hel052.doc
Published at 3:27 pm on 27 September 2000

WG3: HEL-052

25 September 2000

Authoritative version: hel052.pdf

ISO

International Organization for Standardization

British Standards Institution
IST/40
Data Management and Interchange
ISO/IEC JTC 1/SC 32
Data Management and Interchange
WG 3
Database Languages

Project:	32.03.05.02.00 (SQL:2002)
Title:	Possible problems with characters, the story continues.
Author:	J M Sykes (United Kingdom)
Source:	UK Expert
Status:	SQL:2002 Discussion paper

Abstract: **We continue the exploration of possible problems with SQL's treatment of UCS (Universal Character Set) character data, i.e. encoded in accordance with ISO/IEC 10646 or Unicode3. Some likely future proposals are outlined.**

References: **A list of sources is to be found in Annex B, Sources**

1 The story so far

[BHX080], in Section 2, includes a brief résumé of the history of SQL's treatment of character data and identified a number of issues in need of consideration.

[BHX117] expanded on these issues somewhat, in the light of subsequent research.

Both these papers were discussed by WG3 at its meeting in Warwick in July 2000, and the minutes [HEL001] record that the following should be listed as possible problems:

[1] Counting Characters 691, 774

[2] Normalization (Language Opportunity 776)

[3] Collations (Language Opportunity 775)

Other items requiring attention are PPs 634, 643, 759, and LOs 134, 212, 268, 426, 713, 758, 761. These are not addressed in this paper.

The purpose of the present paper is to carry the discussion a little further.

We assume the reader to be aware of the character model of ISO/IEC 10646 and the Unicode Standard 3.0, at least as far as is relevant to the current discussion. The reader who lacks such awareness is recommended to consult Annex B. The reader who is confused by the variety of concepts and the multiplicity of terms defined for each concept, may find it useful to consult Annex A. In selecting the terms to use ourselves, we have followed what appears to be fairly general usage and used *code point* for what Unicode calls *(Unicode) scalar value* and ISO calls *code position*, and *code unit* for what Unicode calls *(Unicode) code value* and for which ISO defines no specific term.

Annex A "Terminology and Notation" contains a tentative correspondence between [ISO 10646] and [Unicode3] terminology.

Annex B contains a lightly annotated list of sources which have been referred to in the preparation of this paper, and to which the reader may wish to refer for authentication.

In what follows, the emphasised phrase "**We propose**" means that we intend to submit a formal proposal in the future, unless serious objections are raised.

2 Basic principles

We propose as a fundamental principle that the analogy between the treatment of character data and that of numeric data should be as close as possible.

A perfect analogy would imply the following basic principles:

1) A character is an abstraction. It is not a visible symbol on paper or screen (a glyph), nor is it a pattern of bits (a code).

   Problem 1: As [CharSetHarmful] points out, the term *character* is dangerously confusing when used to denote this abstraction. *Grapheme*[1] is used with this meaning, but raises further problems of culture dependency.

   Problem 2: It is impossible to determine in every case whether or not a syntactically valid Unicode encoding represents what some user thinks of as a character, whether culture-dependent or not.

2) **We propose** that all character data is normalized, automatically and invisibly to the user, in the same way as, and for the same reason that all approximate numeric data is normalized.

[1] *grapheme* is defined in [UTR#18], and the definition is referred to in Annex A.1.

Problem: More than one normalization form is available, and even the same user may sometimes prefer one, sometimes another.

3) **We propose** that casting between the various representations of the same character string is automatic and invisible.

   The three UCS Transformation Formats (UTF-8, UTF-16 and UTF-32) are interconvertible without loss, so this objective is achievable. Nevertheless, some implementors might be reluctant to provide hidden, automatic casting, and the potential costs might be considered unacceptable.

4) **We propose** that results of operations on character data should be independent of representation (including normalization form).

   If *CS* is a character string, then CHAR_LENGTH (*CS*) should give the same result irrespective of its transformation format (UTF-8, UTF-16 or UTF-32). Thus, in UTF-16, a surrogate pair (of code units) must count as, at most, 1 (one) 'character' (whether, by *character* is meant code point or grapheme).

   Note that OCTET_LENGTH is obviously representation-dependent, so that this principle cannot be applied to it.

Sadly, perfection is not achievable; nevertheless, it is highly desirable to approach it as closely as practicable.

There are two principal obstacles to perfect analogy:

a) Cultural differences. It is impossible to satisfy all of the users all of the time. There will be some who insist on a character model appropriate to their own culture, others (multi-nationals) who have good reason for requiring a culture-independent compromise. Cultural collations already exist, and a user option to apply one to an individual comparison operation or to all such operations within some scope (SQL-session, server, schema, ...) will go some way towards solving the problem. But the string that includes text from different cultures may still exist, and must be dealt with somehow. If mark-up is used within the string to indicate different cultural requirements, is this the concern of SQL? Probably not.

   We note that cultural dependencies in general are usually specified as part of a *locale*, a term not used in SQL.

b) Invalid strings. There are various classes of invalid data, ranging from the invalid code point or unmatched surrogate, both of which must be rejected, to the nonsensical combining character sequence that cannot reasonably be recognised as such, but which cannot always be detected. (See, e.g., [UTR #17], section 4 "*Character Encoding Form (CEF)*")

3 An exemplar

There are at least two good reasons why SQL should consider adopting, with adaptation as necessary, the "Character Model for the World Wide Web" [W3C-CharMod].

1) The character handling problems of the World Wide Web have much in common with those of SQL.

2) Users of Java, SQL/OLB, and XML will expect treatment of characters to be consistent in the different contexts.

One or two comments are necessary on [W3C-CharMod]:

1) The existence of UTF-32 is not recognized, for reasons that are obvious if one looks at the dates on the documents. When it is, we would expect "UCS-4" in the first bullet of the third bullet list of section 1.1 "Why is this document necessary?", to be changed to "UTF-32".

2) At least one issue of great importance to SQL is not resolved, viz. string indexing (meaning the use of a number to indicate a position within the string). Section 5, "String Indexing" refers to the earlier requirements document [W3C-CharReq], which presents a thorough statement of requirements but

reaches only one conclusion useful to SQL, viz. that "If string indexing is based on early uniform normalization, then this may help to make implementation easier". Sadly, while it ackowledged that "The String indexing specification shall be prepared quickly", it doesn't seem to have appeared yet.

SQL needs to find its own resolution of this issue, with due regard to whatever other work is being done.

3) Understandably, [W3C-CharMod] focuses on transmission and rendering of character data, and is consequently less concerned than many database administrators with the validity of the data. It is therefore hardly surprising that constraints of the kind required on data stored in a database are not discussed.

See also Annex A.3, "A note on notation".

4 <character set name>s

[CharSetNames] describes itself as a register, maintained by the Internet Assigned Numbers Authority. It lists "official names for character sets that may be used in the Internet and may be referred to in Internet documentation". Of the 200+ names on the list, the ECMA registry is cited as the source of about 80. However, for reasons we have not, as yet, been able to discover, the ECMA registry is not cited as the source of the six Unicode/UCS entries whose source is ISO-10646 or some variant of it. (The remainder of the names are mostly from industry sources).

Clearly, it is desirable that <character set name>s acceptable to an SQL-implementation should registered *somewhere*. While SC32 WG3 owes no allegiance to IANA, it would seem to be both suitable and convenient, if politically acceptable. Alternatively, we should explore the ECMA registry, which does not appear to show its face on the Web.

UCS names registered at IANA currently are:

Name	Source (recorded in registry)
ISO-10646-UTF-1	Universal Transfer Format (1), this is the multibyte encoding, that subsets ASCII-7.
ISO-10646-UCS-2	the 2-octet Basic Multilingual Plane, aka Unicode
ISO-10646-UCS-4	the full code space.
ISO-10646-UCS-Basic	ASCII subset of Unicode. Basic Latin = collection 1 See ISO 10646, Appendix A
ISO-10646-Unicode-Latin1	ISO Latin-1 subset of Unicode. Basic Latin and Latin-1 Supplement = collections 1 and 2. See ISO 10646, Appendix A. See RFC 1815.
ISO-10646-J-1	ISO 10646 Japanese, see RFC 1815
UTF-8	RFC 2279 (which cites Unicode and ISO10646)
UTF-16BE	RFC 2781 (which cites Unicode and ISO10646)
UTF-16LE	RFC 2781
UTF-16	RFC 2781

UTF-32 is effectively an alias of ISO-10646-UCS-4, and will presumably be resistered as such in due course.

Since SQL names cannot be hyphenated, **we propose** that the names UCS2, UTF8 and UTF16, as already used in [FoundWD], together with UTF32, be registered with the appropriate authority, as aliases where appropriate of character sets already registered, and the character sets so named be referred to collectively as *UCS character sets*. This will require some redrafting of [FoundWD] subclause 4.2.4 "Named character sets".

5 Specification of UCS <character string type>s

At present, [FoundWD] Clause 6.1 "<data type>" effectively contains:

```
<predefined type> ::=
        { CHARACTER [ <left paren> <length> <right paren> ] | ... }
        [ CHARACTER SET <character set name> ]
        | ...
```

It might be considered that Unicode is really only one character set (UCS), or possibly two (UCS-2 and UCS-4). At the other extreme, one might suppose that UTF-8 or UTF-32BE, or even UTF16-BE-NFKC should be considered distinct character sets. There are, however, three orthogonal properties involved:

Character repertoire: UCS-2 or UCS-4

Transformation format: UTF-8, UTF-16 or UTF-32, with endian options for the last two

Normalization form: NFD, NFKD, NFC and NFKC

All combinations of the above make sense, though UCS-2 and UTF-32 are unlikely to occur together. Moreover, endian options are probably not relevant to SQL.

We propose to recognise only the currently defined character set names, adding an option to <character set specification> to allow normalization form to be specified. This will then permit, for example:

```
CHARACTER (255)[2] CHARACTER SET UTF8 (NFD)
```

Furthermore, **we propose** that <user-defined character set name> be deleted, on the grounds that, while an implementor might provide such a facility if he wishes, few users will wish to use it, and the standard should not impose on implementors a burden of such dubious value.

6 Validity of character data

As [UTR #17] explains in section 4, a sequence of code units (ostensibly representing a code point) is either valid, in which case the code point may nevertheless be unassigned, or illegal being either incomplete (e.g. an unmatched surrogate) or explicitly illegal (e.g. `u+FFFF`).

We propose that an exception be raised if illegal data is detected; but whether an unassigned code point should get the same treatment is more doubtful.

[ISO10646], in Annex C (which is normative, and specifies UTF-16) states that "the only possible type of syntactically malformed sequence is an unpaired RC-element", meaning a code point that does not itself encode any character, but is reserved for use as a high or low surrogate (code unit). **We propose** that this condition also should produce an exception if detected.

We are uncertain whether an SQL-implementation should be required to check for invalid UCS data. However, normalization and, by implication, construction of a collation key will always reveal the above-mentioned invalidities.

7 Normalization

7.1 Responsibility for normalization

Since all character data is required to be normalized, the questions arise:

[2] For the units of <length>, see 9.1 "<length> in <character string type>", below.

1. What assumptions is the SQL-implementation permitted to make?
2. Should the SQL-implementation check such assumptions, and if so, when?
3. What should happen if the assumption is found to be false (some character string is found not to be normalized)?
4. What is the worst that can happen if unnormalized data goes undetected?

There are three classes of data to consider:

SQL-data: The worst that can happen if data is unnormalized is that a binary comparison may return *False* when the user might have expected *True*. The converse cannot occur. The collation algorithm normalizes in any case. **We propose**, therefore, that the SQL-implementation be permitted to assume that all SQL-data is normalized, and does not need to be checked explicitly.

Literals: **We propose** that every character literal be normalized immediately. The fact that `'a`' = 'à'` returns *True* should raise no more eyebrows than the fact that `1E0 = 1.0` already returns *True*.

Host variables and routine parameters and results. All string data leaving the SQL-environment will be normalized. **We propose** that the SQL-implementation be permitted to assume that all character data entering the SQL-environment is normalized. If a user is unsure of this, they can invoke a normalize function (see 7.3 below) on it, on its way in.

However, if this is considered to be sacrificing safety for performance, then **we propose** that, the SQL-implementation be required to ensure that all UCS data is normalised on arrival in the SQL-environment, but that the users be given an option, with some appropriate scope (SQL-session, SQL-module?) to absolve the SQL-implementation of that responsibility, and accept it themselves.

7.2 A NORMALIZE function

Since a NORMALIZE function is simple to specify and not difficult to implement, **we propose** that it should be required in every SQL-implementation that supports a UCS character set. This will enable users who are unsure whether their host variables are normalized to reference them as, for example, NFC(:string).

7.3 An IS_NORMALIZED predicate

We propose that, although an IS_NORMALIZED predicate might appear useful, and clearly cannot be prohibited, it should not be required for conformance. The net performance cost of normalizing unnecessarily versus testing followed by normalization only when found necessary is unlikely to be great enough to bother about.

7.4 Concatenation operation

Although the operands of a concatenation operation may be normalized, a simple butted joint will, in some cases, produce an unnormalized result. For example, the result of " `'a^' || '.'` " (where `'.'` denotes DOT BELOW, u+0323) would be " `'a^.'` " whereas the normalized form would be " `'a.^'` " (example adapted from [UTR#15]).

We propose that conformance require the (appropriate) correct normalized form (whether or not other abnormalities in the operands are disregarded).

8 Collations

[ISO14651] is an IS for collations, whose application to UCS/Unicode is defined by [UTR#10]. The algorithm is simple and universally applicable, and it would be folly for SQL to disregard this and do its own thing.

A collation defined according to [UTR#10] has three or more levels; for the Latin script, these levels correspond roughly to:

1. alphabetic ordering
2. diacritic ordering
3. case ordering.

These three levels allow a default collation to satisfy far more requirements than a single SQL-type collation.

We propose that implementation of the UCS collation algorithm be required for conformance, and at least one collation (the default) provided.

Obviously, other, possibly culture-dependent, collations may be provided.

We note that some vendors provide for the use of locale-dependent collations which are invoked quite differently from the way specified in [FoundWD].

If a user wishes to define their own collation, they should not be required to do so in a way that is applicable only to one (or more) SQL-implementations. If [UTR#10] is followed, the collation can be specified by the contents of a text file.

We propose to allow a <collate clause> to be specified for every operation (possibly excluding exact equality) that directly or indirectly invokes the rules of the subclause "<comparison predicate>". Note that this must include POSITION, LIKE and SIMILAR.

Furthermore, **we propose** that <collate clause> be modified to allow the specification of the level of collation required.

Since, as far as we are aware, [ISO14651] does not provide any facility for ignoring trailing <space>s at the end of a character string, PAD SPACE will still have to be dealt with explicitly prior to invocation of the UCS collation algorithm.

9 Counting characters

Characters have to be counted in a number of contexts, and users are entitled to expect consistency, without unacceptable performance cost.

Counting either bytes or code units is not invariant with respect to transformation format; counting code points is.

However, counting code points is not invariant with respect to normalization form; counting graphemes is.

There are two kinds of context to be considered: the value of <count> in a <character string type> and the various parameters and results of SQL built-in functions whose semantics require characters to be counted.

9.1 <length> in <character string type>

<length> in, for example, <column name> CHARACTER (<length>), serves two purposes: to the user, it is a constraint on the number of characters allowed, which he/she will expect to agree with the result of CHAR_LENGTH (<column name>); to the implementor, it is an indication of the amount of storage required. The problem of counting in graphemes is that there is no upper limit on the number of combining characters that can follow one base character and hence constitute a single grapheme. We are not even aware of the practical

upper limit that may occur in those scripts that use combining characters frequently. Nor is the length in code points stable except in UCS-2 or UTF-32, since both UTF-8 and UTF-16 are varying width encodings.

Consequently, **we propose** that the existing <length> be treated as a constraint, in the same units as used for parameters and results (see 9.2 below). In addition, **we propose** that an option be provided to allow the user to specify the expected maximum length in bytes. Whether, if this additional option is not used, some algorithm should be specified for deriving maximum byte length from <length> is open for discussion. This would have to take into account the normalization form (explicit or implicit).

9.2 Parameters and results of SQL built-in functions

We have to consider:

o the numeric results of

- `CHAR_LENGTH (`*`SVE`*`)`
- `POSITION (`*`SVE1`*` IN `*`SVE2`*` )`

o the numeric parameters of

- `SUBSTRING (`*`C`*` FROM `*`S`*` FOR `*`L`*` )`
- `OVERLAY (`*`CV`*` PLACING `*`RS`*` FROM `*`SP`*` [ FOR `*`SL`*` ] )`

Applying basic principle 4 (see section 2, above) **we propose** that a character be defined to be a grapheme, since the number of code points is sensitive to normalization form.

More precisely *a grapheme* here means either: one control character or one format character or one (locale-independent) grapheme (informally: one base character plus all the combining characters that follow it and precede the next noncombining character. See UTR #18). A combining character that is not in a grapheme has nothing to combine with and hence counts as one character.

However, we recognise that users may require to use code points in some circumstances, and a suitable means must be found to make their choice clear.

10 Other considerations

There are a small number of other considerations, of less importance than the foregoing. We have no plans to submit a proposal on any of them at this time.

10.1 Script constraints on character data

It might be of assistance to users if they were able to specify simple constraints on character data, particularly that it is restricted to one or more scripts, see [UTR #24], though this is not necessarily the only desirable way to specify subsets of UCS.

Although it could be argued that such constraints can already be expressed in existing SQL, they would be laborious to specify in the absence of any way of specifying a contraint to be applied to every character in a string rather than to the whole string. Moreover, if they are specified explicitly, then they can be more efficiently implemented.

They also differ from existing constraints in that a fixup might be specified. See transliteration, below.

As yet, we are unconvinced of the usefulness of scripts, since in most cases more than one would be needed and checking would carry a performance penalty.

10.2 Transliteration

Transliteration is defined in the dictionary as the: action or process of ... rendering of the letters or characters of one alphabet in those of another. If we substitute the word *script* for *alphabet*, we can immediately see how a function that implements generally agreed transliteration processes could be useful. However, the specification of such processes is not within the scope of SQL, though SQL would do well to acknowledge their existence and provide links to them.

Transliteration would be a useful option in cases where violation of a script constraint is detected, see above.

Note: Transliteration is not to be confused with form-of-use conversion. It is a more usual word for what SQL currently calls translation (SQL definition: A method of translating characters in one character repertoire into characters of the same or a different character repertoire. Dictionary definition: The action or process of expressing the sense of a word, passage, etc., in a different language.)

10.3 Transcoding

The verb *transcode* is defined in the dictionary as "convert from one form of coded representation or signalling to another", and [Unicode3] effectively defines it in the same way ("Conversion of character data between different character sets"). It thus appears to have the same meaning as SQL defines for *form-of-use conversion*.

However, it is desirable to distinguish between:

a) Conversion between one coding scheme to an essentially different one, e.g. Unicode to EBCDIC, which is, in principle, a table look-up exercise that may be bedevilled by differences in repertoire.

b) Conversion from one transformation format of a coding scheme to another, e.g. UTF-8 to UTF-32, which is a simple, purely algorithmic operation.

10.4 Escape characters in literals

Some (many?) programming languages allow exotic characters to be included in character literals by preceding the hexadecimal digits identifying the codepoint with an escape character. Consideration should be given to introducing similar, preferably compatible, notation into SQL <character string literal>s. It is much easier to write or key 'a\u0323' than it is to discover how to get the same effect in SQL:1999.

This is a simple, and almost certainly worthwhile language opportunity, and is completely independent of any UCS considerations.

Annex A

Terminology and Notation

This Annex is intended to be purely factual, and is provided for the benefit of any reader who is more familiar with SQL than with Unicode. It has been prepared with care, but has not been validated, so it may contain errors. Should anyone find an error, we should be grateful to be told.

A.1 Corresponding terms in [Unicode3] and [ISO10646]

In a number of cases, [Unicode3] and [ISO10646] appear to use different, though similar, terms for the same meaning. The following table shows the *apparent* correspondences. As far as practicable, the text is taken directly from the relevant document.

Unicode3	**ISO-IEC 10646**
An *abstract character*: A unit of information used for the organisation, control, or representation of data (an abstraction that is represented by a coded character representation or by a combining character sequence.) (*D3*)	**4.6 character:** A member of a set of elements used for the organisation, control, or representation of data.
coded character representation An ordered sequence of one or more code values, that is associated with an abstract character in a given character repertoire. Except where explicitly stated otherwise, the term ***character*** always means *coded character representation*. (*D7+)*.	**4.4 CC-data-element (coded-character-data-element)**: An element of interchanged information that is specified to consist of a sequence of coded representations of characters, in accordance with one or more identified standards for coded character sets.
An *encoded character* is the (encoding) relationship between an abstract character and its scalar value.	**4.8 coded character:** A character together with its coded representation.
A (Unicode) *scalar value* is a positive integer less than or equal to $10FFFF_{16}$. Also known as *code position* or *code point*.	**Code position** is much used, but not defined in [ISO 10646]
A (Unicode) *code value* (aka *code unit*) is the unit of encoding. Thus: in UTF-8 it is 8 bits; in UTF-16 it is 16 bits, and is also known as a *Unicode value*.	**4.33 RC-element:** a two-octet sequence comprising the R-octet and the C-octet (see 6.2) from the four octet sequence that corresponds to a cell in the coding space of this coded character set.
A code value is one of:	
- A high-surrogate	RC-element from high-half zone
- A low-surrogate	RC-element from low-half zone
- A nonsurrogate	
A character is one of:	
	4.20 graphic character: A character, other than a control function, that has a visual representation normally handwritten, printed, or displayed.

Unicode3	**ISO-IEC 10646**
- A control or formatting character	**control function**
A graphic character is one of:	
- A base character	**non-combining character**
- A combining character	**4.12 combining character:** A member of an identified subset of the coded character set of ISO/IEC 10646 intended for combination with the preceding non-combining graphic character, or with a sequence of combining characters preceded by a non-combining character (see also 4.14).
A combining character is one of:	
- A spacing mark	
- A non-spacing mark	
A combining character sequence is a one of:	**4.14 composite sequence:** A sequence of graphic characters consisting of a non-combining character followed by one or more combining characters (see also 4.12). 1 A graphic symbol for a composite sequence generally consists of the combination of the graphic symbols of each character in the sequence. 2 A composite sequence is not a character and therefore is not a member of the repertoire of ISO/IEC 10646.
- An (effective) combining character sequence	
- A defective combining character sequence	
An effective combining character sequence is a base character followed by a sequence of combining characters. A defective combining character sequence is a sequence of combining characters that is not preceded by a base character.	
A grapheme (see [UTR #18]) is one of:	
- A locale-independent grapheme	
- A locale-dependent grapheme	
A locale-independent grapheme is what is represented by a combining character sequence.	
A locale-dependent grapheme is implemented as a collation grapheme. See [UTR #18] section 4.2 and [UTR #10].	

A.2 A note on code units and code points

The following table shows how many code units, of what size, are required for one code point in each transformation format:

	bits per Code unit	Code units per Code point	
		UCS-2	UCS-4
UTF-8	8	1 to 4	1 to 6
UTF-16	16	1 (i.e. no surrogate pairs)	1 or 2 (i.e. possibly surrogate pairs)
UTF-32	32	1	1

A.3 A note on notation

In what follows, '`h`' represents a hexadecimal digit.

[ISO 10646] subclause 6.5 defines "a short identifier for each **character**". Of the various options offered, the most commonly used is that used in ISO 9075], viz. `U+hhhh`. This is an abbreviation of an eight hexadecimal digit form, available only if the first four digits are all zero, i.e. if the character is in the Basic Multilingual Plane.

[Unicode3] defines the same form as representing "a Unicode value", which means a *code value* or *code* ***unit***, rather than a *code point* (see definitions above).

[W3C-CharMod] says "UCS codepoints [sic] are denoted as `U+hhhh`, where `hhhh` is a sequence of hexadecimal digits." Notice that this is not strictly correct, though it would be if every code **point** required only one UTF-16 code **unit**. Replacing 'UCS codepoints' with either 'UCS-2 codepoints' (though anything beyond the BMP is then excluded) or 'UTF-16 code units' (which excludes UTF-8) would remove the problem.

[UTR #17], in discussing the validity of sequences of code **units**, uses the notation `0xhh`, where the number of '`h`'s is variable, the maximum being six.

[UTR #18] section 2.1, in discussing the representation of Unicode characters in literal strings, says 'The most standard notation for listing hex Unicode characters **within strings** is by prefixing with "`\u`" '. However, noting that the "`u`" is referred to as UTF16_MARK, we realise that what is being identified here is a UTF-16 code **unit**, rather than a code point. Section 3.1 introduces "`v`" as UTF32_MARK, to be followed by exactly six hexadecimal digits. For why six will always be enough, see [UTR #19].

The opportunity for confusion is made worse by the fact that ISO 10646 and Unicode began separately, using different terminology, and the subsequent process of alignment has (apparently) required modifications to both, while terminology has not been reconciled.

Annex B

Sources

SQL Working Drafts

[FrameworkWD]	*(ISO Working Draft) Database Language SQL - Part 1: Framework (SQL/Framework*), September, 2000
[FoundWD]	*(ISO Working Draft) Database Language SQL - Part 2: Foundation (SQL/Foundation*), September, 2000
[CLI-WD]	*(ISO Working Draft) Database Language SQL - Part 3: Call-Level Interface (SQL/CLI*), September, 2000
[PSM-WD]	*(ISO Working Draft) Database Language SQL - Part 4: Persistent Stored Modules (SQL/PSM*), September, 2000
[TemporalWD]	*(ISO Working Draft) Database Language SQL - Part 7: Temporal (SQL/Temporal*), September, 2000
[MED-WD]	*(ISO Working Draft) Database Language SQL - Part 8: Management of External Data (SQL/MED*), September, 2000
[OLB-WD]	*(ISO Working Draft) Database Language SQL - Part 9: Object Language Bindings (SQL/OLB*), September, 2000
[SchemataWD]	*(ISO Working Draft) Database Language SQL - Part 10: Schemata (SQL/Schemata),* September, 2000

Other relevant International Standards and Drafts

[ISO10646]	*(Working paper of JTC1/SC2/WG2) ISO/IEC 10646-1, Universal Multiple-Octet Coded Character Set (UCS), Part 1:Architecture and Basic Multilingual Plane,* Second Edition text, Draft 2, ISO/IEC JTC1/SC2/WG2 N 2005, May 1999 (http://anubis.dkuug.dk/JTC1/SC2/WG2/docs/standards, under ISO/IEC 10646-1). Although this is, strictly, the authority (or will be when published), we have relied mostly on [Unicode3] and Unicode Technical Reports, for reasons explained in [W3C-CharMod], Section 3.4.2.
[ISO14651]	*ISO/IEC 14651:2000 – International string ordering and comparison – Method for comparing character strings and description of the common template tailorable ordering,* April 2000
[DIS15924]	*ISO/IEC DIS 15924 – Code for the representation of names of scripts,* May 2000 (http://www.egt.ie/standards/iso15924/document/index.html)

Industry standards

[Unicode3]	*The Unicode Standard, Version 3.0,* The Unicode Consortium, Addison Wesley Longman Publisher, Feb 2000

[CharSetNames] *Official Names for Character Sets*, IANA (Internet Assigned Numbers Authority) available at (ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets).

Unicode Technical Reports

All are available at (http://www.unicode.org/reports/). Most are also available, though not necessarily in the latest version, on the CD-ROM that comes with [Unicode3], as are helpful demonstrations of collation and normalization.

[UTR#10] Unicode Collation Algorithm, (Unicode Technical Report #10), Revision 5.0, November 1999

The relevance of this Unicode Technical Report is self-evident.

[UTR#15] Unicode Normalization Forms (Unicode Technical Report #15), Revision 18.0, November 1999

The relevance of this Unicode Technical Report is self-evident.

[UTR#17] Character Encoding Model (Unicode Technical Report #17), Revision 3.0, November 1999

Particularly useful for the clear distinction between the concepts denoted by the various terms such as glyph, grapheme, (abstract) character, code point and code unit.

[UTR#18] Regular Expression Guidelines, (Unicode Technical Report #18), Revision 5.0, November 1999

Contains the Unicode thinking on graphemes, both locale-dependent and locale-independent.

[UTR#19] UTF-32 (Unicode Technical Report #19), Revision 6.0, May 2000

A UTR that not only defines UTF-32 but also (in Revision 6.0) reports what appears to be the ultimate rapprochement between the Unicode Consortium and ISO/IEC JTC 1/SC 2.

[UTR#24] Script Names (Proposed Draft Unicode Technical Report #24), Revision 1.0, June 2000

A recent draft that proposes to assign every UCS code point to exactly one script, and to maintain a register thereof.

World Wide Web Consortium documents

The following are available from the W3 Web-site, at the URLs indicated:

[W3C-CharMod] Character Model for the World Wide Web, World Wide Web Consortium Working Draft 29-November-1999 (http://www.w3.org/TR/charmod)

A useful paper because it considers the issues that should be resolved in a standard way in a specific context.

[W3C-CharReq] Requirements for String Identity Matching and String Indexing, World Wide Web Consortium Working Draft 10-July-1998 (http://www.w3.org/TR/WD-charreq#2.10)

A precursor to [W3C-CharMod], specifying requirements.

[CharSetHarmful] *Character Set* Considered Harmful, D Connolly, Internet-Draft, May 1995 (http://www.w3.org/MarkUp/html-spec/charset-harmful)

An interesting paper that clarifies some of the confusion arising from the use of the term *character set*, and others.

Papers of ISO/IEC JTC 1/SC 32/WG 3

[BHX080] A Review of Some Possible Problems with SQL character features, Sykes, (WG3: BHX-080) June 2000

Possibly of historical interest.

[BHX117] Further discussion of issues raised in BHX-080, Sykes, (WG3: BHX-117) July 2000.

An elaboration of some of the issues raised in [BHX080], and further elaborated in the present paper.

[HEL001] Minutes of July 2000 meeting of ISO/IEC JTC 1/SC 32/WG 3, Warwick, England, (WG3:HEL001) July 2000

Includes record of the acceptance of possible problems.

[HEL-047] Hugh Darwen, *Problems with <collate clause>*, (WG 3: HEL-047) September 2000.

***** End of paper *****