Space Minimizing Storage of Variable Sized Attribute Data in SECONDO

Markus Spiekermann

23.09.2008

1 Introduction

In connection with SECONDO the term *Attribute Data Type* denotes a data type which is suitable to be used as attribute in relations. Therefore it must fullfill some programming requirements explained in detail in the Programmer's Guide. In contrast to other database systems SECONDO can store data items also outside of relations, e.g. one can create a database object *pi* of type *real*. But the storage of data types in tuples is handled by different programming routines. For tuples the C++-class *Tuple* will manage the storage of byte blocks to disk and recreation of objects out of disk records vice versa.

Right from start SECONDO supports persistency of C++-classes by a coarse mechanism of storing the complete byte block of a class instance to disk. When this byte sequence is read back into memory a so called *cast*-function is applied to tell the C++ runtime system that there is an object of type X at address Y.

Unlike other programming languages like Java C++ has no ability to save object states to disk and to reload them into memory, thus this mechanism makes life easy for the implementor of SECONDO data types, since the programmer must not care about it.

However, the price paid for this convenience are limitations for the implementations of datatypes, for example no pointer variables can be used as class members, which implies that no 3rd party classes like *vector*, *string*, etc. can be utilized. Moreover the default mechanism yields in non-optimal disk space utilization, since class internal pointers and alignment constraints blow up their size. Variable sized data is supported by means of FLOBs and

DBArrays but those have a noticeable space overhead for data whose size varies only in a small range, for example between 1 and 256 bytes.

For example, in the current system class *CcInt* which represents the SECONDO *int* type has a size of 12 bytes. As a consequence big databases with many columns of standard data types such as the TPC-Benchmark data need much disk space, which results in bad performance compared to other relational database systems like *PostGres*, *Oracle*, etc. Therefore we will present a design which uses less disk space.

2 The Design of Class Tuple

To be flexible in various ways the class *Tuple* should fullfill the following requirements:

1. The number of attributes is nearly unlimited. A soft limitation is that a tuple must fit into memory otherwise a runtime error will occur.
2. Random instantiation for attribute data in constant time. However, the data block the tuple resides in will be fetched from disk, but for example creating an instance of the k-th attribute should not depend of the instantiation of other attributes.
3. Attribute data types may provide their own functions for storing the instance state into a byte sequence and vice versa. This mechanism will be called Custom-Serialization which comes in two flavours: Core- and Extension-Serialization. If no special functions are provided the Default-Serialization will be used instead.
4. The tuple is organized in two parts, the first (core) has a fixed size which is determined by the tuple type and the second (extension) has a variable length which stores all data of variable length needed either by Custom-Serialization or by FLOBs.

From now on we distinguish between 3 types of attribute data:

(a) Default-Serialization. Variable data ist managed by FLOBs and stored in the extension.

(b) Core-Serialization. A more compact data representation which has always a fixed size and is stored directly in the core part. For example an integer can always stored by a sequence of 5 bytes which encode the defined flag and a 4 byte value.

(c) Extension-Serialization. Variable sized data not managed by FLOBs but stored inside the extension part. The core part will only contain 2 byte used as offset into the extension.

For this purposes the byte block of a tuple has the following format:

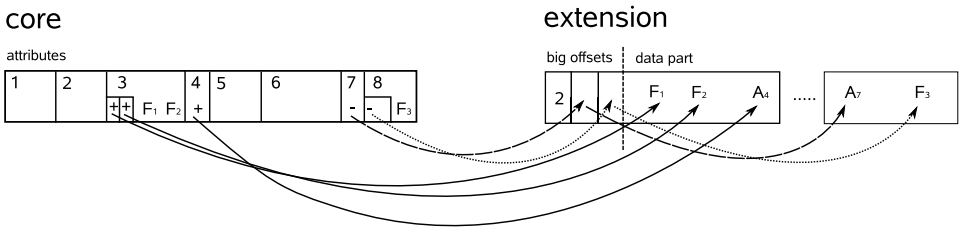

Figure 1: Block Layout for SECONDO Tuples.

The figure has a gap between core and extension. This is only done for a better distinction. In the implementation the tuple is one contigous byte block.

Trying to make this byte block as compact as possible for fairly sized tuples we only store 2 byte offsets for variable sized data. If 2 bytes are not sufficient we interpret negative numbers as index into an array of 4 byte offsets which is located at byte 2 of the extension part. This allows by default offsets up to 16k.

If this is not enough some space overhead is needed to fall back to the 4 byte offsets. However, when a tuple is saved the sizes are known and the optional 4 byte offset array will be written only if needed. The first 2 bytes stores the size n of the array, thus the start position for the data part is determined by $coresize + 2 + n * 4$.

The two byte offsets are stored for each attribute which is of type custom-variable and for each attribute which has up to k-flobs the first $2k$ bytes contain offsets for the flob data.

Note: Actually we use 4 byte offsets in general to keep the implementation a little bit easier.

In older versions, the metadata of a FLOB causes a reasonable space overhead (about 48 bytes) since it may have many special states which need their specific metadata but only few of them need to be persistent. By changing the implementation of class *FLOB* to maintain only a pointer to its metadata we can also implement a more compact disk storage of FLOBs.

3 Programming Interface for Data Types

The class *Tuple* got a new variant of open methods which additionally passes by a vector of attribute indices. This one will only create the requested attributes and was utilized to implement the operator **feedproject** which does feed and project in one step.

class *Attribute* got a new member function which indicates the serialization type. A datatype can overwrite the default by implementing its own function with another return value.

To support Custom-Serialization there are three virtual functions inherited from class *Attribute* which need to be implemented. Below the implementations for class *CcInt* are shown

```
inline virtual StorageType GetStorageType() const { return Core; }

inline virtual size_t SerializedSize() const
{
  return sizeof(int32_t) + 1;
}

inline virtual void Serialize(char* storage, size_t sz, size_t offset) const
{
  WriteVar<int32_t>(intval, storage, offset);
  WriteVar<bool>(IsDefined(), storage, offset);
}

inline virtual void Rebuild(char* state,  size_t sz )
{
  size_t offset = 0;
  ReadVar<int32_t>(intval, state, offset);
  ReadVar<bool>(del.isDefined, state, offset);
}
```

Function *SerializedSize* returns the size, a replacement for the *sizeof* function. The functions *Serialize* and *Rebuild* write bytes to a byte block or read data from a byte block. Note that the parameter *sz* is only of interest for the default implementation. For scalar data types the template functions *WriteVar* and *ReadVar* are helpful to store and restore member variables. If no custom implementation is provided the Default-Serialization as shown

below will be used:

```
enum StorageType { Default, Core, Extension };

inline virtual StorageType GetStorageType() const { return Default; }

inline virtual size_t SerializedSize() const
{
  return 0;
}

inline virtual void Rebuild(char* state,size_t sz)
{
  memcpy(this, state, sz);
}

inline virtual void Rebuild(char* state, size_t sz, ObjectCast cast)
{
  Rebuild(state,sz);
  cast(this);
}
```