The Original Pile Engine Demystified

Ralf Westphal, www.ralfw.de

The Original Pile Engine (OPE) as implemented by Erez Elul and Miriam Bedoni has boggled the minds of the Pile community for quite some time. Several attempts have been undertaken to understand its implementation – unfortunately without really succeeding. Recently Dirk Reuter with the help of Miriam has taken a stab at it again and written up a document trying to explain the algorithms and data structures used in the OPE (see his posting on Jan 16 2006 to the pileworks mailing list). Although this explanation is somewhat easier to understand than earlier writings it almost fully remains in the peculiar pileworks terminology.

To make Pile and its first implementation easier to understand and to pave the way for future implementations the following explanation will undertake a translation of the OPE algorithms and data structures to more usual software development terminology.

What this Paper is about – and what not

This paper is about *implementation* and not about *concept*. It tries to explain a certain software design which realizes the Pile idea. It thus does not make any statement about the quality or feasibility of Pile concepts. Rather it wants to help make future discussions about Pile concepts easier, by getting implementational details out of the way.

Pile as a whole is like a coin with two faces: One is the conceptual side of Pile, the other the implementational.

Conceptually Pile for sure is very interesting and worthy to invest research into. Where can a Pile view on the world of information processing make a difference? That´s an important question to answer. Erez Elul´s "three equations" are part of the Pile concept. The very notion of relation as opposed to data is part of the Pile concept.

But the concept must not be confused with its implementation. Because if implementational details creep into a discussion about the concept, the concept might look less attractive or infeasible. So whether a relation is implemented as an integer or whether relations are managed in buffers or trees in a Pile Engine should be of no concern to whoever is interested in the Pile concept.

So far, though, many discussions about the Pile concept sooner or later referred to Erez´ OPE, attributing it some capabilities without which Pile as a concept would not work. Thus since this implementation was so hard to understand, Pile´s success so far was limited by it.

Now, by demystifying the OPE´s inner workings this paper is trying to make the valuable Pile concept independent of any implementation. Of course, in the end Pile needs to be implemented somehow. But there can and should be more than one implementation with different qualities.

Pile implementations can differ in speed or memory consumption, but as is shown in this paper, they do not need to rely on some "magical algorithm" or the like in order to be called Pile Engines. Certainly it´s a challenge to find the right data structures and algorithms for an implementation be fast and have small memory footprint. But that´s primarily matters for software developers.

"Pile theorists" on the other hand should not be concerned too much about how a Pile Engine looks inside.As this paper will show, they can rest assured there is nothing in the OPE (or any other implementation so far) that needs to be understood first.

Pile theory is about a relational space with certain basic operations to build it and navigate through it. But whether this relational space is spanned by the OPE or another implementation, whether internally relations are integer values, or whether they are arranged in a sparsely populated 2D coordinate system, is not important for the Pile theory.

To make clear the distinction between concept and implementation of Pile, to direct research towards applications of Pile (Pile Agent) instead of implementation (Pile Engine), and to make the Pile idea independent of any particular realization in programming code: that´s the purpose of this paper.

Basic Pile Concepts

A Pile consists of a large amount of *Relations* (or Associations). Each Relation has a unique value, called a *Handle*. The OPE implements a Handle as a 32 bit value.

Also each Relation is uniquely identified by its *Parent Relations* (or Parents for short):

R=(X, Y)

R is the Relation defined by its Parents X and Y. R, X, Y each have their own values, e.g. R=23, X=99, Y=17. The value of a *Child Relation*, e.g. R, cannot be "calculated" from the values of its Parents.

The left Parent is called the Normative Parent (or Np), the right Parent is called the Associative Parent (or Ap). Normative or Associative are called the *Manner* of a Relation.

R is the Child of its Parent Relations and can be called more specifically the Normative Child (Nc) or Associative Child (Ac) depending on the Parent taken into focus (Nc of Np, Ac of Ap).

Each Relation also is attributed with a *Quality* (Q) which is encoded in the upper n bits of its value. The OPE uses n=8 bits for Q. To define a Child to be of a certain Quality it can be defined like this:

R=(X [Q] Y)

Each Relation can be Parent (either Np or Ap) to any number of Child relations, e.g.

R=(X, Y)
S=(X, Z)
T=(Z, Y)

In turn this means, each Relation can be Child (either Nc or Ac) to any number of Parents.

Only so called *Top Relations* (Tops for short) which represent entities outside a Pile do not have Parents; but of course they too are identified by a Handle.

Basic Pile Engine Functions

To navigate the relational mesh of a Pile four basic functions are necessary:

- **GetChild(X, Y)**: Retrieve the Child Relation for a particular pair of Parent Relations.
- **GetParents(R)**: Retrieve the Parent Relations for a particular Child Relation.
- **FindNormativeChildren(R)**: Find all Nc where R is Np.
- **FindAssociativeChildren(R)**: Find all Ac where R is Ap.

Since the **Find…()** functions only differ in the Manner they are traversing along the Relations they can be combined into one function: **FindChildren(R, Manner)**.

To populate a Pile with Relations one more function is needed:

- **CreateChild(X, Y, R)**: Create a new Child Relation for two particular Parent Relations. If the Parents already have a Child Relation, no new Child is created.

These four functions can be grouped into two access categories:

- Retrieving/Creating a Child: **GetChild()**, **CreateChild()**
- Traversing the Pile up and down: **GetParents()**, **FindChildren()**

In addition to the simple *FindChildren()* and *GetParents()* functions others can be defined to make traversal easier for certain access patterns, e.g. a function to find all Ap for all Nc of a Relation.[1] However, any Pile application can be build upon just the above basic functions.

OPE Data Structures and Algorithms

The OPE´s data structures are best understood with the basic Pile Engine functions in mind. Since the OPE strives for high performance its data structures are honed to serve the different access categories.

GetParents

To retrieve the Parents (X, Y) of a Child (R), OPE uses R´s Handle as an address into an array of buffers which store the Handles of its Parents (see *Parents* struct in Appendix B).

Each buffer holds 2^16=65536 *Parents* structures, i.e. its length is 524288 bytes = 0.5 MB, e.g.

```
Parents parentsBuffer[] = new Parents[65536];
```

To retrieve the Parents for a Child, OPE uses *LSB16(R)* as an index, e.g.

```
Parents rParents = parentsBuffer[LSB16(R)];
```

The pointers to all Parents buffers are kept in an *index table*[2] and *MSB16(R)* is used to find the buffer for a particular Child Handle. So the index table contains 65536 references each being 4 bytes (= 0.25 MB).

```
ParentsBuffer[] parentsBufferIndex = new ParentsBuffer[][65536];
```

The buffers are allocated as needed. Index table and buffers are of fixed size.

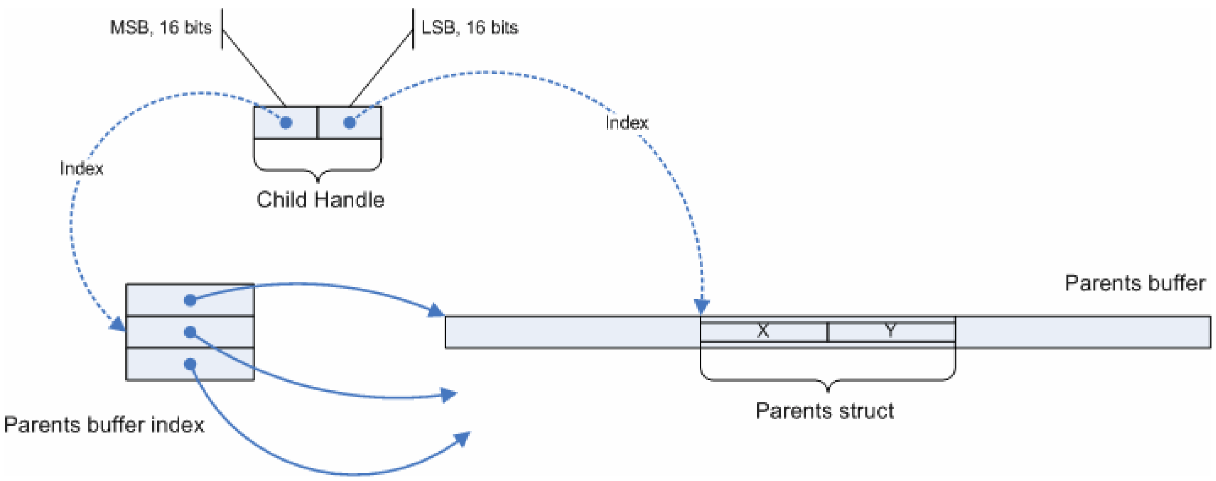

Figure 1: Finding the Parent Handles for a Child Handle

[1] This could be a useful function for fulltext search applications where neighboring characters in a text are found using a so called „valley search“-

[2] This index table is called *Con0* in the original Pile literature. It´s said to be of level 0, whereas the buffers are of level 1.

GetChild

Finding the child – if existent at all – for a pair of Parent Handles is a multistep process in the OPE involving several buffers.

Basically what needs to be achieved is the implementation of a 2-dimensional array – which unfortunately is very large and only sparsely populated. Each element in the array represents the Child Relation of two Parent Relations interpreted as the element´s coordinates, e.g.

```
uint[,] children = new uint[2^32, 2^32];

uint rHandle = children[xHandle, yHandle];
```

Since this is infeasible to implement due to memory restrictions, another data structure must be used. OPE has chosen to realize the 2D array by crafting its own memory management where Handles are used as “addresses” in the “Pile memory” in several ways.

This “Pile memory” can be said to be “paged”, since its root again is a table containing just pointers to buffers (Figure 2). So there is no contiguous space of bytes to keep the Child Handles in. This *buffer root table*[3] contains 65536 elements each consisting of two pointers to buffers, i.e. it occupies 0.5 MB in total.

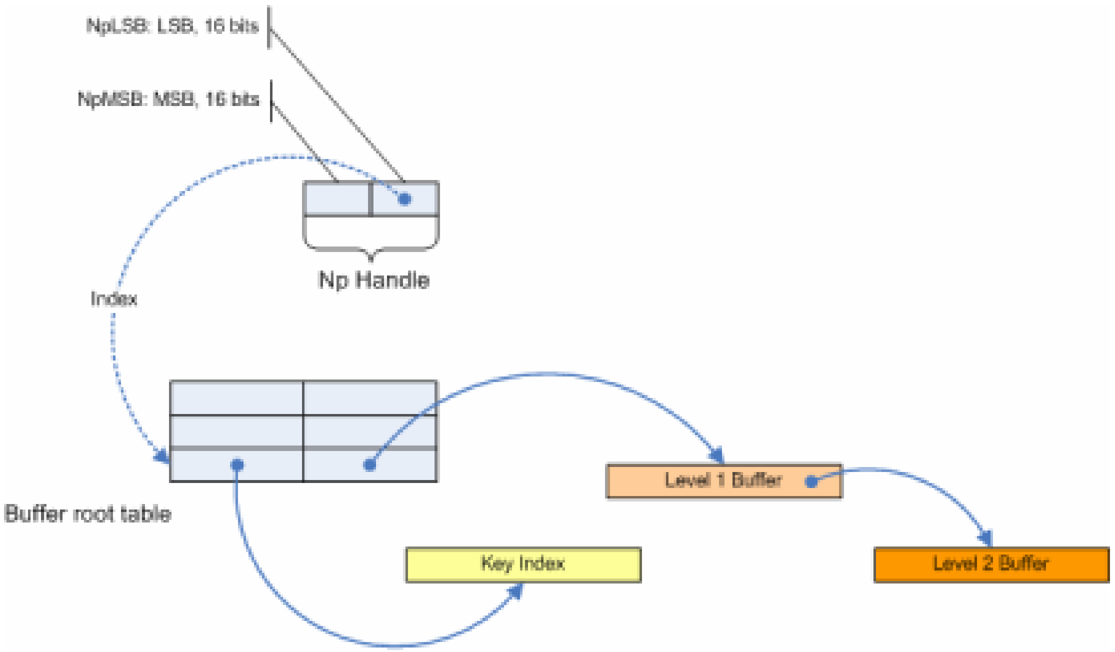

Figure 2: The “Pile memory” implements a sparsely populated 2D array of Child Handles using a cascade of buffers

Step 1: Finding a Child Relation starts with the Np Parent at the buffer root table to look up the relevant “page”. Of the Np Parent the *LSB16(xHandle)* are used as an index into the buffer root table. The address space has now been reduced from 64 bits to 48 bits.

Step 2: Of the remaining 48 bits 32 bits are taken to determine the address of the then remaining 16 bits (Figure 3). *MSB16(xHandle)* and *MSB16(yHandle)* are combined to form a 32 bit key which is

[3] The Pile literature refers to the buffer root table as *Con1*; it´s of level 0.

looked-up in the *key index*, the first buffer pointed to by the buffer root table entry just retrieved (Figure 3).

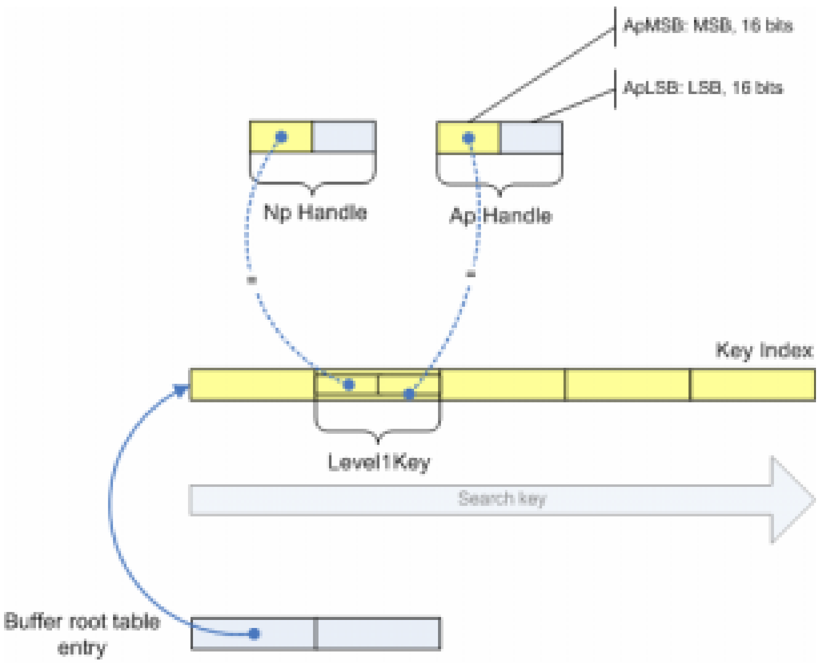

Figure 3: Look up a 32 bit key for the Child Handle in the key index. A binary search is done on the key index buffer.

The look-up is done using a binary search algorithm. The keys are stored in the key index in ascending order.

The goal of this key look-up is to determine the index (!) of the key in the key index buffer. So the search result is a value ranging from 0 to n-1 (with n being the number of keys in the key index, not its size in bytes!).

Step 3: The key's index is then used to access the entry representing the remaining 16 bits of the 64 bits address and the Child value. It is located in the *level 1 buffer* pointed to by the buffer root entry retrieved in **step 1** (Figure 4).

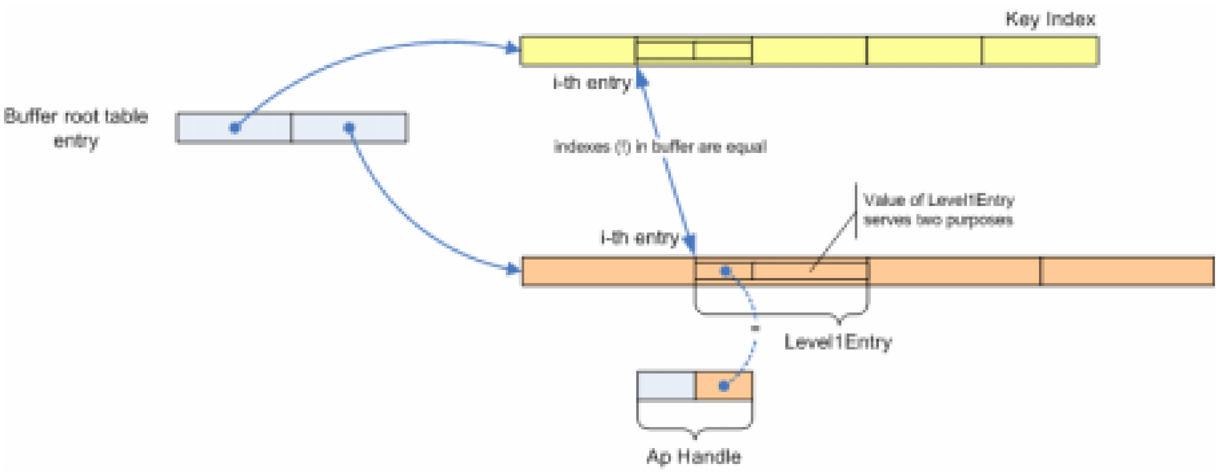

Figure 4: The index of the key found in the key index is used to access the corresponding entry for the remaining 16 bits of the 64 bits address plus the Child Handle in the level 1 buffer.

Step 4: If the LSB16 stored in the *Level1Entry* equals the *LSB16(yHandle)* then the Child Handle of (Np, Ap) is the Handle in the *Level1Entry*.

Otherwise, the *Level1Entry* needs to be checked, if it contains a Child Handle at all – or if the 4 bytes reserved for the Child Handle contain a pointer to a level 2 buffer.[4] A *Level1Entry* thus serves two purposes: it either contains a Child Handle or is a pointer to a buffer of Child Handles.

Step 5: If the *Level1Entry* contains a pointer to a level 2 buffer, because there are several *LSB16(yHandle)* values belonging to the 48 bit address (*LSB(xHandle)* & *MSB(xHandle)* & *MSB(yHandle)*) followed so far, then the *LSB16(yHandle)* is looked-up in the level 2 buffer via a binary search (Figure 5).

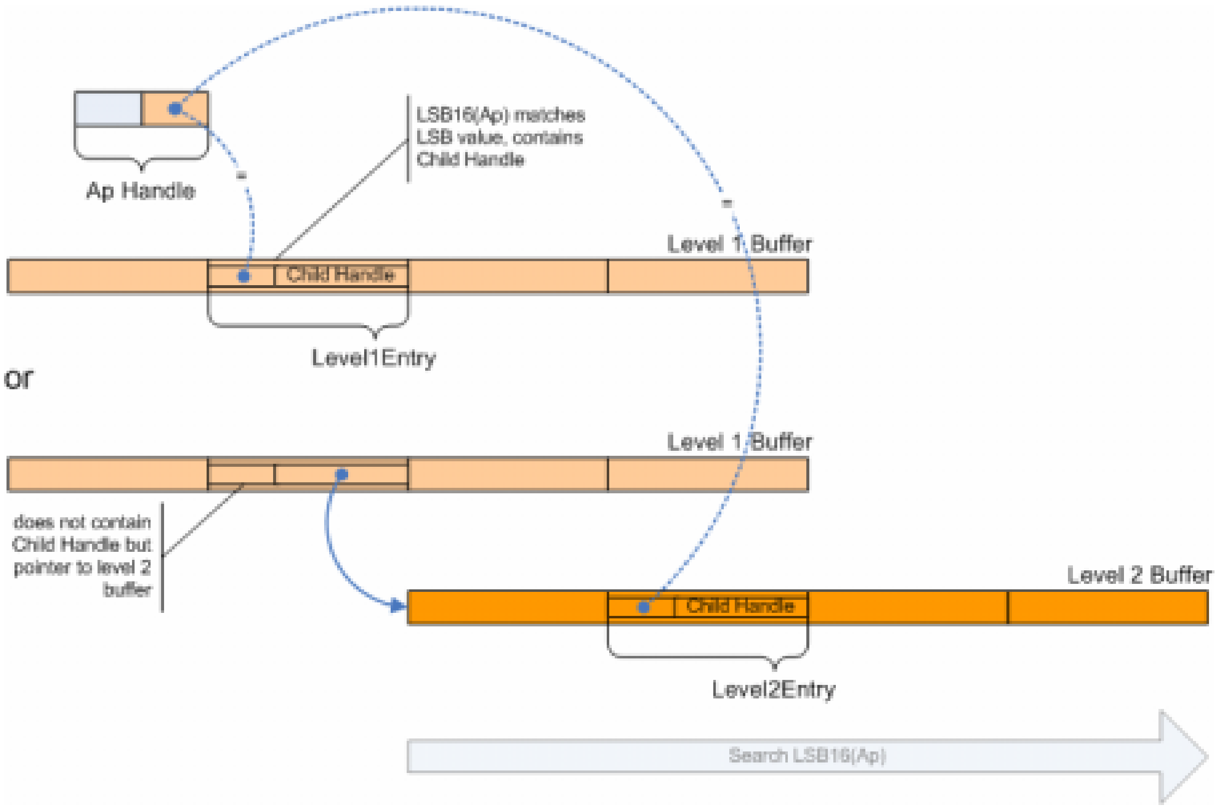

Figure 5: If the level 1 buffer entry contains a pointer to a level 2 buffer, then search the level 2 buffer for the LSB16(yHandle) entry, which contains the Child Handle

Thoughts on the OPE data structures

Why is the addressing of the buffer root table done using the LSB16 whereas the index table for *GetParents()* is addressed with the MSB16? *GetParents()* strives for few large buffers which are easy to handle, because the information looked for can be directly accessed (using the LSB16). *etChild()* however is looking up information using a binary search and thus needs to keep the entries in its buffers always sorted (see *CreateChild()*). The OPE thus favors many small buffers which can quickly be searched and easily kept sorted.

[4] How this is done in OPE is not entirely clear. The *PGM_6_bytes* structure corresponding to the *Level1Entry* struct in Appendix C bears no hint.

Why does a *Level1Entry* serve two purposes? As long as there is only one *LSB16(yHandle)* for a particular *LSB(xHandle)* & *MSB(xHandle)* & *MSB(yHandle)* combination, one indirection and binary search is saved. The Child Handle is found right in the *Level1Entry* pointed to by the *MSB(xHandle)* & *MSB(yHandle)* key´s index.

However, it seems this is just a rare case. For Piles with several thousand Relations most *LSB(xHandle)* & *MSB(xHandle)* & *MSB(yHandle)* combinations certainly point to several *LSB16(yHandle)* values. Thus it seems, the added complexity of the level 1 buffer is not really necessary. Instead the key index could contain a level 2 buffer pointer which then would always be used to store *LSB16(yHandle)* and the Child Handle.

CreateChild

Adding a Child Relation[5] to the buffer root table and its buffers essentially works like looking up a Child Handle. Wherever information is not found, it is added:

Step 1: If a buffer root entry for the *LSB16(xHandle)* is not found (not yet initialized), then initialize it with references to a small key index and level 1 buffer.

The key index, level 1 buffer as well as level 2 buffer are all dynamic buffers. They grow as needed. Whenever new data is added and there is no more room in a buffer, its size is doubled.[6]

Step 2: If the *MSB(xHandle)* & *MSB(yHandle)* key is not found in the key index, it is inserted at the expected location and all keys behind it need to be moved.[7] The same then is done in the level 1 buffer, since if the key is missing there is also the corresponding *Level1Entry* missing. The *Level1Entry* initialized with the LSB16(yHandle) and the Child Handle. No level 2 buffer needed yet.

Step 4: If the *Level1Entry* does not match the *LSB16(yHandle)* but has not yet been turned into a pointer to a level 2 buffer, the level 2 buffer is created and initialized with the current Level1Buffer entry. A reference to the level 2 buffer is placed in the *Level1Buffer*.[8] Also the current *LSB16(yHandle)* and the Child Node are added to the new level 2 buffer at the right location.

Step 5: If there was already a level 2 buffer and it does not contain the *LSB16(yHandle)* it is inserted at the expected location including the Child Handle.

In case a pair of Parents already has a Child Handle, the existing Child is returned from *GetChild()*. No new Child is added to the Pile.[9]

Of course a new Child Relation needs not only be inserted into these buffers, but also into the index table buffer of *GetParents()*.

A Second Set of Buffers

The data structures described so far are sufficient to support the *GetParents(), GetChild(), CreateChild()* functions. They are even sufficient to support *FindChildren()* if looking for Nc. However, for finding the Ac of a Parent Relation, a second set of key index and level 1/2 buffers are needed, where X and Y are exchanged. For this second set of buffers Step 1 uses *LSB16(yHandle)*, Step 2 uses *MSB(yHandle)* & *MSB(xHandle)* and Step 4/5 use *LSB16(xHandle)*.

[5] The OPE creates the Handle for a new Child Relation when adding it whereas this explanation assumes the Child Handle to be passed to *CreateChild()*. A different approach was chosen here to make understanding easier of the overall working of the OPE.
[6] This of course entails an expensive copy operation, since all data needs to be moved from the old buffer to the new one.
[7] Again this can be a pretty expensive memory operation.
[8] It´s not clear how the *Level1Entry* is flagged as being now a pointer.
[9] The OPE sports a flag signalling if a new Relation was created or an existing one returned.

This second set of buffers needs to be maintained by *CreateChild()*, but it needs to used only by *FindChildren()*.[10]

FindChildren

When looking up the children of a Parent Relation the set of buffers used depends on whether Nc or Ac are to be found. For Nc the buffers described in the *GetChild()* section are used. For Ac the second set is used. Usage of the select set of buffers in both cases is the same, though.

Step 1: Use *LSB16(parentHandle)* to find the buffer root.

Step 2: Use *MSB16(parentHandle)* to find all (!) keys of the form *MSB16(parentHandle)* & *MSB16(*)*. This is easy since the keys are sorted and the MSB16 is at the beginning of each key.

Step 3: For each key found get all Child Handles from level 1 buffer or level 2 buffer (if present).[11]

Summary

Why have the data structures been designed like this?

The data structures (and corresponding algorithms) strive for high performance and little memory consumption. Direct information access via index is used where possible, binary search is limited to as small memory areas as possible.

If performance would be better or worse if binary trees (e.g. Red-Black tree) would be used instead of the buffers, is not known yet. Memory consumption probably would go up, though, since trees require space for additional information (e.g. pointers to childnodes, red/black flag).

And why is the OPE so hard to understand?

One reason is the unusual language used in the OPE documentation, another the lack of almost any graphical description. Yet another the OPE´s own memory management in addition to it's not so easy to understand data structures.

However, since the fundamental workings of the OPE have been explained here, one thing should be clear: There is no magic in the OPE. No unique and mandatory data structure or algorithm had been invented. No "knot" is tied in the OPE. Its whole operation is indeed pretty straightforward, once the different buffers and pointers have been distangled.

It now stands to hope, that a multitude of future Pile Engines will now strive to implement the basic functions to surpass the OPE in terms of speed and memory usage in order to prove the essential Pile idea right – or wrong.

[10] The Pile literature also mentions choosing between the two sets of buffers for *Get/CreateChild()*, because one might have an advantage over the other due to the distribution of information in them. However, this is just a matter of optimization.

[11] In the Pile literature the *FindChildren()* function usually also can filter Child Handles according to their Q. This is neglected here for ease of explanation of the general operation of the OPE.

Appendix A – Helper Functions[12]

```
ushort MSB16(uint handle)
{
    return (ushort)(handle >> 16); // 16 bits, 2 bytes
}

ushort LSB16(uint handle)
{
    return (ushort)(handle & 0x0000FFFF); // 16 bits, 2 bytes
}
```

Appendix B – Finding the Parents of a Relation

```
private struct Parents
{
    public uint xHandle, yHandle;
}

private Parents[][] parentsBufferIndex = new Parents[65536][];


public Parents GetParents(uint rHandle)
{
    Parents[] parentsBuffer = parentsBufferIndex[MSB16(rHandle)];
    if (parentsBuffer != null)
        return parentsBuffer[LSB16(rHandle)];
    else
    {
        parentsBufferIndex[MSB16(rHandle)] = new Parents[65536];
        return new Parents();
    }
}
```

Appendix C – Finding the Child Relation of two Parents

```
public struct Level1Key
{
    public ushort msb0, msb1;
}

[StructLayout(LayoutKind.Explicit, Pack=1)]
public struct Level1Entry
{
    [FieldOffset(0)]
    public ushort lsb;

    // define union of two fields:
    [FieldOffset(2)]
    public uint handle;

    [FieldOffset(2)]
    public Level2Entry[] level2Buffer;
}

public struct Level2Entry
{
    public ushort lsb;
    public uint handle;
```

[12] The appendices contain C# code mimicing parts of the OPE code/data structures to make the explanations more tangible. This, though, is not to suggest the algorithms/data structures of the OPE could or should be ported 1:1 to C# (or any other language).

```
}

public struct BufferRoot
{
    public List<Level1Key> keyIndex;
    public List<Level1Entry> level1Buffer;
}

BufferRoot[] bufferRootTable = new BufferRoot[65536];
```