<int>)persists[4]).Load(new BinaryReader(ms), (idx, value) => {
((Data<Tick>)array[idx]).Value.BidSize = value; }, count);
        };

        actions[5] = () =>
        {
            using (MemoryStream ms = new MemoryStream(buffers[5]))
                ((IIndexerPersist<int>)persists[5]).Load(new BinaryReader(ms), (idx, value) => {
((Data<Tick>)array[idx]).Value.AskSize = value; }, count);
        };
        actions[6] = () =>
        {
            using (MemoryStream ms = new MemoryStream(buffers[6]))
                ((IIndexerPersist<String>)persists[6]).Load(new BinaryReader(ms), (idx, value) => {
((Data<Tick>)array[idx]).Value.Provider = value; }, count);
        };

        Parallel.Invoke(actions);
    }
```

The *persists* instance in the code is a `IIndexerPersist`[] array containing already created compression engines for each of the primitive Tick members. Every IIndexerPersist implementation can compress and decompress a sequence of values from that type (for example: StringIndexerPersist – for *string* values, Int32IndexerPersist – for *int* values, etc).

As we can see, the generated code strictly follows the concrete properties in the key and record types. The engine generated the code using vertical compressions running in tasks. The default generated compression code reduces the database size and increases the performance.

NOTE: Parallel vertical compressions are used by the engine for all table records founded in the WaterfallTree leafs (for the internal nodes the engine uses the generated single record persist). Thus, we practically compress only the heaviest nodes in the tree - the leaf nodes.

If the key type and the record type are linear types the engine can generate such methods. These methods are invoked from the created IIndexerPersist<IData> instances, automatically assigned respectively to KeyIndexerPersist and RecordIndexerPersist table descriptor properties. For non-linear types the engine disables the vertical compressions (KeyIndexerPersist and RecordIndexerPersist properties are *null*), because the needed expressions code becomes too complicated for generation and the net effect would be questionable.

All these 4 steps are automatically performed when a table is opened. We shall refer to all the generated code - for the keys and for the records – compare, persist etc. as “table environment code” or just “table logic”.

Once the table logic is generated, every further work with this table or with a table with the same key and record types will not cause the engine to generate it again. While the engine is working it keeps all of the generated code in caches. And when the database is closed and reopened lately again the engine resolves the table types by their full names and gradually rebuilds this code cache.

Custom comparer and persist logic

There are cases where custom comparers or custom persist logic are needed for some reason. The engine of STSdb offers the capability to replace the entire comparer and persist logic with custom implementations.

Let's consider the following examples, demonstrating how to change the default table logic.

Suppose we have again the following table:

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

First, we have to write the needed custom logic. In the current example we will replace all of the default logic – comparer, equality comparer, persist, indexer persist.

1. Comparer:

```
public class CustomLongComparer : IComparer<IData>
{
    public int Compare(IData x, IData y)
    {
        long value1 = ((Data<long>)x).Value;
        long value2 = ((Data<long>)y).Value;

        if (value1 > value2)
            return -1;
        else if (value1 < value2)
            return 1;
        else
            return 0;
    }
}
```

2. Equality comparer:

```
public class CustomLongEqualityComparer : IEqualityComparer<IData>
{
    public bool Equals(IData x, IData y)
    {
        long value1 = ((Data<long>)x).Value;
        long value2 = ((Data<long>)y).Value;

        return value1 != value2;
    }

    public int GetHashCode(IData obj)
    {
        long value = ((Data<long>)obj).Value;

        return value.GetHashCode();
    }
}
```

3. Persists for both *long* and *Tick*:

```
public class CustomLongPersist : IPersist<IData>
{
    public void Write(BinaryWriter writer, IData item)
    {
        Data<long> data = (Data<long>)item;
        writer.Write(data.Value);
    }

    public IData Read(BinaryReader reader)
    {
        long value = reader.ReadInt64();

        return new Data<long>(value);
    }
}
```

```
public class CustomTickPersist : IPersist<IData>
{
    public void Write(BinaryWriter writer, IData item)
    {
        Tick tick = ((Data<Tick>)item).Value;

        writer.Write(tick.Symbol);
        writer.Write(tick.Timestamp.Ticks);
        writer.Write(tick.Bid);
        writer.Write(tick.Ask);
        writer.Write(tick.BidSize);
        writer.Write(tick.AskSize);
        writer.Write(tick.Provider);
    }

    public IData Read(BinaryReader reader)
    {
        Tick tick = new Tick();

        tick.Symbol = reader.ReadString();
        tick.Timestamp = new DateTime(reader.ReadInt64());
        tick.Bid = reader.ReadDouble();
        tick.Ask = reader.ReadDouble();
        tick.BidSize = reader.ReadInt32();
        tick.AskSize = reader.ReadInt32();
        tick.Provider = reader.ReadString();

        return new Data<Tick>(tick);
    }
}
```

4. Indexer persist for *long* and *Tick*:

```
public class CustomLongIndexerPersist : IIndexerPersist<IData>
{
    public void Store(BinaryWriter writer, Func<int, IData> values, int count)
    {
        for (int i = 0; i < count; i++)
        {
            Data<long> data = (Data<long>)values(i);
            long item = data.Value;

            writer.Write(item);
        }
    }

    public void Load(BinaryReader reader, Action<int, IData> values, int count)
    {
        for (int i = 0; i < count; i++)
        {
            Data<long> data = new Data<long>(reader.ReadInt64());
            values(i, data);
        }
    }
}
```

```
public class CustomTickIndexerPersist : IIndexerPersist<IData>
{
    public void Store(BinaryWriter writer, Func<int, IData> values, int count)
    {
        for (int i = 0; i < count; i++)
        {
            Tick tick = ((Data<Tick>)values(i)).Value;

            writer.Write(tick.Symbol);
            writer.Write(tick.Timestamp.Ticks);
            writer.Write(tick.Bid);
            writer.Write(tick.Ask);
            writer.Write(tick.BidSize);
            writer.Write(tick.AskSize);
            writer.Write(tick.Provider);
        }
    }

    public void Load(BinaryReader reader, Action<int, IData> values, int count)
    {
        for (int i = 0; i < count; i++)
        {
            Tick tick = new Tick();

            tick.Symbol = reader.ReadString();
            tick.Timestamp = new DateTime(reader.ReadInt64());
            tick.Bid = reader.ReadDouble();
            tick.Ask = reader.ReadDouble();
            tick.BidSize = reader.ReadInt32();
            tick.AskSize = reader.ReadInt32();
            tick.Provider = reader.ReadString();

            values(i, new Data<Tick>(tick));
        }
    }
}
```

Indexer persist classes are used by the engine to serialize a collection of IData objects. For that reason these persists inherit the IIndexerPersist<T> interface which has the following definition:

```
public interface IIndexerPersist<T> : IIndexerPersist
{
    void Store(BinaryWriter writer, Func<int, T> values, int count);
    void Load(BinaryReader reader, Action<int, T> values, int count);
}
```

The two delegates here - Func<int, T> and Action<int, T>, along with *count* parameter provide a *universal way of accessing an indexed collection* of objects, regardless of its type.

After we have implemented the custom logic, we have to tell the engine to use it. This can be done through the table descriptor (via the table instance or via the scheme), before any write/read operation from any table:

```
    using (IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
    {
        var table = engine.OpenXTable<long, Tick>("table");

        //long
        table.Descriptor.KeyComparer = new CustomLongComparer();
        table.Descriptor.KeyEqualityComparer = new CustomLongEqualityComparer();
        table.Descriptor.KeyPersist = new CustomLongPersist();
        table.Descriptor.KeyIndexerPersist = new CustomLongIndexerPersist();

        //Tick
        table.Descriptor.RecordPersist = new CustomTickPersist();
        table.Descriptor.RecordIndexerPersist = new CustomTickIndexerPersist();
    }
```

```
    using (IStorageEngine engine = STSdb.FromFile("test.stsdb4"))
    {
        //long
        engine["table"].KeyComparer = new CustomLongComparer();
        engine["table"].KeyEqualityComparer = new CustomLongEqualityComparer();
        engine["table"].KeyPersist = new CustomLongPersist();
        engine["table"].KeyIndexerPersist = new CustomLongIndexerPersist();

        //Tick
        engine["table"].RecordPersist = new CustomTickPersist();
        engine["table"].RecordIndexerPersist = new CustomTickIndexerPersist();
    }
```

XFile

STSdb 4.0 supports sparse files called XFile. We can work with XFile as we work 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);
    }

    engine.Commit();
}
```

XFile uses special XTable<long, byte[]> implementation to provide effective sparse file functionality.
The user can seek at any position – including beyond the end of the file and write to it. The engine keeps only the real pieces of data. When the user starts to read from the file, the engine fills on the fly the empty spaces with zeroes.

One storage engine can have many 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 does not support ACID per table transactions. Instead the database supports atomic commit at storage engine level – engine.Commit() commits all changes in all opened tables and files. After a database has been opened it is always in the state of its last successful commit.

Client/Server

From the client side, we can create a client connection with:

```
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, we can start the server with:

```
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.

NOTE: With client/server mode the database does not support direct user tables – all created tables are in portable mode (raw or via transformers).

Memory Usage

STSdb 4.0 provides parameters to control how much memory it will use. These parameters practically set the basic WaterfallTree properties:

- min/max children (branches) in each internal (non-leaf) node
- max operations in the root node
- min/max operations in each internal node
- min/max records in each leaf node
- number of cached nodes in the memory

The above values are controlled by the following StorageEngine fields:

```
StorageEngine engine = (StorageEngine)STSdb.FromFile("test.stsdb4");

//min/max children (branches) in each internal node
engine.INTERNAL_NODE_MIN_BRANCHES = 2;
engine.INTERNAL_NODE_MAX_BRANCHES = 5;

//max operations in the root node
engine.INTERNAL_NODE_MAX_OPERATIONS_IN_ROOT = 8 * 1024;

//min/max operations in each internal node
engine.INTERNAL_NODE_MIN_OPERATIONS = 32 * 1024;
engine.INTERNAL_NODE_MAX_OPERATIONS = 64 * 1024;

//min/max records in each leaf node
engine.LEAF_NODE_MIN_RECORDS = 8 * 1024;
engine.LEAF_NODE_MAX_RECORDS = 64 * 1024; //at least 2 x MIN_RECORDS

//number of cached nodes in memory
engine.CacheSize = 64;
```

The values in the code are defaults. The default settings assume that the stored objects will be with relatively small size (for example 50-100 bytes) and each W-tree node will contain from few thousands to tens of thousands of records. The idea is that **each W-tree node must be at least several megabytes** (6-10MB), to minimize the relative weight of seek penalty versus the recording node time.

In WaterfallTree each node has internal buffers with operations. Each operation is usually a triple - {*code, key, record*}. For example: {replace, 5, "a"}, {insertOrIgnore, 5, "b"}, {delete, 5} etc (these are the asynchronous operations with a certain key). The number of buffered operations in each internal node, except the root, is controlled by:

```
//min/max operations in each internal node
engine.INTERNAL_NODE_MIN_OPERATIONS;
engine.INTERNAL_NODE_MAX_OPERATIONS;
```

INTERNAL_NODE_MAX_OPERATIONS controls when an internal node is *overflowed* with operations. When it overflows, a part of the buffered operations are poured down to the child nodes until the number of operations became smaller than INTERNAL_NODE_MIN_OPERATIONS.

For performance reasons the number of buffered operations in the root node (when it has children) is controlled by a separate variable:

```
//max operations in the root node
engine.INTERNAL_NODE_MAX_OPERATIONS_IN_ROOT
```

There is no minimal limit for the operations in root.

The records in each leaf node are controlled by:

```
//min/max records in each leaf node
engine.LEAF_NODE_MIN_RECORDS
engine.LEAF_NODE_MAX_RECORDS
```

LEAF_NODE_MAX_RECORDS controls when a leaf node is *overflowed* with records. When a leaf node is overflowed it is splited into two leaf nodes and the new child is added to the parent node. LEAF_NODE_MIN_RECORDS controls when a leaf node is *underflowed* with records, i.e. when it contains too small number of records. When it is in an underflowed state, the leaf node has to be merged with its neighbor node.

When the number of children of an internal node exceeds the INTERNAL_NODE_MAX_BRANCHES, the internal node is splitted into two nodes, on its turn increasing the number of child nodes of its parent. INTERNAL_NODE_MAX_BRANCHES is practically the max W-tree degree. On the other side, INTERNAL_NODE_MIN_BRANCHES controls when an internal node has too small number of branches, i.e. when it has to be merged with its neighbor node.

If you change the W-tree parameters, we recommend testing the different values in advance to achieve max performance for your average record size. You can test different scenarios – small nodes and large number of cached nodes, large nodes and small number of cached nodes etc. Deviating from the default min/max ratio is also allowed.

NOTE: All split and merge actions are asynchronous and often work in parallel - they are not performed immediately when the split/merge conditions occurs. Instead, they are performed with a delay. Thus, in most cases, the W-tree can exceed the so defined boundaries, but generally it will keep the nodes around these values.

The number of all cached WaterfallTree nodes, no matter internal or leaf, is controlled by:

```
//number of cached nodes in memory
engine.CacheSize
```

It is strongly recommended for the number of cached W-Tree nodes to be greater than or equal to the maximum depth of the W-tree. Otherwise, the performance will be affected - because WaterfallTree needs to keep at least maxDepth number of nodes in the memory (one path of nodes).

Minimum and maximum W-tree depth for *recordCount* number of records contained in the database can be calculated by the formulas:

```
public int GetMinimumWTreeDepth(long recordCount)
{
    int b = INTERNAL_NODE_MAX_BRANCHES;
    int R = INTERNAL_NODE_MAX_OPERATIONS_IN_ROOT;
    int I = INTERNAL_NODE_MAX_OPERATIONS;
    int L = LEAF_NODE_MAX_RECORDS;

    double depth = Math.Log(((recordCount - R) * (b - 1) + b * I) / (L * (b - 1) + I), b) + 1;

    return (int)Math.Ceiling(depth);
}
```

```
    public int GetMaximumWTreeDepth(long recordCount)
    {
        int b = INTERNAL_NODE_MAX_BRANCHES;
        int L = LEAF_NODE_MAX_RECORDS;

        double depth = Math.Log(recordCount / L, b) + 1;

        return (int)Math.Ceiling(depth);
    }
```

The above methods are available in the StorageEngine class.

For example, for database with 1 000 000 000 records, STSdb 4.0.4 with default settings will create W-tree with max depth 7. The default engine.CacheSize is 64, which is fine (64 > 7).

The number of all cached operations (in the internal nodes) in one StorageEngine instance can reach up to:

```
    var cachedOperations = engine.INTERNAL_NODE_MAX_OPERATIONS * (engine.CacheSize - 1) +
engine.INTERNAL_NODE_MAX_OPERATIONS_IN_ROOT;
```

The number of all cached records (in leaf nodes) in one StorageEngine instance can reach up to:

```
    var cachedRecords = engine.LEAF_NODE_MAX_RECORDS * engine.CacheSize;
```

The maximum number of all cached records - no matter weither they are operations in internal nodes or pure records in leaf nodes, varies between the above two values, depending on the ratio between cached internal and leaf nodes.

NOTE: Keep in mind that W-tree nodes are heterogeneous - each node contains operations, keys and records from different tables. So we have to tune the above min/max parameters and CacheSize property according to the overall expected average key + record size.

Heap

The STSdb heap system is responsible for managing database space – space allocation, deallocation etc.

STSdb 4.0 has a replaceable heap system. The engine can work with any `IHeap` implementation:

```
    IStorageEngine engine = STSdb.FromHeap(heap);
```

The `IHeap` interface definition is simple:

```
public interface IHeap
{
    /// <summary>
    /// Register new handle. The returned handle must be always unique.
    /// </summary>
    long ObtainNewHandle();

    /// <summary>
    /// Release the allocated space behind the handle.
    /// </summary>
    void Release(long handle);

    /// <summary>
    /// Is there such handle in the heap
    /// </summary>
    bool Exists(long handle);

    /// <summary>
    /// Write data with the specified handle
    /// </summary>
    void Write(long handle, byte[] buffer, int index, int count);

    /// <summary>
    /// Read the current data behind the handle
    /// </summary>
    byte[] Read(long handle);

    /// <summary>
    /// Atomic commit ALL changes in the heap (all or nothing).
    /// </summary>
    void Commit();

    /// <summary>
    /// Close the heap and release any resources
    /// </summary>
    void Close();

    /// <summary>
    /// Small user data (usually less than one physical sector), atomic written with the Commit()
    /// </summary>
    byte[] Tag { get; set; }

    /// <summary>
    /// Total size in bytes of the user data
    /// </summary>
    long DataSize { get; }

    /// <summary>
    /// Total size in bytes of the heap.
    /// </summary>
    long HeapSize { get; }
}
```

The engine uses the heap implementation to store its tree nodes. The idea behind the interface is obvious – each `IHeap` implementation must provide functionality for writing and reading of blocks of data referenced by logical keys (handles). After sequence of writes and reads made by the engine, the heap must also provide the opportunity to commit all the changes (all or nothing). The engine expects that the heap implementation is also thread-safe.

The heap implementation can rely on the fact that the majority of blocks created by the engine are with relatively large size (> 2MB).

The current STSdb 4.0 heap system provides a default heap implementation (the `Heap` class). It can be created over any seekable stream:

```
IHeap heap = new Heap(stream);
IStorageEngine engine = STSdb.FromHeap(heap);
```

Thus, with the provided stream implementation the developer can have the needed behavior - database partitioning, backup strategies, data encodings etc. The default heap implementation does not provide any special behaviour – it is created over a simple FileStream/MemoryStream instance.

RemoteHeap

Unlike the Client/Server mode where the entire database works on the server side and the client sends commands and data to it, the RemoteHeap allows users to physically separate the StorageEngine from the Heap. Since the StorageEngine and the Heap are ideologically separated, it is not necessary for them to be close to each other. With the RemoteHeap the StorageEngine works on the client side, while the Heap layer works on the server side. The server only holds the heap, i.e. the raw data, while the entire database engine works on the client side.

For example, we can create a StorageEngine that works with remote heap:

```
using (IStorageEngine engine = STSdb.FromHeap(new RemoteHeap("host", 7183)))
{
    ITable<int, string> table = engine.OpenXTable<int, string>("table");

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

    engine.Commit();
}
```

The RemoteHeap will internally connect with the specified host and port to a listening HeapServer. On the heap server side we have to start a HeapServer instance:

```
IHeap heap = new Heap(new FileStream("test.stsdb4", FileMode.OpenOrCreate));

HeapServer server = new HeapServer(heap, 7183);
server.Start();
```

RemoteHeap and HeapServer are only a bridge between the storage engine and heap layers. HeapServer can work with any IHeap implementation, thus providing different heap topologies.

Since the database engine works closer to the client, the user can work with all supported database types, avoiding the limitations of the classic database client/server mode.

NOTE: Our tests show that with the remote heap the database is about 20% faster in write and 40 to 60% faster in read than the equivalent write and read in client/server mode. But keep in mind that the remote heap is still experimental - we release it mostly for demonstration purposes. Probably the model will be changed and developed within the database releases.

Roadmap

Version 4.0 is creativity and tradition, accumulated experience and accomplished innovation. The main priority of STSdb 4.0 was the introduction of WaterfallTree™. This version was also an actual testing of the technology. In addition, the above conditioned its realization more like a key/value core, rather than a vigorous completed database system.

The future development of STSdb will include:

- ACID transactions;
- Snapshots;
- Hot schema changes;
- Conceptual and ideological WaterfallTree improvements;
- Additional IData model improvements;
- Additional speed and size improvements;
- Even more structuring and clearing – layering of the class hierarchies, improvements of functionality, refinements of modularity, preparation for scalability.

Our idea is to develop the product STSdb from the bottom up. From a small and fast key/value innovative embedded database, it will reach to a level of *innovative autonomous systems.* The project itself over time will become a summary name of innovative solutions, constructed in a such way, so they can be used as together (in STSdb as a product), as well as separately like individual technologies (in completely different projects). The developers will be realizing the internal base relations and will be able to replace, predefine, preconfigure practically the entire system called STSdb. The database will be as externally scalable, as well as internally replaceable in its eco-system.

STSdb will become **ultra-fast functional scalable transactional database core**, which the customers can easy use in the systems, designed by them. They will be able to multiplicate and deploy it in various by scale and topology systems - from small databases to large infrastructures. STSdb will be at the same time a component, and a complete ready for use database. This is our mission.