WB

B-tree Implementation
Version 2b4

Roland Zito-Wolf and Aubrey Jaffer

This manual documents the WB B-tree implementation, version 2b4 released February 2020.

Table of Contents

1 Overview

1.1 Description

WB is a disk based (sorted) associative-array package providing C, SCM, Java, and C# libraries. These associative arrays consist of variable length (0.B to 255.B) keys and values. Functions are provided to:

- create, destroy, open and close disk-files and associative arrays;
- insert, delete, retrieve, find next, and find previous (with respect to dictionary order of keys); and
- The atomic '`put`' and '`rem`' operations allow associations to be used for process mutexs.
- apply functions, delete, or modify values over a range of consecutive key values.

The (database) disk files interoperate between the various language libraries. The interface to the SCM Scheme implementation supports longer data values and SLIB relational databases. WB, SCM, and SLIB are packages of the GNU project.

The WB implementation has a file size limit of 2^32 * block size (default 2048.B) = 2^43 bytes (8796.GB). WB routinely runs with databases of several hundred Megabytes. WB does its own memory and disk management and maintains a RAM cache of recently used blocks.

Multiple associative arrays can reside in one disk file. Simultaneous access to multiple disk files is supported. A structure checking and garbage collecting program and a viewer are provided. Compiled, WB occupies approximately 66 kilobytes.

WB is implemented using a variant of B-tree structure. B-trees give slower access than hashing but are dynamic and provide an efficient determination of successor and predecessor keys. All operations are O(log(n)) in the size of the database. B-trees are commonly used by database systems for implementing index structures. B-trees are optimized for using the minimum number of disk operations for large data structures. Prefix and suffix key compression are used for storage efficiency in WB.

1.2 History

The origination of B-trees is credited to [BM72] R. Bayer and E. McCreight in 1972.

Working at Holland Mark Martin between 1991 and 1993, Roland Zito-Wolf, Jonathan Finger, and I (Aubrey Jaffer) wrote the *Wanna B-tree* system.

Jonathan Finger wrote a MUMPS-like byte-coded interpreter *Sliced Bread* using WB. The integrated system was heavily used by Holland Mark Martin for the rest of the decade.

In 1994 I wrote a Scheme implementation of the relational model with an independent object-oriented base-table layer for SLIB:

`http://people.csail.mit.edu/jaffer/slib_6.html`

In 1996 Holland Mark Martin assigned the copyriht for WB to the Free Software Foundation. I released WB as a library for C and SCM. I also wrote `wbtab.scm`, a base-table interface enabling SLIB's relational database to be backed by WB.

In 2002 I added color dictionary relational databases to SLIB.

In 2003 I added *next* and *previous* operations to the SLIB relational package, and wrote `rwb-isam.scm` for WB.

In 2004 I wrote FreeSnell, a program to compute optical properties of multilayer thin-film coatings. At the core of FreeSnell is a rwb-isam spectral refractive-index database for over 300 materials.

In 2006 I decided to reimplement ClearMethods' Water language on top of WB. In 2007, in order to make Water available on the great majority of browsers and servers, Ravi Gorrepati adapted Schlep (the SCM to C translator) to make translators to Java and C#. He also ported the support files and test programs to Java and C#.

I continue to maintain WB. The most recent information about WB can be found on WB's *WWW* home page:

`http://people.csail.mit.edu/jaffer/WB`

1.3 File Organization

The source files for WB are written in the SCM dialect of Scheme:

`wbdefs.scm`
: SCM configuration definitions.

`segs.scm`
`handle.scm`
`blink.scm`
`prev.scm`
`del.scm`
`ents.scm`
`scan.scm`
`stats.scm`
: SCM code for WB-trees.

`blkio.scm`
: wimpy POSIX interface to the disk. Replace this if you have a more direct interface to the disk.

These files are translated into the C, C#, and Java targets by SCM scripts named *scm2c*, *scm2cs*, and *scm2java* respectively. The function and variable data types in the target languages are determined by pattern-matching the first-element strings in the associations *scm2c.typ*, *scm2cs.typ*, and *scm2java.typ* respectively.

Files translated to C are put into the `wb/` directory. Files translated to Java are put into the `wb/java/` directory. Files translated to C# are concatenated with `wb/csharp/Cssys.cs` and `wb/csharp/SchlepRT.cs` and written to `wb/csharp/Wb.cs`.

In the `Makefile`:

'`s2hfiles`'
: Derived *.h files for C.

'`s2cfiles`'
: Derived *.c files for C.

'`s2jfiles`'
: Derived java/*.java files for Java.

'`csharp/Wb.cs`'
: Single derived source file for C#.

WB comes with a C utility program for database files stored on disk.

`wbcheck` *path* [Program]

Checks the structure of the database named by *path* and reclaims temporary trees to the freelist.

Manifest

`wb.info`	documents the theory, data formats, and algorithms; the C and SCM interfaces to WB-tree.
`ChangeLog`	documents changes to the WB.
`example.scm`	example program using WB-tree in SCM.
`wbsys.h`	The primary C include file for using the WB layer is is `wbsys.h`, which includes several other files from the directory. `wbsys.h` also defines WB's internal data types.
`wbsys.c`	Shared data and low-level C accessors.
`wbsys.scm`	Shared data and low-level accessors for debugging in SCM.
`wbscm.c`	C code for the SCM interface to WB-trees.
`db.scm`	code for SCM interface when debugging in SCM.
`scm2c.scm`	SCM code which translates SCM code into C.
`scm2c.typ`	rules relating variable names to types in generated C.
`scm2cs.scm`	SCM code which translates SCM code into C#.
`scm2cs.typ`	rules relating variable names to types in generated C#.
`scm2java.scm`	SCM code which translates SCM code into Java.
`scm2java.typ`	rules relating variable names to types in generated Java.
`test.scm`	file for testing WB-tree system.
`test2.scm`	more tests for WB-tree system.

`Makefile`	Unix makefile
`VMSBUILD.COM`	command script for compiling under VMS.
`all.scm`	loads all the SCM files for debugging.
`wbtab.scm`	SCM code allowing WB to implement SLIB relational databases.
`rwb-isam.scm`	SCM code allowing WB to implement SLIB relational databases with numerical and lexicographic key collations.
`wbcheck.c`	program for checking, repairing, and garbage collecting WB-tree databases.
`wbview`	SCM script for displaying low-level WB database associations.

1.4 Installation

WB unpacks into a directory called '`wb`'.

If you plan to use WB with SCM, the directories `scm` and `wb` should be in the same directory. Doing '`make wbscm.so`' in the scm directory compiles a dynamically linkable object file from the WB C source. Including the '`-F wb`' option to an executable build compiles the WB interface into the executable. It is not necessary to compile anything in `wb` directory.

`make all` Compiles `libwb`, `wbscm.so`, `java/wb.jar`, `csharp/Wb.dll` and the `wbcheck` executable.

`make install`
Installs `libwb`, `wbscm.so`, `java/wb.jar`, and `wbcheck` in the `$(prefix)` tree, as assigned in the `Makefile`.

1.5 Building from Scheme Sources

Scheme Infrastructure

SCM source is available from: `http://groups.csail.mit.edu/mac/ftpdir/scm/scm-5f3.zip`

Also available as source RPM: `http://groups.csail.mit.edu/mac/ftpdir/scm/scm-5f3-1.src.rpm`

SLIB is a portable Scheme library which SCM uses: `http://groups.csail.mit.edu/mac/ftpdir/scm/slib-3b6.zip`

Also available as RPM: `http://groups.csail.mit.edu/mac/ftpdir/scm/slib-3b6-1.noarch.rpm`

Testing Scheme Source

From the wb directory, do '`scm all test`'. This will load the Scheme version of WB-tree with test code. Typing '`(main)`' will construct a test database `z` in this directory. If this runs without errors then you are ready to build the C code. Exit from scm with '`(quit)`'.

Regenerating C Sources.

```
make all
```

Testing Compiled DBSCM

Run '`scm -rwb -ltest`'. This should build the test database `z` much more quickly than before.

Type '`(quit)`' to exit from DBSCM. Now run '`./wbcheck z`'. This will check the structure of the database and collect temporary files. This should reclaim 52 blocks and report no errors. If you run it again, no blocks will be collected.

1.6 License

1.7 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
`http://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to

as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    “Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

    “CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

2 Theory

We used [SAGIV] as our starting point for concurrent b-tree algorithms. However, we made significant changes and (we feel) improvements. We have particularly tried to simplify the algorithms to reduce the implementation and especially debugging effort. There are also a lot of complex details involved in building a complete implementation. The goal of this document is to describe and explain the major design decisions and any subtleties that arose.

2.1 B-tree Structure and Access

2.1.1 Definitions

B+-tree

- is a structure of blocks linked by pointers
- is anchored by a special block called the root, and bounded by leaves
- has a unique path to each leaf, and all paths are equal length
- stores keys only at leaves, and stores reference values in other, internal, blocks
- guides key search, via the reference values, from the root to the leaves

block

- is either internal or a leaf, including the root block
- contains at most n entries and one extra pointer for some fixed n
- has no fewer than [n/2] entries, the root excepted

root block

- is a leaf when it is the only block in the tree and will then contain at least one entry
- must have at least 2 pointers to other blocks when it is internal

internal block

- contains entries consisting of a reference value and a pointer towards the leaves
- its entries point to data classified as greater than or equal to the corresponding reference value
- its extra pointer references data classified as less than the block's smallest reference value

leaf block

- contains entries consisting of a key value and a pointer to the storage location of data matching the key
- its extra pointer references the next leaf block in the tree ordering; leaves linked in this manner are neighbors

prefix compression

Instead of storing the full text of each key, store the length of match with the previous key (in this block) and the text of the key which doesn't match.

suffix compression

In non-leaf blocks, only store enough of the split-key to differentiate the two sequential blocks at the lower level.

2.1.2 Block Format

The first 20 bytes of the block header format is the same for all types of blocks.

0 `blk:id` The ID of this block.

4 `blk:top-id`
: The ID of the root block of this tree.

8 `blk:nxt-id`
: The ID of the next block in the chain (at same level as this one).

12 `blk:time`
: The 32-bit time/date when this block was last modified.

16 `blk:end`
: Two-byte length of data in block, including header.

18 `blk:level`
: Block level; 0 is leaf.

19 `blk:typ`
: Block type, one of:

- `dir-typ` Directory tree.
- `ind-typ` Other tree.
- `seq-typ` Sequential chain (unused).
- `frl-typ` Free-list chain.

20 Start of data for `SEQ-TYP` blocks. Data spans to `blk:end`.

20 `leaf-split-key`
22 Start of data for other block types. Data spans to `blk:end`.

2.1.3 Tree format

The WB-tree (the "Wanna B-tree")

We use the B-link tree format (see [SAGIV]). Each leaf block contains some number of (key,value) pairs; each non-leaf (ie. index) block contains (key, down-pointer) pairs. Each block contains one additional key value, the SPLIT key, that is strictly greater than any other key in the block. The tree is augmented by chaining together the nodes of each level of the tree in split-key order (the NEXT pointers).

We allow the tree to be missing any index-insertion or block-deletion updates so long as certain key-order invariants are maintained:

A Blocks are chained in order of increasing split key at each level, and all valid blocks appear in the chain.

B Split keys are unique at each level.

C Every DOWN pointer from level N+1 points to a block B at the next lower level N whose split-key S is less than or equal to the key S1 associated with the pointer.

C1 If the split key for block B (at level N) is strictly less than the key associated with the ptr from level N+1 (S < S1), then it must be the case that

a. a block B' with that split key S1 exists at level N;

b. B' is reachable from B by following NEXT pointers; and

c. no pointers to either B' nor any blocks between B and B' exist in level N+1. We call the subchain from B through B' the SPAN of the (key,ptr) entry (split(B'),B).

C2 It is invalid for a down pointer to contain a key not present as a split key at the next level.

Note that a key in an index must match its block's split key exactly; if a key K is less than the split-key S of the block B it points to, searches for intervening keys will be misdirected (to the next block); and if K is greater than S, then splits of the block after B at key values K' where S<K'<K will be mis-inserted in B's paren, because K' should logically go AFTER K, but K'<K.

The notion of the span of an index entry is useful. We note that each block split can be thought of as an EXTENSION of some span at the next-higher level, while each PARENT-INSERT-UPDATE can be thought of as a corresponding span REDUCTION. A span that has only one block in it will be called FULLY REDUCED; a b-tree is fully reduced when all its spans are fully reduced, meaning that all pending/deferred insert-updates have been performed. Lastly, we can express rule C1 more succinctly in terms of spans:

C,C2 SPANs must be well-formed (span must be closed; keys must match exactly)

C1 The SPANS of the entries at any index must not overlap.

Key/Value Order; Uniform Leaf/Node Format

We originally had index nodes in (value,key) order because it made insertions simple, but abandoned that because it made all the insertion/deletion propagation code special-case. In fact, because of the asymmetry of INSERT and DELETE, either one or the other will require that a single operation sometimes modify two blocks (see Section 2.1.5 [Insertion method], page 17). However, using (key,value) ordering at all levels of the tree simplified the code considerably.

2.1.4 Split keys

Split keys seem to be an accepted method for B-tree organization. In a Blink-tree, where one might potentially have to chain forward to find a desired key, the split key allows one to determine when the search can be terminated. Having the split key at the *end* of the block saves one block access per search. (If the split key were at the front of the block, one would have to access the next block in the chain to determine that a search could terminate.)

Last-key values

It is useful to define a *largest key* to be the split key of the last block at any given level; the code is made more complex if the last key is some special value (eg. the null string) that doesn't follow lexicographic order, not to mention the problems that arise with inserting into empty (last) blocks. We've reserved the set of strings starting with '`0xff`' as split keys only; they cannot be used as real key values. Last-key values are of the form xFF<level>. This makes the last key in level N+1 strictly larger than any in level N, so the end-of-chain key for level N need not be treated specially when it is inserted as a key at level N+1.

Fixed split keys/Independence of levels

We make the split keys at index levels act like those at the leaf level, that is, the split key in a block *NEVER CHANGES*. The advantage is that inserts and deletes cannot cause split-key changes, avoiding the need for code to propagate these changes, which are a pain to test. It also makes the levels genuinely independent, a useful conceptual simplification.

The disadvantage is that this introduces a *dead zone* between the last key-value pair of an index block and its split key. This complicates the code slightly, and makes the average search path slightly longer than log(N) (by a constant factor of 1+(1/BF), where BF is the branching factor of the tree). More annoyingly, it interacts with insertions for block splits (see INSERTION).

2.1.5 Insertion method

Insertion in a index needs to appear atomic so that the key-order invariant is maintained. Since we are using key/value-ordered index blocks, parent-update insertion has a special extra step: we insert the new key and pointer in index block B, then swap value fields with the next entry in sequence. Unfortunately, when the insertion is the end of the block, the next entry is in the next block B', so two blocks must be modified. (The advantage of value/key ordering is that insertion never modifies more than one block; however, you get screwed in the code for deletions instead.) The code checks for this case and locks both blocks before proceeding. (An alternate solution considered is to back up the split key of B so that the new key would go at the front of B', but that introduces other problems.)

While the above method preserves correctness (even under concurrency), there is unfortunately a 2nd-order screw WHICH STILL NEEDS TO BE IMPLEMENTED: how to write the 2 blocks in an order that preserves correctness in case of disk crash. If only B is written, the database will contain 2 pointers to the block whose split caused the insertion; if only B' is written, the pointer correctness will be violated. (It would seem that this kind of problem is inherent to any operation that needs to maintain consistent changes across multiple blocks.) The fix seems to be to surround the write with a notation in the root that says this special case is being exercised and indicates what blocks were being modified (B,B') and what the pointers should be:

- Write (B,B',down-ptrs) to Blk 0
- Write B'
- (Write new block if B was split by insert)

- Write B
- Remove flags from Blk 0 and write.

We elect to write out block B' first for consistency with the case where B' is a new block created by a block split. The screw case will create a damaged DB, but the danger of deleting a doubly-pointed-at block is avoided. This is adequate information to recover directly. (JONATHAN?)

(Other alternatives were considered. Backing up the split key at next level allows a subsequent insertion to be atomic, but the backing-up operation has the same two-block consistency problem in keeping ITS parent index consistent. If the parent key isn't backed up, subsequent splits of the following block will result in incorrect updates at the next level.)

2.1.6 Deletion

Deleting can be divided into two parts: the block to be deleted must be unlinked from both parent and predecessor, after which the block can be reclaimed.

Unlinking the block

At the moment, we've decided to simplify the delete operation by requiring that both the parent and the previous block be available for modification, else the deletion fails (is deferred). This makes block deletion an atomic change, which avoids several problems introduced by permitting partial deletions (see CONCURRENCY).

Alternatives: [WANG] gives a clever way to do deletion w/o PREV, by swapping blocks. This seemed too hard to make bulletproof for concurrency so we stuck with the PREV method, which after all only does extra block accesses 1/BF of the time.

Crashes during DELETE

What about the order for writing the parent and predecessor blocks in a delete? We elect to write out the PARENT block first, so that if the system should crash the chain will still be intact. The block that we were trying to delete will still be in the chain, but it cannot be used, it is "dead." Section 2.2.3 [Deferred Index Updates and concurrency], page 21, discusses how to deal with "dead" blocks.

Reclaiming the block

One major change from [SAGIV] is that SAGIV assumed multiple copies of a block could exist, which makes reclaiming deleted blocks complex (he suggests a timeout-based protocol). We opted instead to track how many pointers to each block are extant by adding a lock for that purpose, the NAME lock (essentially, a DELETE lock). A routine must get NAME lock on a pointer BEFORE releasing READ/WRITE on the block from which the pointer is obtained. (SAGIV's method is almost certainly more efficient in the sense that our method incurs overhead on every operation as opposed to just on the problem case of a READ request during a WRITE; several empirical studies of such tradeoffs support this conclusion.) On the other hand, NAME lock is useful for other things, such as insuring that the block you are PREVing from can't be deleted while you're looking for its parent or predecessor . . .

Non-symmetry of INSERT and DELETE

It is worth remembering that INSERT and DELETE are not symmetric, in the sense that a postponed insertion is NOT equivalent to deleting a (KEY,PTR) pair. The latter operation leaves a block whose pointer is missing unaccessible via the index, while the former leaves the block accessible through the NEXT pointer chain.

This asymmetry has been the death of more proposals for fixing various problems involving concurrency than I care to recall!

EXAMPLE:

1. Start with blk p with split key k at level n;
   - the index (level n+1) contains: . . . k,p, . . .
2. split p at k', adding block p';
   - the index should now contain . . . k',p,k,p', . . .
3. Now neither deleting p nor p' yields the original index contents:
   - If we delete p, the result is . . . k,p', . . .
   - If we delete p', the result is . . . k',p, . . .

Therefore, it is not possible for a subsequent delete to "cancel" an insert; the deletes must either wait for the relevant inserts to complete or else do special work to maintain correctness.

2.1.7 Non-delete of last block in the chain

We currently do not support deletion of the last block at any level, that is, the one with the end-of-block key. This is because this deletion requires special case to retain the end-of-block key. One way to achieve this is to copy forward the contents of the next-to-last block, and deleting that instead. [There are details to be worked out, eg. preservation of correctness during this operation.]

2.1.8 Prev

[DESCRIBE HOW IT WORKS]

SCREW CASE FIX UNDERSTOOD BUT NOT YET IMPLEMENTED: if down-pointers are missing to blocks immediately after the first block of a level, PREV will miss those blocks. The problem occurs when a PREV-BLOCK of block B at level N occurs, causing a PREV at level N+1. If down-pointers are missing, the block B' associated with B's split key may be some predecessor of B rather than B itself. In this case PREVing at level N+1 is wasteful but correct; it would require fewer block accesses to chain from B' than from PREV of that entry. However, if the B' entry is the FIRST at level N+1, PREV will erroneously conclude that it is at the start of the chain. It either has to (a) always look at B's entry at level N+1 to check that the current block's ptr is really up-to-date, or (b) just check when it hits the START-OF-CHAIN.

2.1.9 Root Block protocol

Root uniqueness

We guarantee that the root block number never changes and thus can be used as a unique identifier for a given WB-tree. Other systems provide a unique tree ID by introducing a level of indirection in root references; this is inefficient, as root references are frequent. When the root is split, we allocate a new block to hold the data that would have remained in the root block, then use the old root block for the new root. This does mean that one cannot depend on a block's ID being unchanged if it splits!

ROOT DELETE and Reducing number of levels in a tree

NOT IMPLEMENTED.

2.1.10 Other tree organizations

There are other possibilities for tree layouts. It is possible that some of these may simplify operations that in the current layout are complex, such as the insert-screw-case. Other possible choices include:

- split key at start rather than the end of the block;
- reversing the order of (key,value) pairs in the blocks;
- not using the split key
- using TWO split keys (one at each end of the block);
- running the chains backward rather than forward;

My (RJZ's) favorite alternate assumption is allowing multiple versions of blocks to exist, as in [SAGIV]. This would mean:

- processes would be allowed to maintain STATE in a block, such as current position, such as for successive NEXTs; on the other hand, NEXT would have to check for NEXT(X)<X and skip forward accordingly.
- Write/Read interlock overhead should be greatly reduced (as WRITEs do not have to wait for reads);
- the need for NAME locking is greatly reduced, since we no longer would be trying to count the processes that "know" about each block;
- reclaiming deleted blocks is harder;
- other issues to be evaluated: effect on rebalance, PREV, and the deferred-operation protocol we've developed.

Perhaps we can explore the interrelationships in some methodical way someday . . .

2.2 Concurrency

2.2.1 Name access (and deleted-block reclamation)

In order to be able to explicitly know when a block is safe to delete, we insist that a user must get NAME lock on a pointer BEFORE releasing READ/WRITE on the block from which the pointer is obtained. NAME lock is useful for other things, such as

- insuring that the block you are PREVing from can't be deleted while you're looking elsewhere;
- CHAIN-PUT uses it to insure that the block just split doesn't disappear during the call to PARENT-INSERT-UPDATE
- others?

2.2.2 Fail-out protocol/access conflict strategy

Blocked operations are at the moment simply going to fail with an error code of RETRY-ERR, meaning that they can be safely retried later. The current idea is to use this whenever a READ-WRITE conflict occurs. (This would not be necessary using SAGIV's method.) However, since various other lockout and wait conditions can occur – waits for block reads, waits on NAME locks, waits on interlocked operations – some such facility would be needed anyway, so it seems reasonable to try to use it to handle READ-WRITE blocking as well.

NOT REALLY IMPLEMENTED YET due to the complexity of reorganizing the code to pass up the appropriate information in all cases. (Top-level routines return these codes but internal routines don't really use it yet, so retry-ability isn't really there yet.)

2.2.3 Deferred Index Updates and concurrency

The basic strategy is to allow a key insert/delete that causes a block to split/become empty to complete, but to then queue up the parent-update/block delete on some process that will retry deferred updates in the background until they succeed.

The problem is that deletes contain two separate operations that can wait: the parent update and the predecessor update. The two updates can be done separately if the block is first marked "dead" so that no insertions into it can occur. Unfortunately, there is a class of rare and complex screw cases involving the correct ordering of deletes and inserts that happen to involve the same key. One might for example delete a block, deleting its key, and then subsequent splits can reintroduce and re-delete it. If operations at the index level are deferred, the ordering of these deferred operations determines whether the resulting tree is correct.

Making block-delete atomic (as defined above) greatly simplifies this process. The block being inserted/deleted can then serve as its own semaphore.

We have decided to adopt a lazy update strategy. That is, rather than keeping around queues of pending parent-updates, we just throw them away if they can't be executed immediately. We can get away with this because the updates for level N+1 can be reconstructed from the chain at level N.

Now, a deferred update only affects performance if the affected path is actually encountered: if we find we have to chain across two blocks, it means a parent update hasn't occurred yet; if we encounter an empty block, it means a delete update hasn't occurred. Our idea is to fix these "inefficiencies" on demand, that is, only when we run into one will we expend the effort to fix it. For the moment, assume ALL block deletes and insert updates are deferred.

The basic algorithm is:

a. If we have to chain forward in searching for a key, there must be pointer missing at the next level (deferred insert). So we attempt to insert it. Forward chaining to reach the NEXT and PREV of a key is normal and hence shouldn't cause update attempts.

b. Whenever we reach an empty block, we attempt to delete it, except for the case where we are doing an insert which should go in that block. The delete attempt will fail UNLESS the relevant parent updates are complete, that is, if and only if the block is within a fully reduced span.

One major concern has been that an I/O failure during a block delete can leave a "dead" block, that is, one which can't be reached from its parent level. It is still in a chain but there is no down pointer to it and no search can terminate at that node.The problem is that once a block becomes dead we need to prevent it from being inserted into or restored into use, because that could result in entries being out-of-order (suppose that, while the block's dead, some inserts of keys less than its split key occur. They'll be directed into the next block by the index, and be posted there. If we then restore the block, the key ordering will be incorrect.) But it turns out that we can prevent this by observing which B-tree operations can encounter which types of deferred-update situation, as follows:

If we think of the tree as a set of nested SPANS, where the SPAN of an index entry is the set of entries in the blocks it SPANS, we note that during operations using search only - GET,PUT,REM – the locus of search stays strictly within the SPAN of some entry in the root. We enforce that a block not be deleted until its parent pointers are completely updates, that is, its pointer has a span of exactly one block. Now suppose a block delete fails halfway, leaving a empty block without any parent pointer. Such a block is unreachable by FIND-NODE and hence by INSERT! This means that if INSERT encounters an empty block, it must be valid to insert into it.

This has several interesting consequences:

a. This means that only the operations that can possibly chain ACROSS spans have to worry about dead blocks: the NEXT and PREV operations. (What exactly is the effect on PREV?)
b. This means that "dead" blocks can't be deleted (unless we look for them specially). But they (a) are only created by a rare kind of event, and (b) won't hurt anything, so long as we arrange that NEXT and PREVs ignore the empty blocks (ie. ignore the value of their split keys).
c. The way we have defined deferred-delete reconstruction, if the block isn't within a fully-reduced span, the delete simply can't succeed. This means that there is no point in trying to delete a blank block when we're chaining though a span, that is, we should only try to delete empty blocks encountered (1) when following a DOWN pointer in FIND-ENT, or (2) due to a NEXT operation (or PREV?).

[Having realized this, we can actually let block-deletes be two-part operations (again). The only additional complexity is that then we'd need to implement a method for detecting dead blocks, that is, differentiating them from empty blocks in the middle of large spans, which we mustn't try to delete. We just check if the block is continued in some span! HOW TO DO THAT EFFICIENTLY?]

In detail, the method of detecting deferred operations goes like this:

1. RECONSTRUCTING DEFERRED PARENT-UPDATES (missing DOWN pointers):

   Whenever we have to follow the NEXT pointer of a block B (in FIND-NODE), we should attempt a PARENT-INSERT-UPDATE using the (key,value) pair (split(B),next(B)). FIND-NODE needs to be fixed to chain right rather than down when the "dead zone" is encountered; and it should not attempt a PARENT-INSERT-UPDATE in this case.

   PARENT-INSERT-UPDATE can fail if:

   a. some other process is already doing the update (the first one to lock the parent wins, the other will fail out);
   b. the key split(B) is already present (means a DELETE is pending – this shouldn't really occur, though);
   c. the update requires a block split but no blocks are available.

2. RECONSTRUCTING DEFERRED BLOCK DELETES:

   Whenever we reach an empty block in FIND-NODE or a NEXT/PREV operation, we'll attempt a PARENT-DELETE-UPDATE.

   PARENT-DELETE-UPDATE should fail when

   a. the key is found but points to a different block (meaning the containing span isn't fully reduced);
   b. the block is NAME-locked (this means its safe to leave name-locks around, the worst that can happen is they cause block deletes to be deferred some);
   c. someone else is doing a PARENT-DELETE-UPDATE on this block (this can be guaranteed by having the delete process first name-lock the block being deleted;
   d. some block needing to be modified (parent or prev) is locked.

3. We need to fix the code that does the NEXT-KEY operation to not stop at potentially-dead blocks. CHAIN-FIND need NOT ignore blank blocks.

   In practice, we'll want to trigger the update routines at the normal times as well, ie. try insert-updates after block splits and delete-updates after a block becomes empty.

   There are a number of new statistics we should keep; these include:

   - deferred PARENT-INSERT-UPDATEs (PUDs)
   - insert-updates that succeed
   - insert-updates that fail (measures overhead of the method)
   - insert-update failure rate

     deferred block deletes
   - deferred block deletes that succeed
   - deferred block deletes that fail (measures overhead of the method)
   - delete failure rate
   - count of dead blocks found (requires extra work)

   (The number of chain-forwards is also a measure of the overhead.)

   [Note: this mechanism also supports a simple queuing method: keep a list of the block numbers at which updates were deferred, and in times of low usage do FINDs to them, which will update them as a side-effect . . .]

[Other strategies were considered but were either significantly more complex or over-heady, or introduced unnecessary delays:

One hack is to serialize the queue of postponed index INSERT and DELETE operations by brute force: to do them in exactly the order they arrived in. Possibly simpler alternative method: sort by key and timestamp them!

I think we also decided that the interpreter could simply devote a process to each deferred operation, since we want to shift resources toward accomplishing the deferred operations if too many queue up.

[WANG] maintains a queue of deleted operation on the DESTINATION block. This has the disadvantage that whenever a block is split or merged the deferred-operation queues have to be split or merged as well – ugh!.]

2.3 Buffer, I/O, and Free-List Management

2.3.1 Reclaiming Buffers

age vs. level vs. dirtiness

This is a perennial nuisance – There is a complex system in place which hasn't really been evaluated or tuned, and there is also a proposal by Jonathan to simply use the first free (or free-able) buffer one finds.

2.3.2 update-access

It seems to me there was some case where it wasn't OK to just do a release to #f and an update from there – but I can't think of it offhand . . .

2.3.3 Deferred writes of data blocks

Note: this is a different sort of referral from the "deferred index updates" discussed earlier. Those deferred operations were correctness- preserving; these are not. The idea here is that we can reduce i/o traffic if we can safely lose a "small" number of updates.

Description and purpose

The general idea here is that we can reduce I/O traffic by deferring writes of leaf blocks, in the sense that the updates can be lost without compromising the structure of the database. This only works where the data updates in question are not critical to database or application. Currently, we always defer leaf updates – both PUTs and REMoves – to data trees, where a data tree is any tree not of type DIRECTORY. (DIRECTORY leaf blocks are written to disk after every update.) The idea is that the user application should have control over how often the data blocks are written.

Also, referral of PUTs and DELETES should be separately controllable. This feature is needed for example by the database itself in maintaining the free list: we can afford to defer INSERTS of deleted block, because the worst that could happen is that a free block gets lost. But DELETES from the free list must update immediately, else a block could be doubly-allocated.

Implementation

The handle field '`wcb`' has the following boolean values:

a. SAP: save block after PUTs
b. SAR: save block after REMOVEs
c. SAC: force block save after cached block changes (not currently implemented)

d. FAC: flush buffer entirely after cached block changes (not currently implemented – future functionality)

These bits are set as follows:

DIRECTORY
    SAP=SAR=1;

FREE LIST
    SAR=1; SAP=SAC=0;

USER DATA
    SAP=SAR=SAC=0;

The state of these bits can be changed at any time using `(han:set-wcb!` *`han new-bits`*`)`. Directory trees force `wcb-sap` and `wcb-sar` when opened or created. Also, `open_seg` forces the free list write control bits to be as shown above, regardless of the block type of the free-list.

Calling `flush-ents` flushes some modified blocks. It is thread safe and can be called by a timer interrupt.

2.3.4 Caching of last (leaf) block used

Works great

NOT IMPLEMENTED YET: We could actually be cleverer and cache the parents of every node, and even the PREVs, using the same TRY-GET-ENT protocol!

2.3.5 Multiple Read Access

Multiple READ access is not supported yet. READ-READ conflicts can occur. We seem to recall noting that this limitation kept us from encountering some other problem, whose identity is lost for the moment.

2.3.6 Free-block management

Outline:

- free list is implemented as a B tree (its root block is in block 2 of the file)
- with cache (free-list-cache, one per segment)
- cache gets filled (to 1/2 full) when it empty and a block is needed; it gets flushed (to 1/2 full) when its about to overflow.
- cache can be filled either from the free-list-tree or if that's empty, the file is extended.
- cache filling is now done efficiently using scan-delete. This also means that only one disk write will be done per fill (in most cases).
- cache filling is interlocked so only one process can be filling the cache at a time.
- currently cache emptying is inefficient as each write of of a block to the tree causes an I/O [deferred writes is intended to fix this]

Current problems:

Last week (1/8) we concluded that (all efforts to date notwithstanding) there were still failure cases in free-block management under concurrent operation, because of problems like:

- multiple processes can eat up however many blocks are in the cache, meaning we still haven't handled the case where a PUT to the free list causes a block split (necessitating a recursive call to FREE-BLOCK).
- Conversely, it is possible for the cache to become overfilled by a cache filling operation, if in the meantime OTHER processes fill the cache with deleted blocks.
- We need to be sure in general that concurrent fills and/or empties don't overfill or over-drain the cache.

On the positive side, we realized that we NEED NOT allow enough free blocks for a free-list insert to split the whole tree – any split except the leaf split can simply be postponed if there isn't a free block available at that time! (And similarly for any insert-updates that happen to be triggered by free-list accesses.)

Come to think of it, if we happen to run out of blocks during a free-list insert, its OK to let the insert fail, we just lose one disk block!! That may just be the answer!!

2.3.7 Buffer management routines

Two routines have been built for purging buffers.

FLUSH-BUFFER(ENT) writes out ENT if it is dirty and unlocked. It returns TERMINATED if ENT is locked, RETRYERR if the write is attempted and fails, and SUCCESS otherwise.

PURGE-BUFFER(ENT) writes out ENT if it is dirty and then frees up the buffer. This IGNORES the access status of the buffer, so it should not be called by users; it always returns SUCCESS.

Use (DO-SEG-BUFFERS SEG FUNC) to apply a function to all the buffers of a given segment; for example, (DO-SEG-BUFFERS SEG FLUSH-BUFFER) can be used to guarantee that segment SEG's disk file is up to date. DO-SEG-BUFFERS halts if FUNC returns other than SUCCESS; the result of FUNC is returned. SUCCESS is returned if all buffers have been successfully processed. To process all segments, use SEG = #f.

(CHECK-BUFFER ENT) checks that the buffer is written and unlocked, and repairs those that are not.

2.3.8 Other issues to document

- bucket-locking method
- treatment of access conflict conditions
- amnesia-ent
- caching criteria (eh?)

2.4 Error Handling

The problem: Calls need to return adequate information to distinguish:

1. restartable operations, for example
    - update-parent on insert

   - update-parent on delete
   - unlink for delete
2. non-restartable failures (eg. disk i/o error)
3. null results (eg. key not found)
4. "real values", eg. the length of a returned string, or an ENT pointer vs. NULL.

The solution: use a bounded range of negative values as "failure" codes, leaving the non-negative return values available as "success" codes. The canonical success code is 0, but a routine that needs to return a value (string length, ENT pointer) can do so and have that value interpreted as a "success" code as well.

There are "degrees" of failure, and the negative codes are partitioned according to increasingly severity, as follows:

value	C name	Meaning
0	success	successful execution
-1	notpres	successful execution, no data present or no change made
-2	terminated	failure, no damage, caller can retry operation
-10	retryerr	failure, no damage, caller can retry operation
-13	keyerr	failure, no damage, call was in error
-15	argerr	failure, no damage, call was in error
-20	noroom	failure, no damage, out of room in file
-30	typerr	failure, file or object was not of correct type
-40	ioerr	i/o error, DB may be damaged
-45	strangerr	internal error, DB may be damaged
-90	unkerr	placeholder code
-100	maxerr	

The first class represent operations that completed without error. The second class represent operations that failed to complete, but are guaranteed to leave the DB in a correct state and are retry-able (or easily correctable). The third class represent operations that failed to complete, did not damage the database, but are not easily fixable or restartable. The last class represent error conditions in which the DB was corrupted, or during which DB corruption was detected.

The predicate (ERR? code) returns #t if the return code is within the range NOTPRES-MAXERR; the predicate (REALERR? code) returns #t if CODE is an actual error, as opposed to a "not there" or "stop processing" message.

2.5 Longer Value Fields

The 256.B length limit for value strings was a barrier to many possible applications for WB. The `db:get` and `db:put!` procedures in `wbscm.c` work with value strings up to 64770.B in length (see Section 6.4 [SCM Record Operations], page 54).

For a value string of length L:

0.B <= L <= 255.B

  The value string is stored literally with given key.

256.B <= L <= 64770.B

  The first 255 bytes of the value string is stored literally with the given key. Successive 255 byte chunks of the value string are stored as L/255-1 sequential key-value pairs. The key for each chunk is the given key with the index byte 1 <= k <= 254 appended to it.

  Note that use of bt:scan is complicated by long value strings.

Proposed Extensions

The rest are proposed extensions to unlimited value string length.

256.B <= L <= *bsiz* - 20

  The b-tree value fields points to a type *datalong*[1] block, which contains the value.

  The *bsiz* argument to `make_seg()` or *block-size* argument to `make-seg` is the size of all WB blocks (pages) in that segment.

bsiz < L

  Value points to head of a chain or tree of datalong blocks. Retrieved value assembly limits practical size. Storage efficiency is good for L >> *bsiz*.

  An interesting variant is to have two trees, one for datalongs and the other for everything else. If the two trees are in separate segments (stored in separate files), then the datalong segment blocksize can be optimized without impacting speed for non-datalong operations.

bsiz < L

  Value designates external file containing data. storage efficiency is good for L >> *bsiz*. Retrieved value needs no assembly; use mmap().

2.6 Unlimited Length Keys and Values

by Jonathan Finger

Use 2 btrees which I will refer to as *main* and *slave*

notation D[123,4567] will translate into a string D \3 1 2 3 \4 4 5 6 7 where \3 is ascii 3 etc (length of following string of digits) this will result in keys sorting in numerical order.

[1] This option requires a new type of WB block having the usual 20-byte header and a 2-byte length field. Each block would hold up to *bsiz*-20 bytes of data.

1. Data (the easy part)

   The first byte of data is a flag.

   IF the data length < 255

   THEN

   store in bytes 1 - (length_of_data + 1)

   ELSE

   In slave btree increment value in key D (D for data)

   As an example assume the new value is 2357 and the data is 10,000 bytes

   ```
   Store the data in
   D[2357,0] = first 250 bytes
   D[2357,1] = second 250 bytes
     .
     .
     .
   D[2357,249] = last 250 bytes
   ```

   When a new value is set, if it is > 254 bytes in length then you reuse the D[2357] names adding or deleting as needed. Note that it is much more efficient to overwrite existing nodes since (most) of them will be the same size and add or delete at the end. If the new value < 255 bytes delete D[2357] and store the value in the master btree.
2. Keys (the hard part) IF key < 250 bytes

   THEN

   Store in master file as usual

   ELSE (key >= 250 bytes)

   break up the key into chunks C0 ... CN; the first 250 bytes long and the rest 220 bytes long.

   Create the entries (assume current value of key V is 243)

   ```
   Master btree
   C0 = (flag)244
   Slave btree
   V[244,C1] = 245
   V[245,C2] = 246
       .
       .
       .
   V[244 + N,CN] = data
   ```

   A get now consistes of breaking up the key and following the chain. The put, delete, next, and prev code will get a bit messy since there will be multiple cases to consider.

I believe this scheme will give unlimited key and data length. Performance probably will not be great but may be acceptable. This should give good key compression.

2.7 To Be Done

I think most of the items in RJZ's list have been done.

RJZ Modified 4/8/1993

B-tree maintenacne

- Implement deferred INDEX updates (in progress).
- Fix bug in PREV re missing down-ptrs.
- Implement fix for insert-screw-case (flag in root or wherever it was).

I/O

- Implement and test random page replacement.
- Implement parent/PREV caching??

Concurrency

- Assure enough FLC blocks for freelist splits in concurrent situation. (there seems to be a class of problems about concurrent free-list operations: a flush can fail if it causes a split and all the free blocks have been used meanwhile; simultaneous flush and fills; and the like. One good idea: since its ok if parent-updates fail, we can reduce the number of blocks a split can REQUIRE to 1 leaf block, rather than log-N blocks).

Error handling

- Finish implementation of error-handling protocol??
- Error log?
- Recode error msg calls to take less (code) space?

Miscellany

- Count SCAN does not need to copy value strings (since block is copied).
- Jonathan wants the ability to create a seg that need not be kept valid on disk – one that can just be rebuilt if the system crashes. This means (a) a WCB bit in HANDLEs to control "essential" updates and (b) an arg to OPEN-SEG specifying SAFETY (it isnt valid to specify this option on a per-handle basis).
- Jonathan wants to replace FLUSH-SOME-BUKS with a call that scans ENT-TAB for a flushable ENT and another call to (carefully) flush it.
- Reduce number of arguments to scan using packets?
- Create memory-resident segment?
- See if packets can be replaced with multiple values?

Design/Documentation

- Spell check!
- Document format of data file (blocks 0,1,2).
- Document type S blocks.
- Write up thoughts on error-info protocol.
- Write up WRITE order for blk splits.
- Read papers (SAGIV, WANG, TRANSACTION book) to see how THEY ahndle the difficult problems of delayed update and their ordering, and the 2-block mod problem for DELETE.

2.8 Miscellany

"Largest keys" (End-of-chain marker strings).

Because we've reserved keys with a first byte of FF for split keys, these keys are unavailable to the user.

NULL keys

WB supports use of the null string as a key (Sliced Bread treats the key specially).

Searching on minimum and maximum keys

Given that the null string may be used as a key, there needs to be a way to specify a "least" key such that NEXT(least) yields the first key, even if it is NULL. The special key strings with length s of -2 and -1 are provided to represent the minimum and maximum key values, respectively. These keys are only useful with NEXT, PREV, REM*, and SCAN; data cannot be associated with these keys.

Need to document TSCAN, TSTATS.

2.8.1 Bibliography

A comprehensive B-tree bibliography can be found at: `http://www.informatik.uni-trier.de/~ley/db/access/btree.html`

[BM72] R. Bayer and E. McCreight. *Organization and maintenance of large ordered indexes.* Acta Informatica, 1:173-189, 1972.

[SAGIV] Yehoshua Sagiv. *Concurrent Operations on B*-trees with Overtaking.* JCSS 33(2): 275-296 (1986)

[WANG] W.E. Weihl and P. Wang. *Multi-version memory: Software cache management for concurrent B-trees.* In Proc. 2nd IEEE Symp. Parallel and Distributed Processing, 650-655, 1990.

3 C Interface

3.1 C Compile-Time Parameters

The C Preprocessor Constants in this section are defined in `wbdefs.h` which is derived from `wbdefs.scm`.

Each segment (file-based or not) maintains a *freelist* of disk blocks. In a newly created segment, nearly all of its block numbers are in the freelist.

`flc_len` [C Preprocessor Constant]

> `flc_len` must be larger than 2 times the maximum number of blocks which would ever be needed for a freelist split. The minimum for `flc_len` is 10.

3.2 C Status Codes

`err_P` (*x*) [C Preprocessor Macro]

> Return *x* if a valid error code (-1 . . . *MAXERR*); else 0.

`success_P` (*x*) [C Preprocessor Macro]

> Not `err_P`.

`success` [C Preprocessor Constant]

> Successful execution (0).

Negative integers are used for errors according to increasingly severity, as follows:

`notpres` [C Preprocessor Constant]

> Successful execution; no data present or no change made.

`terminated` [C Preprocessor Constant]

> Failure; no damage; caller can retry operation.

`retryerr` [C Preprocessor Constant]

> Failure; no damage; caller can retry operation.

`keyerr` [C Preprocessor Constant]

> Failure, no damage, call was in error.

`argerr` [C Preprocessor Constant]

> Failure, no damage, call was in error.

`noroom` [C Preprocessor Constant]

> Failure, no damage, out of room in file.

`typerr` [C Preprocessor Constant]

> Failure, file or object was not of correct type.

`ioerr` [C Preprocessor Constant]

> I/O error, DB may be damaged.

`strangerr` [C Preprocessor Constant]

Internal error, DB may be damaged.

`unkerr` [C Preprocessor Constant]

Placeholder code.

`maxerr` [C Preprocessor Constant]

All error codes are between 0 and '`maxerr`'.

3.2.1 C Diagnostic Channel

The machine translated source utilizes `dprintf` as a platform-independent way to log diagnostic, warning, and error messages. `tdprintf`

`dprintf` ((*diagout*, *const char* ***`template`***, ...)) [C Preprocessor Macro]

The single argument to `dprintf` must be an argument list within parenthesis (eg. double parentheses). The first argument inside this list should be literally `diagout`. *template* is a *printf* style format string followed by the arguments for formatting, as with `printf`.

3.3 C SEGs

`int init_wb` (*int* ***`max_num_ents_cnt`***, *int* ***`max_num_buks`***, *int* ***`max_blk_size`***) [Function]

Initializes the WB system. `init_wb` should be called before any other WB functions.

max_blk_size
: The size of each disk cache buffer to be allocated from RAM. It should be an integer multiple of the file-system block size. The minimum is 1.5 kB.

max_num_ents_cnt
: The number of (RAM) disk cache buffers to be allocated. This should be proportional to the size of the database working set. If too small, then the disk will be kept busy writing and reading pages which get flushed to read in other pages. The minimum is 12 times the number of threads.

: The product of *max_num_ents_cnt* and *max_blk_size* should be less than the size of RAM on your computer.

max_num_buks
: The number of hash buckets for the (RAM) disk cache. It should not be less than *max_num_ents_cnt*. The minimum is 2, maybe 3 (due to how get-free-ent works).

If not all *max_num_ents_cnt* can be allocated (by malloc) then WB can still run properly. The number of buffers actually allocated is returned if successful; a status code is returned otherwise.

If the *bsiz* argument to `make-seg` is larger than the *max_blk_size* which was passed to `init_wb`, then the call to `make-seg` will fail.

`int final_wb ()` [Function]

Frees all memory used by the WB system. All segments will be closed.

To preserve database consistency, it is important to call `final_wb` or `close-seg` before program termination if changes have been made to a segment.

The *bsiz* of a segment (given in call to `make_seg`) is a parameter crucial for performance; balancing CPU time traversing blocks with file-system latency. *bsiz* should be an integer multiple of the file-system block size.

In the 1990s our nominal *bsiz* was 2.kiB; now it should probably be 4.kiB, 8.kiB, or 16.kiB.

`SEGD * open_seg` (*unsigned char * `filename`, int `mutable_P`*) [Function]

Opens the database file *filename* and returns a *seg*, false otherwise. The database will be read-only if the *mutable_P* argument is false. It will be read-write if the *mutable_P* argument is true.

`SEGD * open_segd` (*unsigned char * `filename`, int `mutable_P`, int `even_if_dirty_P`*) [Function]

Opens the database file *filename* and returns a *seg*, false otherwise. The database will be read-only if the *mutable_P* argument is false. It will be read-write if the *mutable_P* argument is true. If the *even_if_dirty_P* argument is false, then `open_segd` will fail if the database file *filename* was not closed cleanly; otherwise it will open, clean or not.

`int close_seg` (*SEGD * `seg`, int `hammer_P`*) [Function]

Closes database segment *seg* and the file containing it. If *hammer_P* is NULL, then if there are any problems freeing buffers, then the close is aborted. A status code is returned.

`SEGD * make_seg` (*unsigned char * `filename`, int `bsiz`*) [Function]

The integer *bsiz* specifies the size of B-tree blocks. *bsiz* should be an integer multiple of the file-system block size. Nominal value is 4096.

`make_seg` returns an open new empty mutable database named backed by file *filename* if successful; otherwise false is returned.

The write-control-bits argument (*wcb*) to these functions controls the latency of updates to the file after various operations. These bits are defined as follows:

value	C-name	Meaning
1	wcb_sap	save block after PUTs
2	wcb_sar	save block after REMOVEs
4	wcb_sac	force block save after cached block changes
8	wcb_fac	flush buffer entirely after cached block changes (not currently implemented)

`int bt_open` (*SEGD * `seg`, long `blk_num`, HAND * `han`, int `wcb`*) [Function]

Opens bt-handle *han* to seg number *seg*, block number *blk_num*, and returns the type of the block. If no such block exists or is not a root block, then a (negative) status code is returned.

`int bt_create` (*SEGD * **seg**, int **typ**, HAND * **han**, int **wcb***) [Function]

Creates a new root block in seg *seg* of type *typ*, opens bt-handle *han* to it, and returns a status code. If *seg* has insufficient room to create a new tree, then the *noroom* status code is returned.

`bt_create` can be used to create temporary b-trees. Temporary trees will be be reclaimed by check program after system crashes. In order to make a tree persistent, add it to a directory (tree).

`int bt_close` (*HAND * **han***) [Function]

Closes bt-handle *han* and returns *SUCCESS*.

Currently, `bt_close` has no effect other than to clear *han*.

3.4 C HANDs and Tree Operations

`void close_bt` (*HAND * **han***) [Function]

Closes and frees *han*; the b-tree associated with *han* remains in the segment.

For `create-db` and `open-db`, the implicit *WCB* argument is the combination of '`WCB-SAP`' and '`WCB-SAR`'.

`HAND * create_db` (*SEGD * **seg**, int **typ**, unsigned char * **name_str***) [Function]

Returns a B-tree whose name has been entered in the root directory if successful; otherwise null.

typ should be either

- `'D'` (directory) or
- `'T'` (regular tree).

B-trees with *typ* `#\D` which are pointed to by special entries in the root block (1) protect all their special entries from garbage collection by the `wbcheck` program. `'T'` is for regular (data) arrays.

`HAND * open_db` (*SEGD * **seg**, unsigned char * **name_str***) [Function]

Returns the B-tree whose name has been entered in the root directory; or null if not found.

`int flush_ents` (*int **attempts**, int **k***) [Function]

k is the number of dirty block buffers to write to disk; *attempts* is the number of times to try. Note that blocks in any segment may be written by `flush-ents`. `flush-ents` returns the number of blocks written.

Note: most of the data-manipulating commands here can return *notpres*, with the followng meanings:

`bt-get`	*key* was not found.
`bt-next`	no *next key* (eg, given *key* was last key).
`bt-prev`	no *prev key* (eg, given *key* was first key).
`bt-rem`	*key* was not found.
`bt-put`	*unused* (could be symmetric with write).
`bt-write`	*key was* found, so no write was done.

`int bt_get` (*HAND * `han`, unsigned char * `key_str`, int `k_len`, unsigned char * `ans_str`*) [Function]

key_str is a string of length *k_len*. `bt_get` stores into the string *ans_str* the value associated with *key_str* in tree *han*. `bt_get` returns the length of the string stored into *ans_str* or an error code.

`int bt_next` (*HAND * `han`, unsigned char * `key_str`, int `k_len`, unsigned char * `ans_str`*) [Function]

key_str is a string of length *k_len*. `bt_next` stores into the string *ans_str* the next key after *key_str* in tree *han*. `bt_next` returns the length of the string stored into *ans_str* or an error code.

`int bt_prev` (*HAND * `han`, unsigned char * `key_str`, int `k_len`, unsigned char * `ans_str`*) [Function]

key_str is a string of length *k_len*. `bt_prev` stores into the string *ans_str* the last key before *key_str* in tree *han*. `bt_prev` returns the length of the string stored into *ans_str* or an error code.

`int bt_rem` (*HAND * `han`, unsigned char * `key_str`, int `k_len`, unsigned char * `ans_str`*) [Function]

key_str is a string of length *k_len*. `bt_rem` stores into the string *ans_str* the value associated with *key_str* in tree *han*; then removes that association from tree *han*. `bt_rem` returns the length of the string stored into *ans_str* or an error code.

If *ans_str* is 0, `bt_rem` removes the *key_str* association from tree *han* and returns *SUCCESS* if successful; an error code if not.

`int bt_rem_range` (*HAND * `han`, unsigned char * `key_str`, int `k_len`, unsigned char * `key2_str`, int `k2_len`*) [Function]

key_str must be a maximum-length (256 byte) string containing a key *k_len* bytes long. *key2_str* is a string of length *k2_len*.

`bt_rem_range` removes [*key_str* . . . *key2_str*) and their values. If *key2_str* <= *key_str* no deletion will occur (even if *key_str* is found). `bt_rem_range` returns SUCCESS if the operation is complete, an error status code if not.

`int bt_put` (*HAND * `han`, unsigned char * `key_str`, int `k_len`, unsigned char * `val_str`, int `v_len`*) [Function]

key_str is a string of length *k_len*. *val_str* is a string of length *v_len*. `bt_put` makes the value associated with *key_str* be *val_str* in tree *han*. `bt_put` returns a status code for the operation.

`int bt_write` (*HAND * `han`, unsigned char * `key_str`, int `k_len`, unsigned char * `val_str`, int `v_len`*) [Function]

key_str is a string of length *k_len*. *val_str* is a string of length *v_len*. If *han* currently contains an association for *key_str*, then `bt_write` does not modify the tree and returns the *notpres* status code.

Otherwise, `bt_write` makes the value associated with *key_str* be *val_str* in tree *han*. `bt_write` returns a status code for the operation.

3.5 C Scan

`int bt_scan` (*HAND * `han`, int `operation`, unsigned char * `kstr1`, int `len1`, unsigned char * `kstr2`, int `len2`, int_function `func`, long * `long_tab`, int * `respkt`, int `blk_limit`*) [Function]

`bt_scan` scans all keys in the range [*kstr1..kstr2*), performing one of several functions:

operation	*func*	RESULT
COUNT-SCAN	NIL	counts all keys in range
COUNT-SCAN	given	counts all keys in range satisfying *func*
REM-SCAN	NIL	deletes all keys in range
REM-SCAN	given	deletes all keys in range satisfying *func*
MODIFY-SCAN	NIL	ARGERR
MODIFY-SCAN	given	updates values for keys in range satisfying *func*

`bt_scan` returns SUCCESS if scan completed; under any other result code the scan is resumable. The possible results are:

NOTPRES
: meaning the *blk_limit* was exceeded;

RETRYERR
: meaning *func* or delete got a RETRYERRR;

TERMINATED
: meaning *func* asked to terminate the scan;

<other error>
: means *func* or DELETE encountered this errror.

Each block of data is scanned/deleted/modified in a single operation that is, the block is found and locked only once, and only written after all modifications are made. Tho only exception is that MODIFY-SCANs that increase the size of values can cause block splits. Such cases are detected and converted to a PUT plus a NEXT. This has two consequences: data is written out each time a PUT occurs, and it is conceivable that *func* may be called more than once on the key value that caused the split if a RETRYERR occurs in the PUT. However, SCAN guarantees that only one modification will actually be made in this case (so that one can write INCREMENT-RANGE, for example).

func is passed pointers to (copies of) the key and value, plus one user argument:

```
func (keystr klen vstr vlen extra_arg);
```

func is expected to return either: SUCCESS for DELETE/COUNT, NOTPRES/NOTDONE for SKIP (ie, DONT DELETE/COUNT), or any other code to terminate the scan resumably at the current point. For MODIFY-SCAN, if changing the value, the new value length is returned. Except for the case mentioned above, the caller can depend on *func* being called exactly once for each key value in the specified range, and only on those values.

If *kstr2* <= *kstr1*, then no scan will occur (even if *kstr1* is found). To make possible bounded-time operation `bt_scan` will access at most *blk_limit* blocks at a time; if you dont care, give it -1 for *blk_limit*.

The number of keys deleted/counted/modified is returned in the `skey-count` field of *respkt*; the key to resume at is returned in *kstr1* (*which therefore needs to be 256 bytes long*); and the new key length is returned in the `skey-len` field of *respkt*. If returns SUCCESS, `skey-len` is zero. NOTE that `skey-count` is cumulative, so the caller needs to initialize it to 0 when starting a new `bt_scan`.

WARNING: when `bt_scan` returns other than SUCCESS, it modifies the *kstr1* string so that the string args are correctly set up for the next call (the returned value is the new length for *kstr1*). Therefore, *kstr1 must be a maximum-length string!*

4 Java Interface

4.1 Java Status Codes

`boolean err_P` (*int x*) [Function]

Return *x* if a valid error code (-1 . . . *MAXERR*); else 0.

`boolean success_P` (*int x*) [Function]

Not `err_P`.

`int success` [Variable]

Successful execution (0).

Negative integers are used for errors according to increasingly severity, as follows:

`int notpres` [Variable]

Successful execution; no data present or no change made.

`int terminated` [Variable]

Failure; no damage; caller can retry operation.

`int retryerr` [Variable]

Failure; no damage; caller can retry operation.

`int keyerr` [Variable]

Failure, no damage, call was in error.

`int argerr` [Variable]

Failure, no damage, call was in error.

`int noroom` [Variable]

Failure, no damage, out of room in file.

`int typerr` [Variable]

Failure, file or object was not of correct type.

`int ioerr` [Variable]

I/O error, DB may be damaged.

`int strangerr` [Variable]

Internal error, DB may be damaged.

`int unkerr` [Variable]

Placeholder code.

`int maxerr` [Variable]

All error codes are between 0 and ‘`maxerr`’.

4.2 Java SEGs

`int initWb` (*int* `maxNumEntsCnt`, *int* `maxNumBuks`, *int* `maxBlkSize`) [Function]

Initializes the WB system. `initWb` should be called before any other WB functions.

maxBlkSize
: The size of each disk cache buffer to be allocated from RAM. It should be an integer multiple of the file-system block size. The minimum is 1.5 kB.

maxNumEntsCnt
: The number of (RAM) disk cache buffers to be allocated. This should be proportional to the size of the database working set. If too small, then the disk will be kept busy writing and reading pages which get flushed to read in other pages. The minimum is 12 times the number of threads.

  The product of *maxNumEntsCnt* and *maxBlkSize* should be less than the size of RAM on your computer.

maxNumBuks
: The number of hash buckets for the (RAM) disk cache. It should not be less than *maxNumEntsCnt*. The minimum is 2, maybe 3 (due to how get-free-ent works).

If not all *maxNumEntsCnt* can be allocated (by malloc) then WB can still run properly. The number of buffers actually allocated is returned if successful; a status code is returned otherwise.

If the *bsiz* argument to `make-seg` is larger than the *maxBlkSize* which was passed to `initWb`, then the call to `make-seg` will fail.

`int finalWb` () [Function]

Frees all memory used by the WB system. All segments will be closed.

To preserve database consistency, it is important to call `finalWb` or `close-seg` before program termination if changes have been made to a segment.

The *bsiz* of a segment (given in call to `make_seg`) is a parameter crucial for performance; balancing CPU time traversing blocks with file-system latency. *bsiz* should be an integer multiple of the file-system block size.

In the 1990s our nominal *bsiz* was 2.kiB; now it should probably be 4.kiB, 8.kiB, or 16.kiB.

`Seg openSeg` (*String* `filename`, *boolean* `mutable_P`) [Function]

Opens the database file *filename* and returns a *seg*, false otherwise. The database will be read-only if the *mutable_P* argument is false. It will be read-write if the *mutable_P* argument is true.

`Seg openSegd` (*String* `filename`, *boolean* `mutable_P`, *boolean* `evenIfDirty_P`) [Function]

Opens the database file *filename* and returns a *seg*, false otherwise. The database will be read-only if the *mutable_P* argument is false. It will be read-write if the *mutable_P* argument is true. If the *evenIfDirty_P* argument is false, then `openSegd` will fail if the database file *filename* was not closed cleanly; otherwise it will open, clean or not.

`int closeSeg` (*Seg* `seg`, *boolean* `hammer_P`) [Function]

Closes database segment *seg* and the file containing it. If *hammer_P* is NULL, then if there are any problems freeing buffers, then the close is aborted. A status code is returned.

`Seg makeSeg` (*String* `filename`, *int* `bsiz`) [Function]

The integer *bsiz* specifies the size of B-tree blocks. *bsiz* should be an integer multiple of the file-system block size. Nominal value is 4096.

`makeSeg` returns an open new empty mutable database named backed by file *filename* if successful; otherwise false is returned.

The write-control-bits argument (*wcb*) to these functions controls the latency of updates to the file after various operations. These bits are defined as follows:

value	C-name	Meaning
1	wcb_sap	save block after PUTs
2	wcb_sar	save block after REMOVEs
4	wcb_sac	force block save after cached block changes
8	wcb_fac	flush buffer entirely after cached block changes (not currently implemented)

`int btOpen` (*Seg* `seg`, *int* `blkNum`, *Han* `han`, *int* `wcb`) [Function]

Opens bt-handle *han* to seg number *seg*, block number *blkNum*, and returns the type of the block. If no such block exists or is not a root block, then a (negative) status code is returned.

`int btCreate` (*Seg* `seg`, *int* `typ`, *Han* `han`, *int* `wcb`) [Function]

Creates a new root block in seg *seg* of type *typ*, opens bt-handle *han* to it, and returns a status code. If *seg* has insufficient room to create a new tree, then the *noroom* status code is returned.

`btCreate` can be used to create temporary b-trees. Temporary trees will be be reclaimed by check program after system crashes. In order to make a tree persistent, add it to a directory (tree).

`int btClose` (*Han* `han`) [Function]

Closes bt-handle *han* and returns *SUCCESS*.

Currently, `btClose` has no effect other than to clear *han*.

4.3 Java HANDs and Tree Operations

All of the methods listed here which take byte-array arguments can also take string arguments, which get converted to UTF-8 byte-arrays.

`void closeBt` (*Han* `han`) [Function]

Closes *han*

For `create-db` and `open-db`, the implicit *WCB* argument is the combination of '`WCB-SAP`' and '`WCB-SAR`'.

`Han` **`createDb`** (*Seg* ***`seg`***, *int* ***`typ`***, *byte* []***`nameStr`***) [Function]

Returns a B-tree whose name has been entered in the root directory if successful; otherwise null.

typ should be either

- `'D'` (directory) or
- `'T'` (regular tree).

B-trees with *typ* `#\D` which are pointed to by special entries in the root block (1) protect all their special entries from garbage collection by the `wbcheck` program. `'T'` is for regular (data) arrays.

`Han` **`openDb`** (*Seg* ***`seg`***, *byte* []***`nameStr`***) [Function]

Returns the B-tree whose name has been entered in the root directory; or null if not found.

`int` **`flushEnts`** (*int* ***`attempts`***, *int* ***`k`***) [Function]

k is the number of dirty block buffers to write to disk; *attempts* is the number of times to try. Note that blocks in any segment may be written by `flush-ents`. `flush-ents` returns the number of blocks written.

4.4 Record Operations

`byte []`**`bt_Get`** (*Han* ***`han`***, *byte* []***`key`***) [Function]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`bt:get` returns a string of the value associated with *key* in the bt which *han* is open to. `bt:get` returns null if *key* is not associated in the bt.

`byte []`**`bt_Next`** (*Han* ***`han`***, *byte* []***`key`***) [Function]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`bt:next` returns the next *key* in bt *han* or null if none.

`byte []`**`bt_Prev`** (*Han* ***`han`***, *byte* []***`key`***) [Function]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`bt:prev` returns the previous *key* in bt *han* or null if none.

`void` **`bt_Put`** (*Han* ***`han`***, *byte* []***`key`***, *byte* []***`valStr`***) [Function]

han is a handle to an open, mutable bt. *key* and *val* are strings less than 255.B in length.

`bt:put!` associates *key* with *val* in the bt *han*. A status code is returned.

`boolean` **`bt_Del`** (*Han* ***`han`***, *byte* []***`key`***) [Function]

han is a handle to an open, mutable bt. *key* is a string less than 255.B in length.

`bt:rem!` removes *key* and it's associated value from bt *han*.

4.5 Mutual Exclusion

These 2 calls can be used for locking and synchronizing processes.

`boolean bt_Insert` (*Han* `han`, *byte* []`key`, *byte* []`valStr`) [Function]

> Associates *key* with *val* in the bt *han* only if *key* was previously empty. Returns true for success, false for failure.

`byte []bt_Rem` (*Han* `han`, *byte* []`key`) [Function]

> Removes *key* and it's associated value from bt *han* only if *key* is present. Returns *key*'s value for success, null for failure (not present).

4.6 Multiple Operations

`int bt_Delete` (*Han* `han`, *byte* []`key`, *byte* []`key2`) [Function]

> Removes *key*s (and their associated values) between (including) *key1* and (not including) *key2* from bt *han*. A status code is returned.

`byte []bt_Scan` (*Han* `bthan`, *int* `op`, *byte* []`kstr1`, *byte* []`kstr2`, *java.lang.reflect.Method* `func`, *int* `blklimit`) [Function]

> `btScan` scans all keys in the range [*kstr1*..*kstr2*), performing one of several functions:

operation	*func*	RESULT
COUNT-SCAN	NIL	counts all keys in range
COUNT-SCAN	given	counts all keys in range satisfying *func*
REM-SCAN	NIL	deletes all keys in range
REM-SCAN	given	deletes all keys in range satisfying *func*
MODIFY-SCAN	NIL	ARGERR
MODIFY-SCAN	given	updates values for keys in range satisfying *func*

> `btScan` returns null if there was an error; an empty byte-vector if scan completed; or the next key to be scanned if *blklimit* was not '`-1`'.

> Each block of data is scanned/deleted/modified in a single operation that is, the block is found and locked only once, and only written after all modifications are made. Tho only exception is that MODIFY-SCANs that increase the size of values can cause block splits. Such cases are detected and converted to a PUT plus a NEXT. This has two consequences: data is written out each time a PUT occurs, and it is conceivable that *func* may be called more than once on the key value that caused the split if a RETRYERR occurs in the PUT. However, SCAN guarantees that only one modification will actually be made in this case (so that one can write INCREMENT-RANGE, for example).

> *func* is passed pointers to (copies of) the key and value, plus one user argument:

```
func (keystr, klen, vstr, vlen, extra_arg);
```

> *func* is expected to return either: SUCCESS for DELETE/COUNT, NOTPRES/NOTDONE for SKIP (ie, DONT DELETE/COUNT), or any other code to terminate the scan resumably at the current point. For MODIFY-SCAN, if changing the value, the new value length is returned. Except for the case mentioned above, the caller can depend on *func* being called exactly once for each key value in the specified range, and only on those values.

If *kstr2* <= *kstr1*, then no scan will occur (even if *kstr1* is found). To make possible bounded-time operation `btScan` will access at most *blkLimit* blocks at a time; if you dont care, give it -1 for *blkLimit*.

4.7 Java Legacy API

This API has identical argument configurations as the C code, including length arguments to complement each byte-vector argument.

Note: most of the data-manipulating commands here can return *notpres*, with the followng meanings:

`bt-get`	*key* was not found.
`bt-next`	no *next key* (eg, given *key* was last key).
`bt-prev`	no *prev key* (eg, given *key* was first key).
`bt-rem`	*key* was not found.
`bt-put`	*unused* (could be symmetric with write).
`bt-write`	*key was* found, so no write was done.

`int btGet` (*Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`ansStr`**) [Function]

keyStr is a string of length *kLen*. `btGet` stores into the string *ansStr* the value associated with *keyStr* in tree *han*. `btGet` returns the length of the string stored into *ansStr* or an error code.

`int btNext` (*Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`ansStr`**) [Function]

keyStr is a string of length *kLen*. `btNext` stores into the string *ansStr* the next key after *keyStr* in tree *han*. `btNext` returns the length of the string stored into *ansStr* or an error code.

`int btPrev` (*Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`ansStr`**) [Function]

keyStr is a string of length *kLen*. `btPrev` stores into the string *ansStr* the last key before *keyStr* in tree *han*. `btPrev` returns the length of the string stored into *ansStr* or an error code.

`int btRem` (*Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`ansStr`**) [Function]

keyStr is a string of length *kLen*. `btRem` stores into the string *ansStr* the value associated with *keyStr* in tree *han*; then removes that association from tree *han*. `btRem` returns the length of the string stored into *ansStr* or an error code.

If *ansStr* is 0, `btRem` removes the *keyStr* association from tree *han* and returns *SUCCESS* if successful; an error code if not.

`int btRemRange` (*Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`key2Str`**, *int* **`k2Len`**) [Function]

keyStr must be a maximum-length (256 byte) string containing a key *kLen* bytes long. *key2Str* is a string of length *k2Len*.

`btRemRange` removes [*keyStr* . . . *key2Str*) and their values. If *key2Str* <= *keyStr* no deletion will occur (even if *keyStr* is found). `btRemRange` returns SUCCESS if the operation is complete, an error status code if not.

`int btPut` (*Han* ***han***, *byte* []***keyStr***, *int* ***kLen***, *byte* []***valStr***, *int* ***vLen***) [Function]

keyStr is a string of length *kLen*. *valStr* is a string of length *vLen*. `btPut` makes the value associated with *keyStr* be *valStr* in tree *han*. `btPut` returns a status code for the operation.

`int btWrite` (*Han* ***han***, *byte* []***keyStr***, *int* ***kLen***, *byte* []***valStr***, *int* ***vLen***) [Function]

keyStr is a string of length *kLen*. *valStr* is a string of length *vLen*. If *han* currently contains an association for *keyStr*, then `btWrite` does not modify the tree and returns the *notpres* status code.

Otherwise, `btWrite` makes the value associated with *keyStr* be *valStr* in tree *han*. `btWrite` returns a status code for the operation.

`int btScan` (*Han* ***han***, *int* ***operation***, *byte* []***kstr1***, *int* ***len1***, *byte* []***kstr2***, *int* ***len2***, *java.lang.reflect.Method* ***func***, *int* []***longTab***, *int* []***respkt***, *int* ***blkLimit***) [Function]

`btScan` scans all keys in the range [*kstr1*..*kstr2*), performing one of several functions:

operation	*func*	RESULT
COUNT-SCAN	NIL	counts all keys in range
COUNT-SCAN	given	counts all keys in range satisfying *func*
REM-SCAN	NIL	deletes all keys in range
REM-SCAN	given	deletes all keys in range satisfying *func*
MODIFY-SCAN	NIL	ARGERR
MODIFY-SCAN	given	updates values for keys in range satisfying *func*

`btScan` returns SUCCESS if scan completed; under any other result code the scan is resumable. The possible results are:

NOTPRES
: meaning the *blkLimit* was exceeded;

RETRYERR
: meaning *func* or delete got a RETRYERRR;

TERMINATED
: meaning *func* asked to terminate the scan;

<other error>
: means *func* or DELETE encountered this errror.

Each block of data is scanned/deleted/modified in a single operation that is, the block is found and locked only once, and only written after all modifications are made. Tho only exception is that MODIFY-SCANs that increase the size of values can cause block splits. Such cases are detected and converted to a PUT plus a NEXT. This has two consequences: data is written out each time a PUT occurs, and it is conceivable that *func* may be called more than once on the key value that caused the split if a RETRYERR occurs in the PUT. However, SCAN guarantees that only one modification will actually be made in this case (so that one can write INCREMENT-RANGE, for example).

func is passed pointers to (copies of) the key and value, plus one user argument:

```
    func (keystr klen vstr vlen extra_arg);
```

func is expected to return either: SUCCESS for DELETE/COUNT, NOTPRES/NOTDONE for SKIP (ie, DONT DELETE/COUNT), or any other code to terminate the scan resumably at the current point. For MODIFY-SCAN, if changing the value, the new value length is returned. Except for the case mentioned above, the caller can depend on *func* being called exactly once for each key value in the specified range, and only on those values.

If *kstr2* `<=` *kstr1*, then no scan will occur (even if *kstr1* is found). To make possible bounded-time operation `btScan` will access at most *blkLimit* blocks at a time; if you dont care, give it -1 for *blkLimit*.

The number of keys deleted/counted/modified is returned in the `skey-count` field of *respkt*; the key to resume at is returned in *kstr1* (*which therefore needs to be 256 bytes long*); and the new key length is returned in the `skey-len` field of *respkt*. If returns SUCCESS, `skey-len` is zero. NOTE that `skey-count` is cumulative, so the caller needs to initialize it to 0 when starting a new `btScan`.

WARNING: when `btScan` returns other than SUCCESS, it modifies the *kstr1* string so that the string args are correctly set up for the next call (the returned value is the new length for *kstr1*). Therefore, *kstr1 must be a maximum-length string!*

5 C# Interface

5.1 C# Status Codes

`bool err_P` (*int x*) [Function]

Return *x* if a valid error code (-1 . . . *MAXERR*); else 0.

`bool success_P` (*int x*) [Function]

Not `err_P`.

`int success` [Variable]

Successful execution (0).

Negative integers are used for errors according to increasingly severity, as follows:

`int notpres` [Variable]

Successful execution; no data present or no change made.

`int terminated` [Variable]

Failure; no damage; caller can retry operation.

`int retryerr` [Variable]

Failure; no damage; caller can retry operation.

`int keyerr` [Variable]

Failure, no damage, call was in error.

`int argerr` [Variable]

Failure, no damage, call was in error.

`int noroom` [Variable]

Failure, no damage, out of room in file.

`int typerr` [Variable]

Failure, file or object was not of correct type.

`int ioerr` [Variable]

I/O error, DB may be damaged.

`int strangerr` [Variable]

Internal error, DB may be damaged.

`int unkerr` [Variable]

Placeholder code.

`int maxerr` [Variable]

All error codes are between 0 and '`maxerr`'.

5.2 C# SEGs

`int initWb` (*int* `maxNumEntsCnt`, *int* `maxNumBuks`, *int* `maxBlkSize`) [Function]

Initializes the WB system. `initWb` should be called before any other WB functions.

maxBlkSize
: The size of each disk cache buffer to be allocated from RAM. It should be an integer multiple of the file-system block size. The minimum is 1.5 kB.

maxNumEntsCnt
: The number of (RAM) disk cache buffers to be allocated. This should be proportional to the size of the database working set. If too small, then the disk will be kept busy writing and reading pages which get flushed to read in other pages. The minimum is 12 times the number of threads.

  The product of *maxNumEntsCnt* and *maxBlkSize* should be less than the size of RAM on your computer.

maxNumBuks
: The number of hash buckets for the (RAM) disk cache. It should not be less than *maxNumEntsCnt*. The minimum is 2, maybe 3 (due to how get-free-ent works).

If not all *maxNumEntsCnt* can be allocated (by malloc) then WB can still run properly. The number of buffers actually allocated is returned if successful; a status code is returned otherwise.

If the *bsiz* argument to `make-seg` is larger than the *maxBlkSize* which was passed to `initWb`, then the call to `make-seg` will fail.

`int finalWb` () [Function]

Frees all memory used by the WB system. All segments will be closed.

To preserve database consistency, it is important to call `finalWb` or `close-seg` before program termination if changes have been made to a segment.

The *bsiz* of a segment (given in call to `make_seg`) is a parameter crucial for performance; balancing CPU time traversing blocks with file-system latency. *bsiz* should be an integer multiple of the file-system block size.

In the 1990s our nominal *bsiz* was 2.kiB; now it should probably be 4.kiB, 8.kiB, or 16.kiB.

`wb.Seg openSeg` (*String* `filename`, *bool* `mutable_P`) [Function]

Opens the database file *filename* and returns a *seg*, false otherwise. The database will be read-only if the *mutable_P* argument is false. It will be read-write if the *mutable_P* argument is true.

`wb.Seg openSegd` (*String* `filename`, *bool* `mutable_P`, *bool* `evenIfDirty_P`) [Function]

Opens the database file *filename* and returns a *seg*, false otherwise. The database will be read-only if the *mutable_P* argument is false. It will be read-write if the *mutable_P* argument is true. If the *evenIfDirty_P* argument is false, then `openSegd` will fail if the database file *filename* was not closed cleanly; otherwise it will open, clean or not.

`int closeSeg` (*wb.Seg* `seg`, *bool* `hammer_P`) [Function]

Closes database segment *seg* and the file containing it. If *hammer_P* is NULL, then if there are any problems freeing buffers, then the close is aborted. A status code is returned.

`wb.Seg makeSeg` (*String* `filename`, *int* `bsiz`) [Function]

The integer *bsiz* specifies the size of B-tree blocks. *bsiz* should be an integer multiple of the file-system block size. Nominal value is 4096.

`makeSeg` returns an open new empty mutable database named backed by file *filename* if successful; otherwise false is returned.

The write-control-bits argument (*wcb*) to these functions controls the latency of updates to the file after various operations. These bits are defined as follows:

value	C-name	Meaning
1	wcb_sap	save block after PUTs
2	wcb_sar	save block after REMOVEs
4	wcb_sac	force block save after cached block changes
8	wcb_fac	flush buffer entirely after cached block changes (not currently implemented)

`int btOpen` (*wb.Seg* `seg`, *int* `blkNum`, *wb.Han* `han`, *int* `wcb`) [Function]

Opens bt-handle *han* to seg number *seg*, block number *blkNum*, and returns the type of the block. If no such block exists or is not a root block, then a (negative) status code is returned.

`int btCreate` (*wb.Seg* `seg`, *int* `typ`, *wb.Han* `han`, *int* `wcb`) [Function]

Creates a new root block in seg *seg* of type *typ*, opens bt-handle *han* to it, and returns a status code. If *seg* has insufficient room to create a new tree, then the *noroom* status code is returned.

`btCreate` can be used to create temporary b-trees. Temporary trees will be be reclaimed by check program after system crashes. In order to make a tree persistent, add it to a directory (tree).

`int btClose` (*wb.Han* `han`) [Function]

Closes bt-handle *han* and returns *SUCCESS*.

Currently, `btClose` has no effect other than to clear *han*.

5.3 C# HANDs and Tree Operations

All of the methods listed here which take byte-array arguments can also take string arguments, which get converted to UTF-8 byte-arrays.

`void closeBt` (*wb.Han* `han`) [Function]

Closes *han*

For `create-db` and `open-db`, the implicit *WCB* argument is the combination of '`WCB-SAP`' and '`WCB-SAR`'.

`Han createDb` (*wb.Seg* **`seg`**, *int* **`typ`**, *byte* []**`nameStr`**) [Function]

Returns a B-tree whose name has been entered in the root directory if successful; otherwise null.

typ should be either

- `'D'` (directory) or
- `'T'` (regular tree).

B-trees with *typ* `#\D` which are pointed to by special entries in the root block (1) protect all their special entries from garbage collection by the `wbcheck` program. `'T'` is for regular (data) arrays.

`Han openDb` (*wb.Seg* **`seg`**, *byte* []**`nameStr`**) [Function]

Returns the B-tree whose name has been entered in the root directory; or null if not found.

`int flushEnts` (*int* **`attempts`**, *int* **`k`**) [Function]

k is the number of dirty block buffers to write to disk; *attempts* is the number of times to try. Note that blocks in any segment may be written by `flush-ents`. `flush-ents` returns the number of blocks written.

5.4 Record Operations

`byte []bt_Get` (*wb.Han* **`han`**, *byte* []**`key`**) [Function]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`bt:get` returns a string of the value associated with *key* in the bt which *han* is open to. `bt:get` returns null if *key* is not associated in the bt.

`byte []bt_Next` (*wb.Han* **`han`**, *byte* []**`key`**) [Function]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`bt:next` returns the next *key* in bt *han* or null if none.

`byte []bt_Prev` (*wb.Han* **`han`**, *byte* []**`key`**) [Function]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`bt:prev` returns the previous *key* in bt *han* or null if none.

`void bt_Put` (*wb.Han* **`han`**, *byte* []**`key`**, *byte* []**`valStr`**) [Function]

han is a handle to an open, mutable bt. *key* and *val* are strings less than 255.B in length.

`bt:put!` associates *key* with *val* in the bt *han*. A status code is returned.

`bool bt_Del` (*wb.Han* **`han`**, *byte* []**`key`**) [Function]

han is a handle to an open, mutable bt. *key* is a string less than 255.B in length.

`bt:rem!` removes *key* and it's associated value from bt *han*.

5.5 Mutual Exclusion

These 2 calls can be used for locking and synchronizing processes.

`bool bt_Insert` (*wb.Han* **`han`**, *byte* []**`key`**, *byte* []**`valStr`**) [Function]

Associates *key* with *val* in the bt *han* only if *key* was previously empty. Returns true for success, false for failure.

`byte []bt_Rem` (*wb.Han* **`han`**, *byte* []**`key`**) [Function]

Removes *key* and it's associated value from bt *han* only if *key* is present. Returns *key*'s value for success, null for failure (not present).

5.6 Multiple Operations

`int bt_Delete` (*wb.Han* **`han`**, *byte* []**`key`**, *byte* []**`key2`**) [Function]

Removes *keys* (and their associated values) between (including) *key1* and (not including) *key2* from bt *han*. A status code is returned.

`byte []bt_Scan` (*wb.Han* **`bthan`**, *int* **`op`**, *byte* []**`kstr1`**, *byte* []**`kstr2`**, *string* **`func`**, *int* **`blklimit`**) [Function]

`btScan` scans all keys in the range [*kstr1*..*kstr2*), performing one of several functions:

operation	*func*	RESULT
COUNT-SCAN	NIL	counts all keys in range
COUNT-SCAN	given	counts all keys in range satisfying *func*
REM-SCAN	NIL	deletes all keys in range
REM-SCAN	given	deletes all keys in range satisfying *func*
MODIFY-SCAN	NIL	ARGERR
MODIFY-SCAN	given	updates values for keys in range satisfying *func*

`btScan` returns null if there was an error; an empty byte-vector if scan completed; or the next key to be scanned if *blklimit* was not '`-1`'.

Each block of data is scanned/deleted/modified in a single operation that is, the block is found and locked only once, and only written after all modifications are made. Tho only exception is that MODIFY-SCANs that increase the size of values can cause block splits. Such cases are detected and converted to a PUT plus a NEXT. This has two consequences: data is written out each time a PUT occurs, and it is conceivable that *func* may be called more than once on the key value that caused the split if a RETRYERR occurs in the PUT. However, SCAN guarantees that only one modification will actually be made in this case (so that one can write INCREMENT-RANGE, for example).

func is passed pointers to (copies of) the key and value, plus one user argument:

```
func (keystr, klen, vstr, vlen, extra_arg);
```

func is expected to return either: SUCCESS for DELETE/COUNT, NOTPRES/NOTDONE for SKIP (ie, DONT DELETE/COUNT), or any other code to terminate the scan resumably at the current point. For MODIFY-SCAN, if changing the value, the new value length is returned. Except for the case mentioned above, the caller can depend on *func* being called exactly once for each key value in the specified range, and only on those values.

If *kstr2* <= *kstr1*, then no scan will occur (even if *kstr1* is found). To make possible bounded-time operation `btScan` will access at most *blkLimit* blocks at a time; if you dont care, give it -1 for *blkLimit*.

5.7 C# Legacy API

This API has identical argument configurations as the C code, including length arguments to complement each byte-vector argument.

Note: most of the data-manipulating commands here can return *notpres*, with the followng meanings:

`bt-get`	*key* was not found.
`bt-next`	no *next key* (eg, given *key* was last key).
`bt-prev`	no *prev key* (eg, given *key* was first key).
`bt-rem`	*key* was not found.
`bt-put`	*unused* (could be symmetric with write).
`bt-write`	*key was* found, so no write was done.

`int btGet` (*wb.Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`ansStr`**) [Function]

keyStr is a string of length *kLen*. `btGet` stores into the string *ansStr* the value associated with *keyStr* in tree *han*. `btGet` returns the length of the string stored into *ansStr* or an error code.

`int btNext` (*wb.Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`ansStr`**) [Function]

keyStr is a string of length *kLen*. `btNext` stores into the string *ansStr* the next key after *keyStr* in tree *han*. `btNext` returns the length of the string stored into *ansStr* or an error code.

`int btPrev` (*wb.Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`ansStr`**) [Function]

keyStr is a string of length *kLen*. `btPrev` stores into the string *ansStr* the last key before *keyStr* in tree *han*. `btPrev` returns the length of the string stored into *ansStr* or an error code.

`int btRem` (*wb.Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`ansStr`**) [Function]

keyStr is a string of length *kLen*. `btRem` stores into the string *ansStr* the value associated with *keyStr* in tree *han*; then removes that association from tree *han*. `btRem` returns the length of the string stored into *ansStr* or an error code.

If *ansStr* is 0, `btRem` removes the *keyStr* association from tree *han* and returns *SUCCESS* if successful; an error code if not.

`int btRemRange` (*wb.Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`key2Str`**, *int* **`k2Len`**) [Function]

keyStr must be a maximum-length (256 byte) string containing a key *kLen* bytes long. *key2Str* is a string of length *k2Len*.

`btRemRange` removes [*keyStr* . . . *key2Str*) and their values. If *key2Str* <= *keyStr* no deletion will occur (even if *keyStr* is found). `btRemRange` returns SUCCESS if the operation is complete, an error status code if not.

`int btPut` (*wb.Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`valStr`**, *int* **`vLen`**) [Function]

keyStr is a string of length *kLen*. *valStr* is a string of length *vLen*. `btPut` makes the value associated with *keyStr* be *valStr* in tree *han*. `btPut` returns a status code for the operation.

`int btWrite` (*wb.Han* **`han`**, *byte* []**`keyStr`**, *int* **`kLen`**, *byte* []**`valStr`**, *int* **`vLen`**) [Function]

keyStr is a string of length *kLen*. *valStr* is a string of length *vLen*. If *han* currently contains an association for *keyStr*, then `btWrite` does not modify the tree and returns the *notpres* status code.

Otherwise, `btWrite` makes the value associated with *keyStr* be *valStr* in tree *han*. `btWrite` returns a status code for the operation.

`int btScan` (*wb.Han* **`han`**, *int* **`operation`**, *byte* []**`kstr1`**, *int* **`len1`**, *byte* []**`kstr2`**, *int* **`len2`**, *string* **`func`**, *int* []**`longTab`**, *int* []**`respkt`**, *int* **`blkLimit`**) [Function]

`btScan` scans all keys in the range [*kstr1*..*kstr2*), performing one of several functions:

operation	*func*	RESULT
COUNT-SCAN	NIL	counts all keys in range
COUNT-SCAN	given	counts all keys in range satisfying *func*
REM-SCAN	NIL	deletes all keys in range
REM-SCAN	given	deletes all keys in range satisfying *func*
MODIFY-SCAN	NIL	ARGERR
MODIFY-SCAN	given	updates values for keys in range satisfying *func*

`btScan` returns SUCCESS if scan completed; under any other result code the scan is resumable. The possible results are:

NOTPRES
: meaning the *blkLimit* was exceeded;

RETRYERR
: meaning *func* or delete got a RETRYERRR;

TERMINATED
: meaning *func* asked to terminate the scan;

<other error>
: means *func* or DELETE encountered this errror.

Each block of data is scanned/deleted/modified in a single operation that is, the block is found and locked only once, and only written after all modifications are made. Tho only exception is that MODIFY-SCANs that increase the size of values can cause block splits. Such cases are detected and converted to a PUT plus a NEXT. This has two consequences: data is written out each time a PUT occurs, and it is conceivable that *func* may be called more than once on the key value that caused the split if a RETRYERR occurs in the PUT. However, SCAN guarantees that only one modification will actually be made in this case (so that one can write INCREMENT-RANGE, for example).

func is passed pointers to (copies of) the key and value, plus one user argument:

```
    func (keystr klen vstr vlen extra_arg);
```

func is expected to return either: SUCCESS for DELETE/COUNT, NOTPRES/NOTDONE for SKIP (ie, DONT DELETE/COUNT), or any other code to terminate the scan resumably at the current point. For MODIFY-SCAN, if changing the value, the new value length is returned. Except for the case mentioned above, the caller can depend on *func* being called exactly once for each key value in the specified range, and only on those values.

If *kstr2* `<=` *kstr1*, then no scan will occur (even if *kstr1* is found). To make possible bounded-time operation `btScan` will access at most *blkLimit* blocks at a time; if you dont care, give it -1 for *blkLimit*.

The number of keys deleted/counted/modified is returned in the `skey-count` field of *respkt*; the key to resume at is returned in *kstr1* (*which therefore needs to be 256 bytes long*); and the new key length is returned in the `skey-len` field of *respkt*. If returns SUCCESS, `skey-len` is zero. NOTE that `skey-count` is cumulative, so the caller needs to initialize it to 0 when starting a new `btScan`.

WARNING: when `btScan` returns other than SUCCESS, it modifies the *kstr1* string so that the string args are correctly set up for the next call (the returned value is the new length for *kstr1*). Therefore, *kstr1 must be a maximum-length string!*

6 SCM Interface

DBSCM is a disk based, sorted associative array package (WB) integrated into the Scheme implementation SCM. These associative arrays consist of keys which are strings of length less than 256 bytes and values which are strings of length less than 64770.B. Associative arrays can be used to form a MUMPS style database which can be used to implement standard record structures without auxiliary files (see example in example.scm).

The WB implementation (compiled) adds about 66 kilobytes to the size of SCM.

6.1 SCM Status Codes

`err?` *x* [Function]

Return *x* if a valid error code (-1 . . . *MAXERR*); else #f.

`success?` *x* [Function]

Not `err?`.

`success` [constant]

Successful execution (0).

Negative integers are used for errors according to increasingly severity, as follows:

`notpres` [constant]

Successful execution, no data present or no change made

`terminated` [constant]

Failure, no damage, caller can retry operation

`retryerr` [constant]

Failure, no damage, caller can retry operation

`keyerr` [constant]

Failure, no damage, call was in error

`argerr` [constant]

Failure, no damage, call was in error

`noroom` [constant]

Failure, no damage, out of room in file.

`typerr` [constant]

Failure, file or object was not of correct type.

`ioerr` [constant]

I/O error, DB may be damaged.

`strangerr` [constant]

Internal error, DB may be damaged.

`unkerr` [constant]

Placeholder code.

`maxerr` [constant]

All error codes are between 0 and `maxerr`.

6.2 SCM Segments

The *block-size* of a segment (given in call to `make-seg`) is a parameter crucial for performance; balancing CPU time traversing blocks with file-system latency. *block-size* should be an integer multiple of the file-system block size.

In the 1990s our nominal *block-size* was 2.kiB; now it should probably be 4.kiB, 8.kiB, or 16.kiB.

`init-wb` *max-num-ents max-num-buks max-blk-size* [Scheme Procedure]

Initializes the WB system. *Max-blk-size* determines the size of the disk cache buffers. *max-num-ents* is the number of disk cache buffers to be allocated. (* *max-num-ents Max-blk-size*) should be less than the size of RAM on your computer. If not all *max-num-ents* cannot be allocated (by malloc) then WB can still run properly. *max-num-buks* is the number of hash buckets for the disk cache. It should be of approximately the same size as or larger than *max-num-ents*. The number of buffers actually allocated is returned if successful; a status code is returned otherwise.

If `init-wb` is called with the same arguments after it has already been called, `NOTPRES` (-1) is returned.

`final-wb` [Scheme Procedure]

Frees all memory used by the WB system. All segments will be closed.

`open-seg` *filename mutable?* [Scheme Procedure]

Opens the database file *filename* and returns a segment if successful, and false otherwise. The database will be read-only if the *mutable?* argument is false. It will be read-write if the *mutable?* argument is true.

`close-seg` *seg hammer* [Scheme Procedure]

Closes database segment *seg* and the file containing it. If *hammer* is #f then if there are any problems freeing buffers then the close is aborted. A status code is returned.

`make-seg` *filename block-size* [Scheme Procedure]

The integer *block-size* specifies the size of B-tree blocks. *block-size* should be an integer multiple of the file-system block size. Nominal value is 4096.

`make-seg` creates a new open empty mutable database *seg* of name *filename*. If successful, *seg* is returned; otherwise a status code is returned.

6.3 SCM B-Trees

The write-control-bits argument (*WCB*) to these functions controls the latency of updates to the file after various operations. These bits are defined as follows:

value	Name	Meaning
1	WCB-SAP	save block after PUTs
2	WCB-SAR	save block after REMOVEs
4	WCB-SAC	force block save after cached block changes (not currently implemented)
8	WCB-FAC	flush buffer entirely after cached block changes (not currently implemented)

`create-bt` *seg typ wcb* [Scheme Procedure]

Creates a new root block in seg *seg* of type *typ* and returns a bt-handle open to it if successful; otherwise #f. This would typically be used to create a temporary b-tree which should be reclaimed by check if system crashes.

`open-bt` *seg blknum wcb* [Scheme Procedure]

Returns a bt-handle open to seg number *seg*, block number *blknum* if successful; otherwise #f. If no such block exists or is not a root block, #f is returned.

For `create-db` and `open-db`, the implicit *WCB* argument is the combination of '`WCB-SAP`' and '`WCB-SAR`'.

`create-db` *seg typ name* [Scheme Procedure]

Returns a B-tree whose name has been entered in the root directory if successful; otherwise #f.

typ should be either

- `#\D` (directory) or
- `#\T` (regular tree).

B-trees with *typ* `#\D` which are pointed to by special entries in the root block (1) protect all their special entries from garbage collection by the check program. `#\T` is for regular (data) arrays.

`open-db` *seg name* [Scheme Procedure]

Returns the B-tree whose name has been entered in the root directory or #f if not found.

Dirty block buffers can also be flushed to disk by calls to `flush-ents`. `flush-ents` can be called at any time after WB is initialized, even by an asynchronous background process.

`flush-ents` *attempts k* [Scheme Procedure]

k is the number of dirty block buffers to write to disk; *attempts* is the number of times to try. Note that blocks in any segment may be written by `flush-ents`. `flush-ents` returns the number of blocks written.

Block numbers are stored in the directory as four-byte integers. In order to make WB files portable between big-endian and little-endian computers, all conversions of four-byte pointers should be done by `str2long` and `long2str!`.

`str2long` *string index* [Scheme Procedure]

Converts the 4 bytes in *string* starting at *index* into an unsigned integer and returns it.

`long2str!` *string index integer* [Scheme Procedure]

Stores *integer* into 4 bytes of *string* starting at *index*.

6.4 SCM Record Operations

`bt:get` *han key* [Scheme Procedure]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`bt:get` returns a string of the value associated with *key* in the bt which *han* is open to. `bt:get` returns #f if *key* is not associated in the bt.

`bt:next` *han key* [Scheme Procedure]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`bt:next` returns the next *key* in bt *han* or #f if none.

`bt:prev` *han key* [Scheme Procedure]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`bt:prev` returns the previous *key* in bt *han* or #f if none.

`bt:put!` *han key val* [Scheme Procedure]

han is a handle to an open, mutable bt. *key* and *val* are strings less than 255.B in length.

`bt:put!` associates *key* with *val* in the bt *han*. A status code is returned.

`bt:rem!` *han key* [Scheme Procedure]

han is a handle to an open, mutable bt. *key* is a string less than 255.B in length.

`bt:rem!` removes *key* and it's associated value from bt *han*.

The value strings `bt:get` and `bt:put!` work with are limited to 255.B in length. `db:get` and `db:put!` work with value strings up to 64770.B in length.

`db:get` *han key* [Scheme Procedure]

han is a handle to an open bt. *key* is a string less than 255.B in length.

`db:get` returns a string (up to 64770.B long) of the value associated with *key* in the bt which *han* is open to. Returns #f if *key* is not in the bt.

`db:put!` *han key val* [Scheme Procedure]

han is a handle to an open, mutable bt. *key* is a string less than 255.B in length. *val* is a string less than 64770.B in length.

`db:put!` associates *key* with *val* in the bt *han*. A status code is returned.

6.5 SCM Mutual Exclusion

These 2 calls can be used for locking and synchronizing processes.

`bt:put` *han key val* [Scheme Procedure]

Associates *key* with *val* in the bt *han* only if *key* was previously empty. Returns #t for success, #f for failure.

`bt:rem` *han key* [Scheme Procedure]

Removes *key* and it's associated value from bt *han* only if *key* is present. Returns *key*'s value for success, #f for failure (not present).

6.6 SCM Multiple Operations

`bt:rem*` *han key1 key2* [Scheme Procedure]

Removes *key*s (and their associated values) between (including) *key1* and (not including) *key2* from bt *han*. A status code is returned.

`bt:scan` *han op key1 key2 func blklimit* [Scheme Procedure]

Applies procedure *func* to a range of keys (and values) and either counts, modifies, or deletes those associations. A list of a status code, the count of records processed, and a new value for *key1* is returned.

If *op* is -1, indicated keys will be deleted; If *op* is 0, indicated keys will be counted; If *op* is 1, the value of each key in the range will be changed to be the string returned by *func*.

Func is called with 2 string arguments, the key and the value. If *op* is 1 and *func* returns #f then no change will be made. If *op* is 1 and *func* returns a string then that string will replace the value for that key. For the other (count, delete) modes, *func* should return #f or #t. If *func* is #t, the association will be counted (and possibly deleted).

Keys from *key1* (inclusive) up to *key2* (exclusive) are scanned. If *blklimit* is -1 then the entire range is scanned; otherwise only as many blocks (internal database structures) as specified by *blklimit* are scanned. The scan can be restarted by using the returned information. For instance, the following expression counts the number of keys between "3" and "4" one block at a time and returns a list of the number of keys and the number of blocks.

```
(do ((res (bt:scan current-bt 0 "3" "4" (lambda (k v) #t) 1)
          (bt:scan current-bt 0 (caddr res) "4" (lambda (k v) #t) 1))
     (blks 0 (+ 1 blks))
     (cnt 0 (+ cnt (cadr res))))
    ((not (= -1 (car res))) (list (+ cnt (cadr res)) (+ 1 blks))))
```

It is good to specify a positive *blklimit* when dealing with large scans. More details on the operation of scan can be found in `scan.scm`.

6.7 SCM Diagnostics

The value returned by the following routines is unspecified.

`check-access` [Scheme Procedure]

Checks that buffers and locks are in a consistent state and fixes them if WB routines have been interrupted.

`clear-stats` [Scheme Procedure]

Resets all the collected statistics to 0.

`stats` [Scheme Procedure]

Prints a summary of operational statistics since the last `clear-stats` or `cstats`.

`cstats` [Scheme Procedure]

Prints a summary of operational statistics since the last `clear-stats` or `cstats`, then calls `clear-stats`.

`show-buffers` [Scheme Procedure]

Prints a table of the status of all disk cache buffers.

7 SCM Relational Databases

These relational database packages are for the SCM interface.

7.1 wb-table

`(require 'wb-table)`

`wb-table` is a straightforward embedding of SLIB base-table (see Section "Base Table" in *SLIB*) in WB with SCM. It supports scheme expressions for keys and values whose text representations are less than 255 characters in length. The primitive types supported are:

boolean	`#t` or `#f`.
string	0 - 255 byte string.
symbol	0 - 255 byte symbol.
atom	internal legacy alias for symbol (or `#f`).
integer	
number	
ordinal	String representation (`<` 256.B) of numbers. Nonnegative integers sort correctly.
expression	Scheme expression (representation must be `<` 256.B).

7.2 rwb-isam

`(require 'rwb-isam)`

rwb-isam is a sophisticated base-table implementation built on WB and SCM which uses binary numerical formats for keys and non-key fields. It supports IEEE floating-point and fixed-precision integer keys with the correct numerical collation order.

In addition to the types described for wb-table, rwb-isam supports the following types for keys and values:

r64	Real represented by one IEEE 64.bit number.
r32	Real represented by one IEEE 32.bit number.
s64	Signed 64.bit integer.
s32	Signed 32.bit integer.
s16	Signed 16.bit integer.
s8	Signed 8.bit integer.
u64	Nonnegative 64.bit integer.
u32	Nonnegative 32.bit integer.
u16	Nonnegative 16.bit integer.
u8	Nonnegative 8.bit integer.

Complex numbers are supported for non-key fields:

c64	Complex represented by two IEEE 64.bit numbers.
c32	Complex represented by two IEEE 32.bit numbers.

Procedure and Macro and Variable Index

Concept Index

This is an alphabetical list of concepts introduced in this manual.