

# ItemSpace Data Structures

Copyright © 2002 Boiler Bay Inc.

This document describes some of the rich variety of data structures that can easily be superimposed on a very simple lower-level data model called an ItemSpace. The superimposed data structures can take the form of sets, extensible relations, relational indexes or inversions, texts, 'binary long objects' or 'BLOB's or 'character long objects' or 'CLOB's, or an 'Entity-Attribute-Value' semantic net, among others. The lower-level ItemSpace is simply an ordered set of Items, each Item being a variable-length sequence of chars. A persistent ItemSpace implementation typically uses a B-Tree, but ItemSpaces can be implemented in a wide variety of ways.

## ***The ItemSpace Model***

In this document we will describe only the Java-language ItemSpace implementation, although in principle C or C++ would work fine. One implementation was in 8086 assembly language. Most variants of Basic would not work.

An ItemSpace is a value-ordered set of Items, each Item being a variable-length sequence of char's. For ordering purposes, the Items are compared character-by-character like String's. However, an Item is not a String, nor is it used in the same way, as the char's in an Item will often contain binary information as well as printable characters. An ItemSpace has no state other than the Item sequence, and is concurrently usable and Thread-safe. An ItemSpace supports accessor methods like next() and previous(), which retrieve the Item nearest a given Item in the ItemSpace, and it optionally supports the modifier methods insert() and delete() which operate on a given Item. Further description of the ItemSpace model will continue in the 'Items and Cursors' section.

## **About the terms 'Item' and 'ItemSpace'**

The ItemSpace data model is normally implemented using a special B-Tree that stores no data attached to each 'key'. The term 'Item' was invented because it is only the existence or non-existence of Items that carries information in an ItemSpace. Using the terms 'key' and 'KeySpace' would be misleading: a key should unlock something and yield the thing that gets unlocked. Furthermore, the term 'key' is still needed, but in a higher-level sense, when describing the Extensible Relation and Dynamic Index data structures, which are explained in the sections below. Extensible Relations and Dynamic Indexes are built out of Items, but parts of those Items can act as primary and secondary keys for the relations. The keys used in that sense do 'unlock' something: the row data of the relations.

## ***Some ItemSpace Data Structures in Brief***

Let's explore the ItemSpace data structures that have been used in practical applications as well as some further possibilities. These data structures may well seem obvious once described, but the fact is that they have rarely been seen in practice, perhaps because they are dependent on at least four basic things: a) the variable-lengths of Items in the ItemSpace, which allows multiple uses of a single ItemSpace as well as high storage efficiency; b) the efficient storage and searching of Items that have common prefixes, since many ItemSpace data structures depend on repeating long prefixes many times; c) a set of encodings for primitive values in an Item that allow Items to sort correctly, unlike the sorting of int's converted to Strings for example, and d) a sufficiently fast in-cache search capability, since repetitive local access is sometimes needed. Fast in-cache searching has become much easier, as CPU speeds generally increase on a steeper curve than disk speed.

Below is a list of some of the basic ItemSpace data structures, with brief notes on the advantages of each. These structures are discussed fully below.

## ItemSpace Data Structures

- *Extensible Relations:*
  - Null values take no space because no Item is stored with that value
  - Column cardinality (single- or multi-valued) and size are runtime alterable and extensible
  - Unused, deleted, or future columns take no space, with no limit on number or size
  - Unused, deleted, or future rows take no space, with no limit on number or size
  - Unused, deleted, or future relations take no space, with no limit on number or size
  - Come into existence at the moment of first use: no schema changes needed in advance.
  - One less disk access is required than indexes using tuple-id's, even if clustered
- *Dynamic Indexes*
  - Inversions like relational indexes may be created, deleted, and maintained on-the-fly
  - Unused, deleted, or future indexes take no space, with no limit on number or size
  - Come into existence at the moment of first use: no schema changes needed in advance.
- *Composite Keys and Column Values*
  - Are simple concatenations of self-delimiting encoded primitives, Dates, limited-length Strings, and limited-length byte and char arrays.
  - Each primitive encoding has an initial type id char., then data char's.
  - A variable number of primitives can be concatenated for each composite occurrence, so both <int,int> and <int,int,int> can be used together and compared to each other for ordering.
  - Primitive types at each position are variable, so <String,int> composites can be mixed with <String,String > composites for example.
- *Sets and Multi-valued Columns*
  - Simply groups of Items having any common prefix, such as a <databaseId,set id> or a <databaseId,relationId,primaryKey,ColumnId > composition
  - Unused, deleted, or future sets take no space, with no limit on number or size.
  - Come into existence at the moment of first use: no schema changes needed in advance.
- *Entity-Attribute-Value Model*
  - A generalization of the relational model. The 'EAV' model is cleaner, more intuitive, and extensible.
  - Uses <EntityClassId,Entity,AttributeId,Value> quads.
- *CharacterLongObjects and BinaryLongObjects (CLOB's and BLOB's)*
  - Look like embedded files
  - Blocks are numbered
  - Efficient serial (and in the future, random) access
  - Unused, deleted, empty or future CLOB's, BLOB's take no space, with no limit on number or size
  - Come into existence at the moment of first use: no schema changes needed in advance.
- *Easily Editable Text*
  - Floating-point line-numbered lines are directly insertable and deleteable.
  - Efficient access randomly or serially
  - Unused, deleted, empty or future texts or text segments take no space, with no limit on number or size
  - Come into existence at the moment of first use: no schema changes needed in advance.
- *Trees*
  - Rely on variable-length composite keys to construct paths.

- A sequence of String components is like a directory structure.
- *Future Designs*
  - Packed Items
  - Item chaining
  - Persistent Objects.

The above data structure descriptions repeat the phrase ‘with no limit on number or size’ very often, and also ‘unused, deleted, or future . . . take no space’, as well as ‘no schema changes needed in advance.’ These are common features of ItemSpace data structures.

## **Items and Cursors**

An Item existing outside the ItemSpace normally lives in a ‘Cursor’. The ItemSpace’s `insert()`, `delete()`, `next()`, and `previous()` methods among others accept Cursors containing Items as parameters. The actual class implementing the Cursor concept is called simply ‘Cu’, which is conveniently short since it occurs so often in the code.. We will refer to ‘Cursors’ in the text. A Cursor has no state other than the Item it contains. A Cursor is used by one Thread at a time.

A Cursor may seem very similar to a Java `StringBuffer`, in that it supports an overloaded `append()` method on primitive values for constructing sequences of `char`’s. However, appending an `int` to a Cursor creates a binary-encoded sequence rather than a sequence of printable chars. Each value appended to a Cursor creates a self-delimiting subsequence of the Item’s characters called a *component*, and it can be compared to other components in the same location in other Cursors in such a way that the ordering is appropriate to the types of the components. In other words, if two `int`’s are compared directly, the result will not always be the same as if they are appended to two `StringBuffers` that are then converted into `Strings` and compared. On the other hand, when two `int`’s are appended to two empty Cursors or to two Cursors containing the same Items, then the Cursors are compared, the results agree with the comparison of the `int`’s themselves. This comparison-consistency comes from the type-specific encoding of the data into the `char`’s of a component, as described in the next section. Another difference with `StringBuffer` is the way `toString()` works: Cu’s produce something useful primarily for debugging purposes.

## **Storing and Retrieving Data**

Here is how to store and retrieve an unknown value or ‘data’ given some ‘key’ or identifying value. The key and data can each be any sequence of components. Append the key and value components to a Cursor, and invoke `insert()` on the Cursor to store them into the ItemSpace. To retrieve the value, place the key into a Cursor and invoke `next()` on the Cursor. The data, if any, is now at the end of the Cursor after the key. This is a slight oversimplification, because there is an additional parameter to `next()` called the ‘protected prefix length’ or usually just ‘pl’, which identifies the number of characters in the Cursor that are to be kept the same after `next()` is invoked. The full signatures of these most important methods of the ItemSpace class are:

```
public void insert(Cu cu) throws IOException;

public void delete(Cu cu) throws IOException;

public boolean next(Cu cu,int pl) throws IOException;

public boolean previous(Cu cu,int pl) throws IOException;
```

Rather than modifying the protected prefix, `next()` returns `false` and the Cursor’s value is preserved. Therefore, the `boolean` return value indicates the existence or lack of a value for the given key. To get the proper value of ‘pl’, normally the original length of the Cursor is used. In a while loop, for retrieving

multiple data values for the same key, the `pl` is the original length of the `Cursor`, and the return value from `next()` breaks the loop when `false`. Examples of these usages are given below.

## Primitive Data Encodings

Each encoded component begins with an internal initial code char or 'idChar' which identifies both the type of the component and sometimes part or all of the value. It is not necessary for an application to use these idChar's directly. The `Cu.append()` method is overloaded to handle `long`, `float`, `double`, `boolean`, `Date`, `String`, `char[]`, and `byte[]` types. The two array types are for short sequences of `chars` or `bytes`: unlimited length sequences use the CLOB's or BLOB's and multiple Items as described below. The actual encodings of the components are in principle not important to an application programmer. Nevertheless, here are some descriptions of the component encodings for the primitive types.

- o `boolean` is stored as one of two idChar's, with no data `chars` following..
- o `long` is appended with an idChar followed by a four-char big-endian binary value with most-significant bit reversed. The most-significant bit of a `long` is reversed so that it sorts correctly. Longs near zero are stored in one total char by using a range of idChar's instead of one idChar as in most of the other primitive encodings.
- o `float` and `double` are stored as one idChar followed by a big-endian binary sequence of two or four `chars` with the all the non-sign bits reversed if the number is negative, and with the sign bit reversed, to get the correct sorting behavior.
- o `Date` is stored somewhat like a `long`, as a number of milliseconds since the epoch, but it has a different idChar and does not use a range of idChars for values near zero.
- o `String` appended to a `Cursor` contains an initial idChar and a zero `char` at the end to delimit it, and binary escape sequences embedded in it to allow any character to occur inside. The escapes that are used for `Strings` are: 0 becomes 1 followed by 2, and 1 becomes 1 followed by 3.
- o `char[]` is stored as an idChar followed by a `char` length and the `chars` in the array starting at the lowest-indexed `char`. Relatively short arrays can be handled.
- o `byte[]` is stored as an idChar followed by a `char` length and the `bytes` in the array starting at the lowest-indexed `char`. Relatively short arrays can be handled.

These encodings are not the only possible ones but have been found good in practice. For backward compatability reasons, these encodings will not be changed without a very compelling reason, but they could be extended in the future. If the component cannot be appended because it would make the `Cursor` too long for the `Cursor`'s implementation, a `CursorLengthException` is thrown, and the actual length does not change.

As a practical matter, these encodings can be very efficient when used with an `ItemSpace` that further subjects `Items` to UTF-8 and then `ZLib` compression before being stored on disk, yielding a one- or two- followed by a three-fold compaction respectively. Because `long`'s can be stored efficiently in this way even when there are many leading zeroes, and especially efficiently for the values near zero, `Cu.append()` for `byte`, `short`, and `int` were omitted. Allowing these other integral types is dangerous for two reasons. One is that it is easy to get the wrong type when using overloading – for example, a `short` can be changed to an `int` or an `int` to a `long` during normal arithmetic promotion or during normal program maintenance. The other is that it is difficult to predict how large a particular number may need to be when designing the schema for a database – it is much better to be conservative, but the tendency is to be stingy when it is perceived that space will be wasted. Using a `long` everywhere may seem extravagant, but there is really no significant space or time overhead given good data compression.

Even without any UTF-8 or ZLib compression, simple common-prefix compression will remove the `idChar` and some of the subsequent 'data' chars in common cases.

When a component is to be extracted from a `Cursor`, the offset of the component is needed, and possibly the type. The actual type of the component can be retrieved using `Cu.typeAt(int offset)`, which returns a distinct `int` for each of the possible component types. The types are static final `int`'s such as `Cu.STRING_TYPE`. These types are `int`'s and are not the same as the `idChar`'s at the beginning of the encoded components. There are also methods like 'String `stringAt(int offset)`' for each of the types that `append()` accepts. Using `stringAt()` with the offset of a component that is not a `String` will result in a `CursorDataTypeException`. There is thus a form of type safety, although it occurs at runtime and is not guaranteed.

The offset of a component can be determined from the preceding component's offset using a method like 'int `skipString(int offset)`' or 'int `skipComponent(int offset)`' if the component type is not known. Most often, the offset of a component will be known from the context, and the entire `Item` need not be parsed from the beginning to find it. The offset of the component will frequently be the same as the prefix length 'pl' that was used in a preceding `next()` invocation. If the offset is too large, `CursorIndexOutOfBoundsException` is thrown. In principle it is possible for an erroneous offset to point into the middle of a component, but this happens very rarely in practice. A bad offset will frequently cause either a `CursorDataTypeException` or `CursorIndexOutOfBoundsException` to be thrown.

There is a way to bypass the strict typing of the components in a `Cursor`, say, for debugging purposes, to understand the component encodings, to create extremely efficient storage formats, or to work with 'PlainText' Items which are unencoded strings. `Cu.charAt(int)` will return the char at any offset in a `Cu`. `Cu.append(char)` appends exactly one char without any type identifier char preceding it. There are also several 'PlainText' related methods in `Cu` which each have 'PlainText' in the method name and which can be used for working with raw character data in a `Cu`. Working with such raw data will not be discussed further here.

It is possible in many situations to use a `Cursor` without dealing with the types of the components inside. One can often stay within the `Cursor-and-ItemSpace` world entirely, avoiding, for example, the construction of `Strings` by `stringAt(int offset)`, which can be a performance limit. There are methods for appending one `Cursor` onto another and for extracting a suffix from one `Cursor` and appending it onto another. Also, generic programming can use `Object Cu.componentAt(int offset)`, `int Cu.skipComponentAt(int offset)`, and `Cu.append(Object)`. For such generic programming, primitives are represented by their corresponding wrappers such as `Long`.

## The Item Length Limitation

Every `ItemSpace` and `Cursor` has an upper limit on the size of an `Item` which it can store, but it is normally much greater than that needed for any practical purpose. For example, the Infinity Database Engine has a 1666 character limit. Almost all other databases have shorter key length limits. This limit is fine given the following reasoning. Items can loosely be broken down into a primitive or short composite 'name or identifier' part and a primitive or short composite 'data' part, and the combination of the two has a natural tendency to be relatively small. Typical names and identifiers of real-world things range from, say, 1 to 100 characters, and the data part will normally be relatively small - typically a primitive, a small number of primitives in a composite, or a text line, for example.

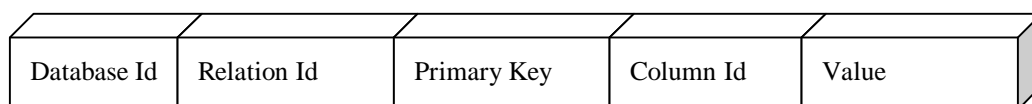
When long Items are expected to arise, a proper data structure design can almost always avoid the problem. Long pieces of data can nearly always be broken up into multiple Items in some way and stored in an `ItemSpace`, sometimes even providing a higher degree of editability, extensibility, and flexibility as a result. As an example, a long text can be broken at line boundaries, and each line can be stored with a line number prefix, or long sequences of chars can be broken into blocks and numbered (see the "CLOB's, BLOB's and Easily Editable Text" section below.) In real-world use, the Item-length limitation is not a

problem. Furthermore, real-world applications require predictable memory use for every operation, so a very long String that comes entirely into memory for each access (this actually happens in some DBMS's) makes application stability dependent on the data. High-reliability programming requires that instantaneous maximum memory use be limited to avoid OutOfMemoryErrors, so data must be handled in some kind of finite-length chunks.

Components often have some kind of externally defined size limitation. Although these days we are being far more liberal in our size limitations, the limitations nevertheless exist. For example, file paths are limited in virtually all modern operating systems to the order of a hundred chars. A length budget can guarantee that total Item length will not be exceeded when the external limits are known. If there are no external limits, CLOB's or the like can always be used.

## Extensible Relations

An Extensible Relation is constructed of nothing more than a set of Items having a common prefix that identifies the database instance and relation, concatenated with a possible composite primary key, then a column identifier, and finally a possibly composite value:



This storage organization would be very inefficient except that an ItemSpace engine can do common-prefix compression to avoid repeating the database Id, relation Id and primary keys or even column Id's in adjacent Items. Common-prefix compression operates at the raw Item level and is unaware of the component boundaries, so if adjacent Items share any char's at the beginning, the shared char's will not be stored. An example of two stored relation rows might look like the table below, with one Item per table row. The cells to the left are blank when there is a common prefix. The fixed-length of each table row below does not indicate the corresponding Items are the same length. Each relation row comes out vertical in this table. The fact that rows of the relation come out vertically in the table below could be called a 'meta twist.'

Database Id	Relation Id	Primary Key	Column Id	Column Value
Chem	Students	Jenkins, Waldo	Student Id	3451
			Class	Inorganic 101
				Proteins 201
		...son, Paul	Student Id	2665
			Class	Organic 402
				Lipid Synthesis 202

Note that the '...son, Paul' starts with a '..' which is intended to indicate that the engine has compressed away those characters. Also note that the 'Class' attribute can be multi-valued.

## Relation Extensibility

With the given structure, there is no limit to the number of databases, the number of relations in a database, the number of rows in a relation, the number of columns, or the number of values in a table cell that can be stored in an ItemSpace. Each component of the Item has a (virtually) unlimited space of combinations. A database, row, or cell that is empty simply has no corresponding Items, and takes no space.

No anticipation of future extensions of an extensible relation is needed in most cases. For example, new columns can be added to a table of an existing database at runtime without requiring the equivalent of a SQL 'MODIFY TABLE *table* ADD COLUMN ...'. Sometimes that SQL statement is not even available, and a database dump/drop/create/restore is needed. The null values for the new column do not need to be stored in the database because they are effectively already there since they are represented by the absence of a corresponding Item. Sparse relations are very efficient because of this.

Data can also be stored into a new relation without creating the new relation in advance. There is no need for the equivalent of an SQL 'CREATE TABLE' statement, and any existing database is immediately ready to store data into any new relation, since an empty relation is represented as the absence of any Items containing its id. A newly extended database is forwards compatible as well as backwards, since a newer database version can be used with an older application that simply ignores the new relations or columns.

## Adding Data

Adding cell values to the database is simple. A Cursor is allocated, components are appended to it, and it is passed to `insert()`. The temporary Cursor may then be returned to the pool for quick re-use:

```
Cu cu = Cu.alloc(); // Get a temporary Cu from the pool. ~1666 chars.
// append() returns the Cu again, so append() can be chained.
cu.append(databaseId).append(relationName)
    .append(primaryKey).append(columnId).append(cellValue);
db.insert(cu);      // 'db' is some pre-existing ItemSpace
Cu.dispose(cu);    // Optionally return Cu to pool for maximum speed.
```

If more Items are to be inserted, the above code can be repeated for each Item, but the `alloc()` and `dispose()` can be shared, surrounding the whole sequence, with the second and subsequent 'cu.append(...)' lines changed to 'cu.clear().append(...)'.

## Accessing Data

The `next()` method can be used with various protected prefix lengths and a truncation technique to enumerate, among other things: all the databases, all the relations in a database, all the keys in a relation, all the records in a relation, all the columns in a record, or all the cell values in a multi-valued column.

Let's examine finding a cell value for a given row and column of the database. First, an Item is constructed having the `databaseId`, `relationName`, `primaryKey` for the desired row, and the `columnId`. Then 'boolean `next(Cu cu, int pl)`' is applied to that Item, with the protected prefix length `pl` set to the initial length of the entire Item. The result will be a `true` or `false` from `next()`, and a possible change in the Item in the Cursor. If a `true` is returned, the end of the Cursor contains the column value, otherwise, the column value is `null` (not the Java `null` but the database concept of `null`, i.e. a missing value.)

Here is the code for retrieving a column value:

```
Cu cu = Cu.alloc().append(databaseId)
    .append(relationName).append(primaryKey).append(columnId);
int prefixLength = cu.length();
if (db.next(cu,prefixLength)) { // Item exists, column is not empty
    System.out.println("value=" + cu.componentAt(prefixLength));
}
Cu.dispose(cu);
```

The actual value extraction is simply a `cu.componentAt(prefixLength)`, which gets a wrapped primitive from the offset `prefixLength` in the Cursor `cu`. If the component type were known to be a `long`, for example, `long longAt(int off)` could have been used and a wrapper construction

avoided. The Cursor `cu` is allocated from the temporary pool in this example, although a preexisting Cursor could have been reused after a `cu.setLength(0)` or `cu.clear()`. The `Cu.dispose()` is optional but recommended for speed, since garbage collecting a 1666 char array can be slow. A Cursor must be used by only one Thread at a time.

If the column had been of a multi-valued type (which is illegal in conventional RDBM's) then the 'if' would be replaced by a 'while' – that is all. As will be described below, multi-value columns can be very useful.

## Enumerating the Rows

Now let's examine a less common operation: enumerating the set of rows in a table. This works, with a few modifications, for enumerating a subsequence of rows as well. First, an Item is constructed containing only the database Id and table Id. In a loop, it is moved along using `next()` until `false` is returned. On each iteration, an Item will be returned that contains, after the database Id and table Id, not only the key, but a column Id and a value. These extra parts – the column Id and value - are simply truncated away, and the key is 'incremented' in order to prepare it for the next iteration. The incrementing adds one to the last char of the Cursor, and if this brings the char around to zero again, the previous char is incremented and so on. Without this incrementing, the Cursor would not advance because the `next()` would only retrieve the same key, column Id and value. The Cursor would only be extended rather than moved forward. Here is the code:

```
Cu cu = Cu.alloc().append(databaseId).append(tableId);
int prefixLength = cu.length();
while (db.next(cu,prefixLength)) {
    String key = cu.stringAt(prefixLength); // Or componentAt()
    System.out.println("key=" + key);
    int offsetAfterKey = cu.skipString(prefixLength);
    cu.setLength(offsetAfterKey); // Truncate
    cu.incrementSuffix(prefixLen); // Avoids same key
}
Cu.dispose(cu);
```

In the above code, setting the Cursor's length to the offset after the key truncates the Cursor before the `columnId` and value. The application can also use `skipComponent(int offset)` to skip over a String or long or other type component without knowing or being dependent on the component's type. Also, `skipComposite(int offset)` can move over several concatenated components at once, allowing dynamic composite keys and values, by internally repeating `skipComponent(int offset)` until a special delimiter component is found which is not one of the primitive encodings, as described below in the Entity-Attribute-Value data structure.

## Dynamic Indexes

Often an extensible relation will need inversions on some of its secondary fields. To do this, Items of the following form can be used to create an index:

Database Id	Index Id	Secondary Key	Primary Key1	Primary Key2 (...)
-------------	----------	---------------	--------------	-----------------------

The set of index Id values here must be disjoint from the set of relation Id's. Accessing by secondary key requires code like this, for a string-valued primary key:

```

Cu cu = Cu.alloc().append(databaseId).append(indexId)
    .append(secondaryKey);
int prefixLen = cu.length();
while (db.next(cu,prefixLength)) {
    System.out.println("primary key =" + cu.stringAt(prefixLength));
}
Cu.dispose(cu);

```

This is almost identical to the code for accessing a column value, except that there is a ‘while’ loop instead of an ‘if’ statement since there may be multiple primary keys that match a given secondary key.

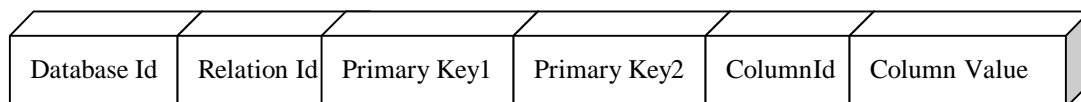
Note that in the data structures built so far, inversions are maintained by the application program and are not automatic as they are in relational systems. The maintenance of the inversions in existing applications can be easily handled instead by utility code that factors out the work. It is often argued that the existence of an inversion on a particular column of a relation is something that should be determined by the DBA at deployment time or later in order to maximize performance. In actual practice, applications make assumptions about which columns are indexed in order to provide acceptable performance. It is not a serious limitation that the application hard-wires the indexing structure. When the DBA makes a mistake by removing an index, an application can be catastrophically slow; when an extra index is added, it is a waste of space and cycles.

But even a hard-wired index structure can be changed by future versions of the application program, or by special application features. New indexes can be created and maintained invisibly, and old indexes can be abandoned or deleted, all without obsoleting the file format. Indexes that are intended only as temporary sorts can be created on-the-fly and deleted when no longer needed, without DBA intervention. ItemSpaces do not provide, at a low-level, the ad-hoc definition of orderings like the ‘ORDER BY’ clause of SQL, but there is no restriction on application code that prevents something similar being implemented at a higher level.

In practice, real relations are often indexed on all meaningful columns so that ‘ORDER BY’ will be reasonably efficient because dynamic sorts are minimized. The same thing is done in typical ItemSpace-based applications as well, so dynamic sorts can often be avoided. Many applications cannot tolerate the delays or unpredictable storage space requirements of dynamic sorting anyway, and will require a fixed set of inversions.

## ***Composite Keys***

Is very often the case that a primary or secondary key needs to be composed of more than one primitive value. Using the formats for data primitives discussed so far, this requires code that is aware of the composition. The Items might be formatted as follows:



To add a column value into an extensible relation that has a composite primary key, you might use code like this:

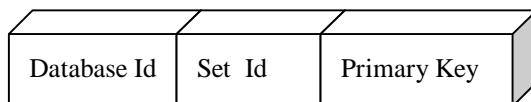
```

Cu cu = Cu.alloc(); // Get a temporary Cu from the pool.
cu.append(databaseId).append(relationName)
    .append(primaryKey1).append(primaryKey2)
    .append(columnId).append(value);
db.insert(cu);      // 'db' is some pre-existing ItemSpace
Cu.dispose(cu);    // Optionally return Cu to pool for maximum speed.

```

## Sets and Multi-Valued Columns

The code above, shown under ‘Dynamic Inversions’, has a ‘while’ loop that allows multiple primary key values to be read in sequence. This is the same kind of code that would be used to access any kind of set of values, such as a set of primary keys for rows that have been flagged as being in a particular category, or a set of values for a multi-valued column. Sets of rows can be created and deleted dynamically simply by inventing unique set Id’s and storing or deleting Items like this:

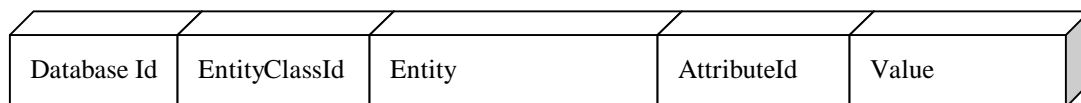


Multi-valued columns are normally prohibited in relation theory, but are actually quite useful. Sometimes a column is considered at schema design time to be single-valued, but is later discovered to need to be multi-valued. This actually happened in the development of the BoilerBase email indexer application. A column containing the file offset of a message in an external mail file needed to become multi-valued and composite because it was realized that the same message could exist in more than one mail file. Some of the mail files might not be available at a particular time, so the multiple values had to be tried in sequence until a working one was found. The field value was simply stored multiple times by inserting multiple Items, one for each message pointer, where a message pointer became a composition of an offset and a mail file name. The rest of the Item components stayed the same.

## The Entity-Attribute-Value Model

The relational model has been almost universally accepted for persistent structural data storage, so a new model has a considerable uphill climb to make before being embraced. However, the Entity-Attribute-Value model can be viewed as an extension of the relational model that is natural in an ItemSpace. It unifies the concepts of relation and index and allows multi-valued and composite columns. The resulting structure is simpler and more satisfying.

The term ‘relation’ is replaced by the term ‘*EntityClass*’. Each primary key value of each relation is considered to identify an ‘*Entity*’ rather than a row. The column id’s are called ‘*Attribute Id*’s and the cell values are still called *Values*. (This may not be considered a significant change in names but it seems helpful in practice.) Data is stored in Items structured like this:



where an Entity can be composite as in relations, but Value’s can also be composite, unlike in relations. Databases contain EntityClasses, which contain simple or composite Enties, which contain Attributes, which contain simple or composite Values. The fact that Values can be composite as well as Entities gives an EAV Item a certain symmetry. Also, the Values of any Attribute may in principle be multi-valued in the EAV model, although the semantics of a particular Attribute may not allow it. Relational table cells, on the other hand, always contain single, non-composite values. The capability of Attributes to be multi-valued

combined with the possibility of composite Values allows relational indexes to be ‘folded’ into Entity Classes.

## An Example EAV Database

Let’s examine the following example EAV database. The table below shows an EAV structure, one Item per row, with no composite Entities or Values. Composite Entities or Values could have been indicated by placing them in single table cells with some kind of separator between the simple values. This table format would not necessarily be visible directly to a user of the application, but would be interesting to the application author for debugging purposes in an “Item Editor”, which is a general-purpose direct ItemSpace browser and modifier.

Database Id	EntityClass Id	Entity	Attribute Id	Value
Chem	Student	Jenkins, Waldo	Student #	3451
			GPA	3.58
			Attends	Organic 101
				Proteins 202
		...son, Paul	Student #	2665
			GPA	3.2
			Attends	Organic 101
	Lecture	Organic 101	Attended By	Jenkins, Waldo
				..son, Paul
		Proteins 202	Attended By	Jenkins, Waldo

As before, blank cells indicate prefix compression. The two occurrences of ‘..son, Paul’ show intra-component prefix compression. The Database Id is usually implemented as a long, for compactness, and the EntityClass Id and Attribute Id’s are implemented using special component types that have not been mentioned above: that is why they do not sort as strings. The Entities happen to be all strings, and the Values have a mix of long and string types. None of the Entities or Values are composite in this example, as this would be difficult to show in a table form. A special ‘Item Editor’ can be used to browse an ItemSpace having composite Entities and Values, or even heterogenous Entities and Values, i.e. Entities and Values with an arbitrary mixture of primitives and mixed composites

Note that Jenkins attends two Lectures, and the ‘Organic 101’ Lecture is attended by two Students. There is a many-to-many relationship connecting Lectures and Students, yet there is no need for a special ‘connector’ relation for it. Also, no indexes on <Lecture, Student> or <Student, Lecture> are needed. For example, in order to enumerate the Students for ‘Organic 101’, the <Chem, Lecture, ‘Organic 101’, ‘Attended By’> prefix Item can be used in a ‘while’ loop to get the two Students using code described under “Dynamic Inversions” above. To get the Lectures attended by Jenkins, a prefix Item of <Chem, Student, ‘Jenkins, Waldo’, Attends> would be used.

The Attributes ‘Attends’ and ‘Attended By’ are called *inverse Attributes*. When an Item is inserted into the database containing one of these Attributes, the inverse Attribute is always used to create an ‘*inverse Item*’ which is also inserted. Deletion works similarly, so that the pairs of inverse Items always occur together or not at all. Also, when an inverse Item is created, its ‘*inverse EntityClass*’ is determined from the Attribute being inverted as well. The inverse EntityClass is like a data type for the Values of the Attribute being inverted. In the example, the Values of the ‘Attends’ Attribute are of Entity Class ‘Lecture’ and the Values of the ‘Attended By’ Attribute are of Entity Class ‘Student’. The enforcement of the pairing between inverse Items constitutes the equivalent of referential integrity constraints in a relational database, but is

totally symmetrical. There is usually no need for the equivalent of a relational index, since inverse attributes serve the same purpose: they connect two EntityClasses, where one of the EntityClasses takes the place of the value domain on which the relational index would have been defined.

There are special component types for EntityClasses and Attributes which are encoded in the Cursor as a special code followed by a normally encoded long id value. The special code and long id value are considered as a single unit. The type codes are `Cu.ENTITY_CLASS_TYPE` and `Cu.ATTRIBUTE_TYPE`, which are returned from `Cu.typeAt(offset)`. There is a `EntityClass.entityClassAt(int offset)` and a `Attribute.attributeAt(int offset)`, as well as `Cu.append(EntityClass ec)` and `Cu.append(Attribute att)`, and the corresponding `Cu.skip..()` methods. There are `EntityClass` and `Attribute` Java classes. An `Attribute` can be constructed using `new Attribute(long id)` and the id value of an `Attribute` can be retrieved with `long Attribute.getId()`. There are `long.entityIdAt(int off)` and `long.attributeIdAt(int off)` for working with EntityClasses and Attributes in an Item in a Cursor without construction.

These special component types allow the boundaries of a composite Entity to be determined at runtime, making composite Entities self-delimiting, and allowing mixing primitive types and component count of composite Entities (primitives are considered to include string, date, and array types.). A generic `Cu.skipComposite()` method allows skipping all primitive types up to the next non-primitive component. `EntityClass` and `Attribute` Objects are normally constructed as `static final`'s and appended to `Cu`'s as needed, but are not normally constructed dynamically. The `static final` `EntityClasses` and `Attributes` serve as type-safe identifiers for their corresponding concepts.

The EAV model is quite practical, not just a concept, and is the actual basis for the BoilerBase Email Indexer and Categorizer application (see <http://www.boilerbay.com/>).

## CLOB's and BLOB's

Arbitrary-length character sequences and arbitrary-length binary objects, known in the relational world as CLOB's and BLOB's for "Character Long Object"s and "Binary Long Object"s are very awkward to deal with in most systems. In an `ItemSpace`, they can be stored broken down into chunks that fit into Items, with the chunks numbered. The Item structure is shown below:

Any Prefix	Index(1)	1024 chars of text in a char[] component
Same prefix as above	Index(2)	1024 chars of text in a char[] component
Same prefix as above	Index(3)	Last chars of text, usually less than 1024.

A CLOB can be written into the `ItemSpace` using `new CharacterLongObjectOutputStream(ItemSpace db,Cu prefix)`, and then writing to this new `OutputStream`. A `CharacterLongObjectInputStream` will read it back in a similar way. This technique of streaming data into and out of the database is independent of the other aspects of the data model, since only a prefix Item is required to identify the CLOB. Thus a CLOB can be stored as the value of an EAV Attribute if desired. A CLOB can be considered another kind of multi-valued Attribute, showing again the importance of the multi-value feature of the Entity-Attribute-Value model.

In order to make it clear that a particular Item or sequence of Items represents a CLOB rather than just a set of values, one last component type has been added: the `Index` type. An `Index` component is - like

EntityClass and Attribute component types - a special internal type id char followed by a normally formatted long component. The internal id char and the long component are considered as a unit for the purposes of appending, skipping, and extraction on a Cursor. There are methods `Cu.appendIndex(long index)`, `Index.indexAt(int offset)`, and `Cu.skipIndex(int offset)`. There is also an `Index` Java class, which is manipulated like the `EntityClass` and `Attribute` component classes discussed above. An `Index` Object contains a long index value which can be retrieved by `Index.getIndex()`. In order to avoid construction of an `Index` Object using `indexAt(int off)`, `long indexValueAt(int off)` can be used. In fact `Index` Objects are not normally constructed at all.

Because of the special `Index` component type, it is possible to store short Strings directly in an EAV value in one Item, while long strings automatically are detected and switch into CLOB mode, using multiple Items This way a single helper method can write or read an unlimited-length String transparently, using the proper encoding of the two for any particular length.

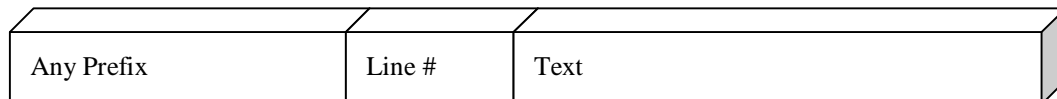
Note that empty, unused, deleted or future CLOB's and BLOB's take no space, and there is no fragmentation or space wasted at the end of a CLOB or BLOB that does not take an integral number of blocks. In principle, there can be random access over a CLOB or BLOB but this has not been implemented as of this writing. Since CLOB's and BLOB's work with streams, the entire char array value or byte array value does not need to come into memory at once. A short piece of code can easily read the entire CLOB into a String if necessary, but of course this risks `OutOfMemoryErrors`, which must be handled by specific application code.

## ***Future ItemSpace Data Structure Designs***

This section discusses data structures that have not been put to use but which offer promise.

### **Easily Editable Text**

Line numbering can be applied to texts, which can be broken down into lines or pieces of lines. A floating-point line number allows lines to be easily deleted, and new lines to be inserted between existing lines up to a certain limit, as shown in the Item below. This technique is much like a CLOB, except for the floating-point line numbers.



The editing limit depends on how well the inserted lines can be numbered so that the logical space between lines can be used up as slowly as possible. Truly random editing - meaning individual lines are inserted between random existing lines - causes each inserted line to use up at least one new bit in the fractional part of the floating point number. The best random algorithm is simply to split the difference between the line numbers of the adjacent lines. It would seem that this would cause the numbering to run out of space quickly, but it actually works rather well when the lines are added in a truly random order. Also, areas of frequent line insertion tend to be associated with frequent line deletion, and deletion yields back some of the numbering space.

When the lines are being added in long sequences at each point of insertion, or especially when lines are appended in large amounts at the end, the 'split-the-difference' system runs out of space quickly. The problem is that the lines at the end of the inserted sequence end up with very little inter-line space. This can be helped by making sure that lines are only added in 'editing sessions' during which all of the edits are accumulated before being applied to the `ItemSpace`. When a particular sequence of lines is to be inserted at a given location as a batch, the difference between the adjacent line numbers in the existing text is divided

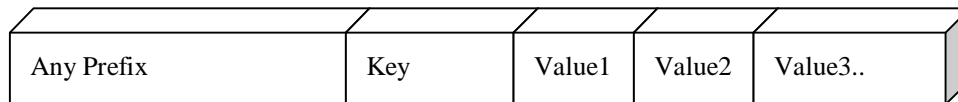
up evenly between the new lines. This technique helps quite a bit because the actual insertion locations of the inserted sequences is rather random, so the loss of fraction bits is minimized. When the floating-point numbering system has run out of fraction bits for some text, the text can be renumbered. There are plenty of other structures possible for randomly editable texts, but the floating-point numbering system is probably the simplest. It also shows, to some extent, the editing history of the text.

In order to extend the effective fractional part, another floating point number can be suffixed onto the first and so on. By suffixing line numbers in this way, it is also possible to do the entire job using long's instead of floats. Note that line numbers never change, so a pointer into a block of text is a fixed composite Item which can be used as a Value elsewhere.

## Trees

Trees can be represented by a concatenation of variable numbers of node names or numbers into a 'path' identifier and used as an Entity. If each node number is a long, there can be a huge number of child nodes for any given node. If the node numbers are String components instead, a directory structure results. The path identifier can be used as an Item prefix for getting at node data. This kind of tree might work well for XML as a kind of DOM.

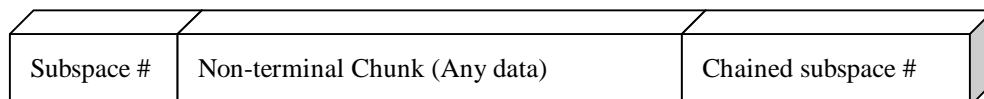
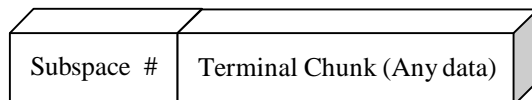
## Item Packing



Frequently accessed sets of values that are accessed using the same Item prefix can be put together into one 'packed' Item like a traditional database record. Quite a few values can be packed together, so a considerable speedup is possible. To the left of the Item, any kind of prefix can be used, followed by some kind of key, and then the value sequence. This structure is only helpful if the access pattern stays inside the cache, since cache misses will be limited by disk speed, and the overhead of the normal multiple `next ( )` invocations will not matter much in the disk-limited case. Nevertheless, a large performance gain can result in many typical situations, at the possible expense of extensibility as compared to the structures described above. Adding fields at the ends is easy, and existing Items that end before the new fields can be interpreted as having null values in those fields. Removing a field is more difficult, but new Items can have that field be replaced by a short component of the same type, and values for that field in existing Items can simply be ignored.

## Item Chaining

Items have a fixed maximum length in practical ItemSpaces, but there may be rare cases where the Items really need to be longer. To allow this, a special Cursor class might provide a view of an ItemSpace as having unlimited-length or 'virtual' Items by breaking the virtual Items down into 'chunks' which are stored in a normal ItemSpace. Each chunk is assigned to a 'subspace'. The chunks are chained together into a tree. Each Item in a subspace looks like one of the following two:



The first type of Item occurs either for short virtual Items or for chunks at the leaf level of the subspace tree; the second type is for the branches. Each subspace is identified by a Long subspace number. There is a special 'root' space, space 0, which contains all the short virtual Items and the initial chunks of longer virtual Items. Each branch chunk ends in a '*chained subspace number.*' A subspace maps one-to-one to a virtual Item prefix, where the prefix is a certain number of complete chunks. The existence of the chained subspace number on the end of a chunk Item is signalled by the length of the chunk Item, not by a special code within the non-terminal chunk data. Note that there is no database Id – that concept appears at a higher layer, as part of a virtual Item.

Providing concurrency in a chained ItemSpace is complex, but it can be done.

## **Schemas**

A future improvement of the EAV model involves putting the metadata, such as the definitions of the inverse Attributes and their corresponding Classes, into the database along with the other data, using the EAV model on itself. This constitutes a kind of schema. The metadata can be brought into memory when the database is opened and used for constraining item insertions and deletions to maintain the pairing of inverse Items, among other things. It would be possible to create a database in which Entity and Value data types and compositions are completely fluid or are tightly constrained, and possibly alterable at runtime.

A full discussion of EAV schema models will have to be provided elsewhere. See U.S. Pat 5,010,478 for a description of a simple schema technique.

## **Persistent Objects**

Object databases could serialize and write Objects using a custom ObjectOutputStream to an ItemSpace, often one Item or one BLOB per serialized Object. References could be resolved either by creating a new identifier and a separately stored Object or by recursion into any contained objects as usual for serialization.

## ***Appendix A: Comparison of ItemSpace with Other Models***

The ItemSpace model is not a universal solution to data storage problems, but is much better than available technologies in certain situations. Here are the chief alternatives and a few of the problems and advantages they have compared with the ItemSpace model. This discussion covers not only model considerations but also implementation issues.

## Custom File Formats

The design of application-specific file formats for persistent data of various kinds has always been full of complications and pitfalls. Often, each successive version of a typical application's file format changes as the weaknesses of the preceding versions' format come to light, and format converters must be created, at least for the forward-compatibility direction. As time goes on, file formats become more and more complex, and performance and extensibility suffer. One basic problem is that the fundamental model of a file as an 'array' of bytes is so very low-level: What is needed is an abstraction which raises the modeling level as far as possible consistent with rich representational capability, high performance, and a simple programming interface. The ItemSpace abstraction can be implemented using a well-established, static file format and provides application-level extensibility so that forwards and backwards incompatibilities are minimized.

## The Relational Model

An obvious choice for complex persistent data structures is the Relational Database Management System. Here are few of the problems with it:

- RDBMS have a high degree of human interaction, including running the proliferating SQL scripts for schema creation and upgrading, dumping the log periodically on tape, monitoring various resource usages, rebuilding indexes, restoring from tape after a dirty shutdown, and so on. An RDBMS instance usually represents a significant and continuing investment in effort, resources, and money. It is often visible to the user of any system it is a part of, or at least to those who service and maintain it, whereas an ItemSpace-based system can be virtually transparent to all involved. Much tuning is often required, such as the creation and dropping of indexes, block sizing, and so on. An ItemSpace is so simple that there is no management necessary.
- RDBMS do provide a powerful concurrent transactional capability, but this comes at a price. Deadlocks may occur, unpredictable delays and variations in response time are common, and the possibility of random rollbacks presents users with occasional error screens and requires programmers to add code that anticipates rollbacks at any time.
- Data storage efficiency in RDBMS' is normally sacrificed in favor of some performance improvement in certain situations, although using the full generality of SQL will bring the performance back down substantially, such as when joins or sorts are required. An ItemSpace can be stored in a single B-Tree that efficiently manages a single pool of blocks.
- An SQL SELECT statement may bring large or unpredictable amounts of data into memory, risking OutOfMemoryException. An ItemSpace retrieves or stores exactly one Item per operation, using no temporary storage, and generating little or no garbage.
- RDBMS licenses are often significant per CPU or per unit sold.
- RDBMS can access data remotely, transferring data as entire tables, if desired, while ItemSpaces are normally (but not necessarily) accessed from inside one JVM. However, server-side cursors are required in order to limit the size of the units of transfer and to keep the response time down. Some RDBMS' do not have server-side cursors, transferring entire result sets all at once.
- RDBMS result sets usually cannot be randomly accessed or accessed sequentially in reverse order. ItemSpaces are randomly or sequentially accessible in either direction, presenting the same dynamic, updateable view to all Threads. ItemSpaces have no result sets, hence no size limitation: the entire ItemSpace is visible at all times.
- SQL is not obvious to many programmers. An ItemSpace has a tiny learning hill to climb.

- SQL statement response time is of the order of 10 milliseconds to tens of seconds or worse, due to possible parsing, typechecking, locking, sorting, and other delays. ItemSpace in-cache response time is usually 100 times less with a worst-case of a few seconds when the block cache is temporarily overloaded with updates.

## The Object-Relational DBMS Model

Object-Relational mapping tools can generate custom Java source code to create and access Objects of special 'Data Access' classes that represent rows in a relation. Each relation maps to one Data Access class. These systems are excellent from a programming-reliability standpoint, since they allow type-safe access and modification of fields in Objects rather than using SQL from the point of view of the business logic. But there are limitations.

- The programming is burdened by the need to generate and support all the special Data Access classes. SQL statements must be created either manually or with special tools to perform the persistence chores associated with reading and writing rows into and out of Data Access Objects.
- There can be considerable overhead in transferring Objects back and forth between the JVM and the RDMBS. For example, if a subset of the rows in a relation are to be retrieved or operated on, they normally must all be retrieved and placed into memory first, such as with a custom SELECT statement, before any can be used. There may be many such rows, and the set of Objects may be large, so they may not fit into memory.
- For the entire time the rows are in memory as Objects and especially if they are potentially modifiable, they may all need to be locked, or inconsistencies may result. Locking creates the potential for deadlocks, and optimism creates the potential for rollbacks. RDBMS are excellent at maintaining overall database integrity, but they can only protect data outside the database by locking.

With the ItemSpace model, all Threads modify exactly the same logical data store at all times without copying the data in or out. This means that the ItemSpace can be used as a relatively fast inter-Thread communications system. The model allows a maximum of concurrency and speed, has no memory size or temporary space usage limitations, no locking delays, and no internal deadlocks. On the other hand, the ItemSpace model is not a solution to the general dynamically-defined concurrent transaction consistency problem, since interference between transactions going on in different Threads is left up to the ItemSpace client application to resolve. An ORDBMS could be built on top of the ItemSpace model, but the real strength of the ItemSpace model is in applications where the transactions can be pre-planned, Thread isolation requirements are understood in advance, and a single, simultaneous global commit for all Threads is sufficient. (The global commit requires only that Threads be coordinated enough to prevent any Thread from committing in the middle of a transaction.)