The Design of the Infinity Database Engine™ B-Tree

The Infinity Database Engine B-Tree is a fast, reliable, memory-efficient, all-Java data storage component. Its features include fast in-cache operations, high concurrency, a file-wide atomic commit with unlimited temporary workspace, guaranteed structural integrity preservation, and variable-length Items with prefix and suffix compression. With these features, it is practical to keep all of an application's data in one B-Tree file, using the file-wide commit to keep all of the application data and the B-Tree structure itself perfectly self-consistent in spite of any non-media failure. Access is fully multi-Threaded, yet there is no form of deadlock or other forced rollback. The Engine is an ideal transparent storage management component in embedded systems with a file store, real-time systems, standalone GUI applications, and modules deep within large, complex systems. Some of the main uses of the Engine, and in fact the main motivations for its creation, are covered in a companion document "*ItemSpace Data Structurese.*" The algorithm used by the Engine is documented thoroughly in U.S. Pat 5,283,894, "*Lockless Concurrent B-Tree Index Meta Access Method for Cached Nodes*" 1994 (except free space management). This document will explain the basic design and structure of the Infinity Database Engine.

Putting All of the Data Into The B-Tree

One of the main advantages of the Infinity Database Engine is that it allows all of the data for an entire application to be placed into one file, taking advantage of the file-wide commit feature to preserve data self-consistency and to allow the flexible upper-layer data structures described in the companion document. On the other hand, most B-Trees have historically been relegated to the role of indexes for the basic relations, which store the actual row data in some separate place, perhaps a separate file, perhaps a separate segment of a database. Unfortunately, most traditional B-Tree implementations have not had high reliability, so many relational systems keep the row data separate from the indexes, at least partially because the indexes must sometimes be rebuilt when they are corrupted or get out-of-synch with the row data for some reason. Getting rid of this and other limitations is the purpose of the Infinity Database Engine B-Tree. (If you are new to B-Trees, see Appendix A, which has a discussion of the basic B-Tree algorithm.)

Putting all the data into the B-Tree requires that the lengths of the Items in the B-Tree be variable and efficiently stored over a wide range of actual lengths. It is important also to have 'prefix compression,' which avoids redundantly storing identical initial parts of adjacent Items. Many of the uses of the Engine involve storing data with intentially repetitive initial prefixes in order to make data formatting easier and more extensible. The non-leaf cells of the B-Tree should also do 'suffix' compression, in which a non-selective suffix of the Items is omitted, not only to save space, but also to increase the branching factor, flatten the tree, and speed searches considerably when many Items are long. These features are provided by the Infinity Database Engine, and are described below.

The Meta Access Method

The Meta Access Method is a special in-memory structure that allows global commit, guaranteed integrity, unlimited in-file temporary workspace, high concurrency, and high in-cache search speed. The Meta Access Method is a special set of indexes called the *Meta Indexes*. There is one *Meta Index* for each level of the B-Tree. The Meta Indexes are a very fast way to find a B-Tree cell in the cache given a key and a level number. Searching a Meta Index can yield either a hit or a miss depending on whether there is a cell in the cache at the given level that covers the range of keys that include the given key. The following table illustrates a possible logical Meta Index structure, corresponding to a single B-Tree level.

Item	Cell Cache Index
`(beginning)`	`158`

`Atlanta`	`(none)`
`Boston`	`29`
`Chicago`	`(none)`
`San Francisco`	`231`
`Santa Cruz`	`112`

The left column represents the Items in the Meta Index which can be the subject of a search, and the right column is the result of a search of the Meta Index, which is an index into the cell cache. To find the B-Tree cell in the cell cache containing ‘Boulder Creek’ one searches the Meta Index for the nearest Item less than or equal to ‘Boulder Creek,’ obtaining ‘Boston’, and a cell index of 29. This is a hit, and the B-Tree cell currently in the cell cache at index 29 is then searched for ‘Boulder Creek.’ On searching for ‘Chinquapin’, the Meta Index yields ‘Chicago’ and a ‘(none)’ cell cache index, which represents a miss.

Searching the B-Tree using the Meta Access Method works as follows. When a key is to be searched for, the leaf-level Meta Index is searched, and on a miss, the next-higher level Meta Index is searched, and so on. A leaf-level hit requires searching only the leaf Meta Index and one leaf cell - a shortcut that is substantially faster than going through all the B-Tree levels from the root each time. The ‘root’ Meta Index always contains only the root cell, which is always in the cache. After a hit at a particular non-leaf level, the search recurses downwards level-by-level the way it does in a B-Tree without any Meta Indexes. As the search descends, the Meta Indexes are updated so that they include the cells that are read in.

Global Commit

The Meta Access Method design provides a ‘global commit’ feature, which makes changes to the B-Tree atomic, durable (permanent), and consistent. The `commit()` method can be invoked at any time, and when it returns, all changes made before `commit()` was invoked will be guaranteed to have been made permanent, or ‘durable.’ There is an instant in time before which none of the changes are permanent, and after which all changes are permanent: it is atomic. Other Threads are not restricted from modifying the B-Tree during the time `commit()` is executing, but the modifications are not guaranteed to take effect as a unit. Because of the atomicity of `commit()`, modifications done by the application that are consistent before `commit()` will always be made permanent in a consistent state as well.

The global commit feature of the Engine provides three of the of the four criteria in the so-called ‘ACID’ test for DBMS transaction handling, where ‘ACID’ stands for ‘atomic, durable, isolated, and consistent.’ The missing criterion is the ‘isolated’ criterion: the Infinity Database Engine does not provide any form of inter-transaction concurrency limitation, such as table locks, column locks, or row locks. These locks, if needed, can be provided at the application level or in an application-provided intermediate layer. Many applications need no locks or only a fixed, predictable set of locks and can easily provide them explicitly. For example, in embedded systems, transaction isolation and atomicity requirements can be predicted on a case-by-case basis.

The fact that the commit is global rather than per-transaction does not prevent overlapping transactions: the worst case is that, in order to preserve atomicity, the flow of transactions must be synchronized from time-to-time in such a way that no transaction overlaps the global commit point. If transactions are unpredictable and random, one way to synchronize them would be to use a timer that occasionally blocks new transactions until the executing transactions are all done. However, a wide range of applications innately serialize transactions: standalone applications for example can simply map the File/Save command directly to `commit()`, and File/Discard_Changes directly to `rollBack()`.

Integrity Preservation

The Engine also provides an internal integrity preservation feature related to the global commit. If a non-media failure occurs at any time, even during the execution of `commit()`, the internal consistency or integrity of the B-Tree will be preserved, although the changes made since the previous successful commit will be lost. Because of integrity preservation, there is no need to check the consistency of the index on every restart, or to rebuild or repair the index after a non-media catastrophe (power loss, operator interruption, out-of- memory, software failure, or other non-media failure). The lack of an initial consistency check or structure recovery allows standalone and embedded applications to come up quickly, allows frequent system restarts, and avoids the need to have the engine installed as a continuously running daemon or service. Standalone applications and many embedded systems are interrupted very frequently, and normally cannot tolerate loss of integrity at all. The integrity preservation guarantee comes from the fact that the existing B-Tree structure in the file (except free space) is not modified in any way prior to commit. At commit, a single write to the file header effectively changes the B-Tree structure all at once, as described below.

The Global Commit Algorithm

In the global commit algorithm, cells are always moved to a new location on disk when they are written back after becoming dirty. Since each cell's pointer in its parent must be updated as a result of the move, the parent must eventually be moved also. There is a ripple effect that moves all the way up the tree to the root. When the root is finally written, it will be in a new location as well. Finally, after all cells have been successfully written, a commit writes the new root's location into the file header, making the new tree permanent, and the old root obsolete. There are not normally a significant number of upper level cells to be written at any point compared to the number of leaf-level cells, so the rippling effect does not significantly limit performance. When a cell that has been modified is moved to a new location on disk, the original copy of it on disk is 'obsoleted.'

When cells are obsoleted, they are added to the free list, and newly allocated cells are removed from the free list. The free list itself uses an algorithm like the B-Tree uses in order to be committed at the instant of writing the file header: the file header contains a pointer to the head of the free list as well as the root pointer. The free list is comprised of blocks the same size as cells, so it can share the same disk area as the cells. Free list blocks are always moved when modified just as cells are (the allocation and freeing of free-list blocks themselves affects the free list itself in an interesting 'self-referencing' way.)

The Meta Access Method allows the B-Tree to be accessed and modified even while cells are being read, modified, and written back to new locations. While the cell moving is going on, there is a forest of work trees being built bottom-up in the free space in the file. The work tree cells have child pointers some of which point at cells from the previous commit cycle, and some of which point at cells that will be part of the B-Tree after the commit succeeds. Because the work trees do not have complete paths from the root, they would not be accessible from the top down, but can still be accessed by the upward recursion through Meta Index levels.

Unlimited Work Storage

The Infinity Database Engine allows an unlimited amount of updating before a `commit()` is requested. With certain extensions, and with special management of the file's free space, the same algorithm that provides the global commit feature is used to allow unlimited updating, as follows.

As cell modifications occur, the cell cache may become nearly completely dirty, and cells will need to be written out to make room for new cells to be read in. This is done from time-to-time by an 'index update,' which is identical to a commit except that a new root is written out without having its new address stored into the file header. The tentative root address is kept in temporary storage in memory instead, but becomes the actual root when a commit occurs, by being written into the file header. If a failure occurs before commit, the tentative head pointer is forgotten.

Managing free space in the file is very important during index updates. One way to keep track of free space would be to use bit maps to flag free cells. This technique was used in an early version of the Engine. There were two bitmaps (old and new), which needed to be initialized by reading the entire B-Tree on startup: a slow process. The growth of the tree was limited by the need to keep the bitmaps in memory. To overcome these limitations, and to allow variable-length cells, a free-list technique is now used. The free list management problem is almost as complex as the problem of managing the tree itself, and will not be elaborated here.

Contrast the above with the equivalent features in a relational DBMS. In most RDBMS's, there is no temporary work storage, but the relations are modified directly, and the 'before image' of each row that is modified is written to a special 'rollback segment' or to the log or to some other temporary store before the row is actually changed. If a rollback or dirty restart occurs, the contents of the rollback segment or log records are copied back onto the relation to repair the relation. A rollback segment has a finite size, but may need to backup an entire large relation: unless the rollback segment is very large, it may run out of room when a large 'UPDATE' transaction is run, such as one that modifies every row in some way. There often needs to be a pool of rollback segments in order to improve concurrency. The rollback feature must be disabled when a table is to be loaded with a large amount of data from outside. A further complication with the rollback segment system occurs in the case of a catastrophe while rolling-back. In this situation, the row data are not completely restored, and the task of rolling back must be completed the next time the database is brought up.

Concurrency

The Infinity Database Engine supports concurrent access and modification by multiple Threads in the same JVM. It does not allow multiple JVM's to open a B-Tree file at the same time, unless all of the JVM's open the file in read-only mode. The concurrency does not benefit from multiple CPU's, but does allow disk I/O to occur in background for any set of Threads while all other Threads freely access and modify any cells in the cache. Inter-Thread interference of all kinds is minimized, and throughput is maximized.

Background on Problems with B-Tree Concurrency

A little background about B-Tree systems and the concurrency problems they have faced may be helpful. The problem has been studied for a long time.

In order to allow multiple Threads to access and modify the tree concurrently, some B-Tree systems lock individual cells while the cells are being read or modified. Only when a parent cell needs to be modified in order to propagate a split or merge upwards is the parent cell locked. This is called *optimistic* locking. As a split recurses upwards, sometimes parent cells will be discovered which are locked by other Threads for splitting or which have already been split and written to disk, and whose key range no longer contains the key being inserted or deleted. The recursion fails, and the index may become unusable, or at least must have the modifications backed-out using the database log. It is possible that transactions underway may at least have to be considerably delayed while indexes are rebuilt or repaired. Any catastrophe can leave the tree in an invalid state requiring the availability of an undo log and a repair process on restart.

In order to prevent index corruption during concurrent modification, *pessimistic* locking can be used. In this case, the path from the root cell to the leaf is locked in exclusive mode in memory during every modification. This is efficient if the tree is mostly being read, but when modifications are mixed in, there can be significant blocking of one Thread by another. A 'safety' based locking scheme improves the concurrency by locking parent cells only when there is the possibility that a child may split because the child cell is nearly full. Safety based systems are more concurrent than fully pessimistic systems, but still initially require exclusive locking from the top down. Unlike optimistic locking, pessimistic locking does not risk loss of structural integrity depending on update patterns and tree structure, but as with optimistic locking, catastrophe can still leave the index unusable, because the data is written back to the same cells as it came from, and there can be an inconsistency between the parent and child cells if a failure occurs during splitting or merging.

Other concurrency systems have been devised that avoid the above problems, but the general problem is very complex. Most systems, even large, popular RDBMS's, still allow B-Tree indexes to become unusable in some situations, especially catastrophe, relying on the log to back out changes on rollback or repair the index on restart. If a log is not available to back-out or repair modifications, indexes may need to be rebuilt from the basic relation storage, either automatically or via manual commands.

Concurrency and the Meta Access Method

The Meta Access Method uses no long-term, per-cell locks at all. Instead there is only a global cache lock - like a synchronization - on the entire cache in order to prevent simultaneous access or modification of any cell in memory. Thus the Engine does not benefit from multi-CPU concurrency (in the current implementation.) However, the in-cache performance of the Engine is already so great from its basic design (65 K searches/second, 33K insertions/second per CPU GigaHerz), that multiprocessor performance increases for in-cache operations will not be important in many applications. The in-cache performance is similar to that of a TreeSet of Strings. The global cache lock is released whenever disk I/O is required of any kind; other Threads have access to the cache without waiting for the disk I/O to complete. Since there are no other locks than the short-term cache lock, the degree of disk concurrency is maximized.

The Meta Access Method allows accesses, modifications, splits, and merges of cells in the cache to occur at any B-Tree level in a level-isolated way. For example, rapid insertion of Items into a leaf-level cell may cause it to split in the cache, but the new key/pointer pair for it that must be inserted in its parent cell is not inserted immediately, instead being deferred until the next index update or commit. This level isolation allows there to be a unification of the index update into single Thread which does all upwards propagation of cell splits. The index update also performs adjacent-cell merges that become necessary due to extensive deleting. During splitting, the index update can make intelligent choices as to the split point, minimizing the resulting storage space and performing suffix compression, which minimizes the length of the parent key/pointer pair. The index update does not interfere with other Threads doing any kind of concurrent access or modification. A dirty cell is never removed from the cache until it has been processed in an index update.

Because the Meta Access Method has no per-cell locks, there can be no deadlock, and no forced rollback of Threads. There is also no need for a deadlock detector.

Storage Density

Variable-Length Stored Cells

Infinity Database Engine B-Tree cells are not stored in fixed-length blocks as they are in virtually all other B-Trees. Instead, the part of the file after the header is divided into a fixed set of variable-length units that have a more-or-less uniform distribution in size, from about 7 KB down to about 2 KB. These units are still called ‘blocks’ even though blocks are normally expected to be a fixed-length, such as 1024 bytes. The size of each block is permanent, and is determined at the moment it is allocated at the end of the file, when a free block of the necessary length cannot be found in the file. The cells and free-list blocks are both stored in the same, single variable-length block area, and free-list blocks are variable-length also.

Because the lengths of blocks in the file at any time have a roughly uniform distribution, it is usually easy to find a free block that is only slightly larger than that requested for an allocation. In fact, the excess space averages around ten bytes, depending on how much searching is done for a good match. In order to provide a reasonable number of candidate free blocks to examine, a pool of free cell addresses and lengths is maintained in memory. The pool can be scanned using a ‘best fit’ policy, or can use a ‘reasonable fit’ policy in which a ‘tolerance’ value (compiled-in) stops the scan early to save time. (An alternative scheme could try to balance fit against locality - trying to keep cells together - but this has not been tried. Note that locality is not useful if the file is fragmented by the OS anyway.) The amount of excess space and other measurements in a given file can be observed using a special B-Tree structure analyser.

The fact that blocks are variable length has another important advantage: cells can be compressed. With fixed-length blocks, compression would simply move the end of the data forwards in the block, increasing the wasted space at the end. The Engine stores cells using the Zlib format layered on UTF-8. UTF-8 compresses ASCII characters from the two-byte Java `char` down to one `byte`, while higher-valued `char`'s are encoded as two or three bytes. The Zlib compression then yields roughly a further tripling in density.

One possible complaint concerning variable-length blocks is that disk I/O may be slowed because the file and disk bock boundaries will not coincide with those of the B-Tree blocks. But the disk I/O actually speeds up. The reason is that the compressed data is so much smaller that it takes less time to transfer the data, regardless of boundaries. Disks are advertised as having enormous transfer rates that actually refer to the speed of the DMA, not the speed of the data coming from the head as it moves over the media. The actual disk speed of a typical (1999) desktop PC may be only one to two megabytes per second. With a 300 MHz Pentium II, the decompression rate of the UTF-8/Zlib combination is about 9MB/second, so it is not the bottleneck. Compression is slower, at about 3MB/second. Overall, the measured performance increase is two to three times. Further processor speed improvements will likely increase the compression and decompression speed faster than typical disk I/O speed increases. In the past, compression might not have been practical given the relative processor-to-disk performance ratio.

Compressing the stored cells also increases the effectiveness of the OS file cache. This can be very important if the 'working set' of cells (those needed often) would not otherwise fit into memory. Without the working set in memory, 'thrashing' occurs in any B-Tree or any other kind of disk-backed-up virtual memory system. Thrashing decreases performance dramatically because cells are continually brought in and then pre-empted for other cells.

Variable-Length Items

The Items in all the cells of the B-Tree whether in memory or on disk are variable-length, from 0 to 1666 Java `char`'s, and are stored in a packed form. Client programs can take advantage of the variable length of Items in many ways, as documented in "ItemSpace Data Structures." With variable-length Items, it is easy and effiicient to store a variety of kinds of information in a single B-Tree.

1666 `char`'s is about 1/3 of a cell. In the case of 1666 `char` Items, cells contain either two or three Items. When Items are small, usually about 100 Items occur per cell. The 'suffix compression' in non-leaf cells increases the number of Items from a worst-case of two to approximately 100 Items per cell, thus flattening the B-Tree and increasing search speed considerably. Suffix compression occurs both in the cell cache in memory and on disk.

Prefix Compression

Because of the packed, variable-length storage of B-Tree Items, it is possible to employ prefix compression. With prefix compression, the second Item of two adjacent Items that share a prefix can be compressed by eliminating the storage for the prefix. A third Item that has a common prefix with the second can also be compressed, and so on. Thus the compression is cumulative within each cell. Prefix-compressing a sequence of Items within a cell actually speeds searching, especially when common prefixes are large. Prefix compression is applied to cells in the memory cache as well as cells on disk.

Prefix compression complicates the choice of split point during the index update, because the initial Item of each cell is not prefix compressed, and after the split there will actually be an increase in the total amount of storage required between the two resulting cells. Choosing the split point well also involves guessing which Items will be accessed together; compressed prefix lengths are an indication of good Item groupings. Split point choice is also affected by the compressed prefix length of the parent Item once it is inserted into the parent cell.

Suffix Compression

Like prefix compression, suffix compression reduces the stored lengths of Items. It is determined only during the indexing of split leaf cells into the 'twig' level, although higher levels benefit as well. When a split point is to be determined after leaf cell splitting, the dividing line key need not be the same as one of the keys in the cell being split. The dividing line key need only be larger than all keys in the 'left' cell and smaller than or equal to all of the keys in the 'right' cell. The dividing line key can be taken as the first key in the right cell, but shortened as much as possible consistent with the preceding rule. There is an interaction between choosing a split point that is good for the leaves and one that is good for the branches, so the choice has to be intelligent. Suffix compression can have a dramatic effect on the length of keys in branch cells, and can flatten the tree considerably. At the same time, the choice of split point affects the growth of the initial item in the right cell, since its prefix can no longer be compressed. The algorithm is complex in that it is difficult to prove that a split can always be performed successfully and in a balanced way in spite of the basic variable-length nature of Items; changes in amount of compressed prefix; the variations in size due to suffix compression and many other factors.

In-Cache Search and Update Speed

The extremely high in-cache search, insert, and delete operations provided by the Engine result from using Meta Indexes and from optimizing the Engine code to take into account the cost of specific JVM operations. No Objects are constructed, synchronizations are kept to a minimum, and method invokations are kept out of tight loops

High in-cache search speed is particularly important for the recommended organization of data in the B-Tree, in which records are stored with one field per B-Tree Item (see the 'Extensible Relations' and 'Entity-Attribute-Value' data structures in "*ItemSpace Data Structures*".) Each retrieved field needs one search. Often, only one or a few fields will be needed, but even when retrieving full records, in-cache speed should not be the bottleneck: disk I/O should be, and disk I/O is the bottleneck in fact in most practical situations. For even better performance, however, Items can be structured as a simple sequence of primitive-valued components representing a traditional record, with the primary key coming first. This organization allows retrieving one record or more in one access, and is extremely fast, although possibly less extensible.

Avoiding GC and OutOfMemoryError

Much effort has been made to avoid the Engine allocating and discarding large amounts of memory during use; this reduces GC frequency, improving performance overall and reducing random GC interruptions, which can be annoying to the user or can be unacceptable for time-critical tasks. More importantly, avoiding large temporary allocations in the Engine reduces the likelyhood of OutOfMemoryErrors when a momentary peak in total memory usage occurs. OutOfMemoryErrors are catastrophes, since they can damage any part of the data structures of a running program in ways that are difficult or impossible to predict or diagnose. The Engine does require the use of temporary memory in the form of temporary Cursors, but the amount of space used is predictable and low, as described below. Other than for Cursors, there are no large peaks of temporary allocation: in fact there is almost no temporary allocation at all.

Contrast this predictable memory use to that of object-oriented or object-relational mapping systems, in which result sets are returned in Vectors, for example. The size of the Vector depends on the data in the database and the query used; OutOfMemoryErrors cannot be ruled out and the application cannot be proven to work in all cases, or even in unanticipated common cases. Whether the application crashes depends on the number of client Threads and the workload, as well. (Also, returning data in Vectors requires that all the data be returned at once, even if only little or none is actually used.)

The Engine allocates a single pool of Cursors. Cursors are needed for holding Items to be inserted, deleted, or retrieved. The Cursor pool contains a settable maximum number of Cursors, defaulting to 20; the pool grows as needed, starting empty. Each Cursor is 1666 chars long. Further Cursors may be needed temporarily if many Threads allocate them at the same time, and if so, new instances are provided which are not added to the pool: thus the pool size does not limit the number of Cursors simultaneously usable, but only the number instantly available without constructing new instances. If there are many Threads, it is possible that a momentary burst of activity will require a large number of Cursors; however, the maximum number of Cursors needed simultaneously by any given Thread is normally only about 5, and the maximum usage can be calculated by multiplying by the maximum number of Threads. Five Cursors amounts to 16KB: about the same as the cost of a Thread. The Cursor pool may be set to any desired size using

Cu.`setCursorPoolSize(int)`. To force OutOfMemoryErrors to happen as early as possible, `Cu.fillCursorPool()` may be invoked to immediately fill each slot in the pool with a Cursor instance; otherwise, the pool fills with Cursor instances only as needed.

The Engine implementation also allocates a cell cache for each open B-Tree. The cell cache grows as cells are brought into memory, up to a certain approximate point which is optionally settable by the client. The default cache size is 2.5MB. The Meta Indexes are also allocated from this cache, so no additional memory will be required as they grow dynamically. (The Meta Indexes will also shrink whenever possible, releasing cache space.) Thus the approximate maximum amount of memory used is known in advance, and can be accounted for in the memory budget. The requested maximum amount may in some situations be exceded by a few cells, but the Engine attempts to avoid going over the boundary by being conservative in estimating the sizes of various internal Objects. (It is not possible as of Java 1.3 to know how much memory an Object takes. Possibly the idea of measuring space consumption by individual Objects is not meaningful anyway.) A certain amount of random access on a file substantially larger than the cell cache will soon fill the cache, so OutOfMemoryError's will show up quickly during routine testing. The cell cache currently does not shrink.

APPENDIX A

Basic B-Tree Structure

Here is a basic description of B-Tree structure. In this discussion the term 'Item' as used above is replaced by 'key': a 'key' is used in a traditional B-Tree to retrieve a 'pointer', while an 'Item' has no associated pointer and carries information merely by existing in the Infinity Database Engine.

A B-Tree is a balanced tree representing an ordered sequence of keys and their associated short data values called pointers. Each key is also a short data value of some kind (possibly composite but still short), such as a string representing a name. The pointers might be the row Id's of rows in a flat file or other table structure inside a relational database system, in which case the B-Tree is serving as a table index. The nodes of the tree, called 'cells' have from B/2 to B branches, where B is an arbitrary number used to optimize the searching and modification, or to fit the cells into disk blocks. All cells contain an ordered sequence of key/pointer pairs, with non-leaf or 'branch' cells containing pointers to child cells, and leaf cells containing pointers that are used by the B-Tree's owner. Alternatively, the leaf cells, instead of containing key/pointer pairs, can contain any kind of key/value pairs, where the values can be for any purpose. Adjacent key/pointer pairs in branch cells define the range of keys for which the child of the left pair is responsible.

Searching the B-Tree to find the pointer associated with a given key is simple. Searching starts at the root, scanning through the cell to find the first key less than the given key. The found key's pointer is followed to a child cell and the scan is performed again, on the child cell. If a scanned cell is a leaf, the found key's pointer, if any, is returned as the result of the search. If no leaf key matches exactly, the search returns 'not found.'

The method for inserting keys into the tree while maintaining its structural rules is to split cells 'bottom up' as follows. When key/pointer pairs are inserted, the tree is searched to find the leaf cell where the key/pointer pair belongs, and that leaf cell is expanded to make room for the key at the right point in the sequence. If the leaf cell already has B key/pointer pairs in it, it must be split into two cells, one having B/2 and the other B/2+1 pairs (assuming B is even). The 'new' child cell has all the keys higher than or equal to a 'split point' key. A new key/pointer pair containing the split point key and a pointer to the new child cell is inserted into the split cell's parent cell. If the insertion into the parent causes it to expand to more than B key/pointer pairs, it is split in the same way, recursing upwards.

Deleting a key/pointer pair requires removing the pair from the leaf cell where it is found in a search. If the sum of the number of key/pointer pairs in adjacent cells which are pointed at by the same parent cell becomes less than B, the two cells are merged and a key/pair is deleted from the parent cell, recursively.

Since determining mergeability requires looking at adjacent cells, and adjacent cells may not be in memory together in a cache, sometimes merging is deferred, possibly until the next index reorganization.

The basic B-Tree algorithm is defined in terms of fixed-length Items, allowing the number of Items to be fixed in the range from B/2 to B; variable-length Items allow the actual number of Items per cell to vary widely, complicating the algorithm.