Dingsbums 6

Christoph Schwering

October 15, 2008

1 Introduction

This article describes *Dingsbums*, a database that differs in some points from traditional relational database management systems.

In the next section, we will shortly sketch the data model of Dingsbums. Section 3 will outline the software architecture of the database which means the core data structures and their relations and usage are described. In section 4 the B$^+$-tree, the central data structure, is defined.

Dingsbums is influenced by BigTable [2] and C-Store [6].

2 Data Model

The goal is to design a disk-storage, read-optimized, distributed sparse map. The data is stored in *tables*. The data model is geared to the relational model described by Codd [3], but is less expressive. In comparison to traditional relational OLTP databases, the concept of an *attribute* at least partly moves from the table schema to the data level.

Each atomary piece of data, a *data object* in a table is uniquely identified by a *row key object* and *column key object*. The concept of row keys is analoguous to the primary keys of relational databases, while the column key corresponds to the attributes of relational schemas.

However, in contrast to relational databases' attributes, column keys are not part of the schema but can appear and disappear in different table states. Hence, column keys can carry data, too.

A time value is planned in addition to the row and column key objects in order to support historical queries. For now, we notionally move the time value into the column key and ignore it.

The suspected operations are:

- insertion of objects
- deletion of objects
- selection σ of rows by a given row key object ($\leq$, $=$, $\geq$)
- selection σ of rows by a given column key objects or data objects (arbitrary relation)
- projection π of columns by a given column key object (arbitrary relation)
- join $\bowtie$ of tables on two row key objects ($=$)
- join $\bowtie$ of tables on a row key object and a data object ($=$)

Renaming ρ is not planned, neither are the set intersection $\cap$ and difference $-$. The set union $\cup$ might be provided as it is easy to implement and perform and it might prove helpful in supporting disjunctive selection and join conditions. Obviously, the data model is not relational complete.

3 Architecture

The core data structure of Dingsbums is the B$^+$-tree which stores (k_r, k_c, v) triples (for row key objects k_r, column key objects k_c and data objects v) and makes the v quickly accessible under search for (k_r, k_c). Since the order of the (k_r, k_c) keys is determined by the row keys in the first place, it is easy and quick to iterate over all (k_c, v) pairs of a certain k_r.

These characteristics of the B$^+$-tree directly support the selection by given row keys (including range queries), projection and the joins of tables.

The selection by given column key objects or data objects – i.e. those operations that involve searching of this unsortedly stored data – is still missing. Both, hash tables and bitmaps are useful in this case. Whereas hash tables are good at searching (while bitmaps definitely are not), bitmaps can efficiently compute disjunctions or conjunctions of selection queries. Both data structures have in common that for each relation one physical instance has to be created.

In many cases it might be reasonable to split a table T vertically into tablets $T_1, \ldots, T_n$. Using pattern matching on the column key object k_c, a single T_i is chosen deterministically into which the tuple (k_r, k_c, v) is inserted.

In addition to vertical splitting, horizontal partitioning is planned. The tables should be distributed on a number of servers in a shard-nothing architecture.

4 B$^+$-Tree

4.1 Formal Definition

A B$^+$-tree is an order-preserving index structure. The B-tree was first described by Bayer and McCreight [1]. Several variants of the B-tree appeared in the aftermath, the most important being the B$^+$-tree [4, 5]. Knuth also mentions a variant that is capable of storing variable-length keys [5].

Here, we give a definition of our view of the B$^+$-tree and the algorithms.

Definition 4.1 (Node)**.** A node $N = (M, E)$ of degree $d = deg(N)$ consists of some meta data M and an entry set $E = \{e_1, \ldots, e_d\}$ of d entries.

The meta data stores the degree $deg(N)$ of the node and the left neighbor, the right neighbor and the parent, denoted by

$left(N)$, $right(N)$, $parent(N)$ respectively. If such a neighbor or parent node does not exist, the value is $\perp$.

The form of an entry depends on the position of the node. An entry e_i of a *inner node* consists of

- a key value $key(e_i)$,
- a count $count(e_i)$, and
- an address $child(e_i)$.

An entry e_i of a *leaf node* consists of

- a key value $key(e_i)$, and
- a data value $value(e_i)$.

For inner nodes, we write $e = (k, c, a)$ and for inner nodes $e = (k, v)$ if the context permits it. If the context permits it, we do not distinguish between addresses and nodes, i.e. $parent(child(e_i)) = N$, for example. If N is a leaf node, $count(e)$ is defined as 1.

Definition 4.2 (Tree)**.** A B$^+$-tree is defined over a domain K of keys that is fully ordered by $\leq \subseteq K \times K$ and a domain V of values. We write $e_i \leq e_j$ in the following iff $child(e_i) \leq child(e_j)$. The tree supports the operations *insertion*, *deletion* and *search*. We can define these operations by looking at the states of the tree:

- In state S_0, the tree is empty and for all k, $search(S_0, k) = \perp$ (i.e. no value v for k can be found)
- In a state S_i, the operation $insert(S_i, k, v)$ reaches a state S_{i+1} in which $search(S_{i+1}, k) = v$, if $search(S_i, k) = \perp$. Otherwise, the insertion fails and the tree state does not change.
- In a state S_i, the operation $delete(S_i, k)$ returns v and reaches a state S_{i+1} in which $search(S_{i+1}, k) = \perp$.

In all states, the key condition which makes searching efficient must hold for all nodes $N = (M, E)$ in the tree:

$$\forall e \in E : (child(e) = (M_e, E_e)) \Rightarrow (\forall c \in E_e : c \leq e).$$

Furthermore, the counts must be consistent in all states:

$$\forall e \in E : child(e) = (N' = (M', E')) \Rightarrow count(e) = \sum_{e' \in E'} count(e').$$

Size Limits. Normal B$^+$-trees require their nodes to have a certain fixed minimum and maximum degree, because each node is intended to be stored in one hard disk block of size B (typically 4096 bytes). In contrast, our tree is not intended to support keys of a fixed size but *variable length keys*. (Note that the sizes of child addresses, count values and data values must be fixed, though.)

Hence, if $size(k)$ is the size of a certain key $k \in K$, we can uniquely determine the size of each entry e which will be denoted by $size(e)$. Then $size(E) = \sum_{e \in E} size(e)$ denotes the size of a complete entries $N = (M, E)$. Since the size of the meta data is fixed, it is easy to compute $size(N) = size(M) + size(E)$. Furthermore we will write $B_{\text{eff}} = B - size(M)$. Finally, we require for each node $N = (M, E)$ two hold the requirements depicted in figure 1.

Remark 4.3 (Key size). Note that req. 1 refers to a node element e which consists of can be either $e = (k, c, a)$ for a key k, a count value c and a child address a or $e = (k, v)$ for a key k and a value v. It follows that the maximum key size is

$$size(k) \leq \lfloor \tfrac{1}{8} B_{\text{eff}} \rfloor - \max\{size(c) + size(a), size(v)\}.$$

This upper bound is statically determined, because the sizes of child addresses, count values and data values must be fixed.

Theorem 4.4 (Node-Overflow)**.** *A node $N' = (M, E \cup \{e\})$ with*

$$size(E) \leq B_{\text{eff}} \text{ but } size(E \cup \{e\}) > B_{\text{eff}}$$

can be split into two nodes such that both fulfill req. 2.

Proof. If $E \cup \{e\} = \{e_1, \ldots, e_n\}$, set $L := \{e_1, \ldots, e_i\}$ such that

$$size(L) \geq \lfloor \tfrac{1}{2} B_{\text{eff}} \rfloor \text{ but } size(L \setminus \{e_i\}) < \lfloor \tfrac{1}{2} B_{\text{eff}} \rfloor$$

and $R := (E \cup \{e\}) \setminus L$. The sizes are bounded as follows:

$$size(L) \overset{\text{def}}{\geq} \lfloor \tfrac{4}{8} B_{\text{eff}} \rfloor$$

$$size(L) < \lfloor \tfrac{4}{8} B_{\text{eff}} \rfloor + \underbrace{size(e_i)}_{\leq \lfloor \frac{1}{8} B_{\text{eff}} \rfloor} < \lfloor \tfrac{5}{8} B_{\text{eff}} \rfloor$$

$$size(R) = \overbrace{size(E \cup \{e\})}^{> B_{\text{eff}}} - \overbrace{size(L)}^{< \lfloor \frac{5}{8} B_{\text{eff}} \rfloor} > \lceil \tfrac{3}{8} B_{\text{eff}} \rceil$$

$$size(R) = \underbrace{size(E \cup \{e\})}_{\leq B_{\text{eff}} + \lfloor \frac{1}{8} B_{\text{eff}} \rfloor} - \underbrace{size(L)}_{\leq \lfloor \frac{4}{8} B_{\text{eff}} \rfloor} \leq \lceil \tfrac{4}{8} B_{\text{eff}} \rceil + \lfloor \tfrac{1}{8} B_{\text{eff}} \rfloor \leq \lceil \tfrac{5}{8} B_{\text{eff}} \rceil$$

By this, it directly follows that $\lfloor \frac{3}{8} B_{\text{eff}} \rfloor \leq size(L), size(R) \leq B_{\text{eff}}$, i.e. L and R fulfill req. 2. □

Theorem 4.5 (Node-Underflow)**.** *When an entry e is removed from a non-root-node with entry set L such that $size(L) \geq \lfloor \frac{3}{8} B_{\text{eff}} \rfloor$ but $size(L \setminus \{e\}) < \lfloor \frac{3}{8} B_{\text{eff}} \rfloor$, either entries from a neighbor can be moved or L can be merged with a neighbor.*

Proof. Let R be the entry set of a neighbor of L. Depending on $size(R)$, either entries must be moved or nodes must be merged:

1. If $\lfloor \frac{3}{8} B_{\text{eff}} \rfloor \leq size(R) \leq \lfloor \frac{5}{8} B_{\text{eff}} \rfloor$, the nodes L and R can be merged to a node $N := (L \setminus \{e\}) \cup R$. The following equation shows that N fulfills req. 2:

$$\lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor \leq size(N) = \underbrace{size(L \setminus \{e\})}_{< \lfloor \frac{3}{8} B_{\text{eff}} \rfloor} + \underbrace{size(R)}_{\leq \lfloor \frac{5}{8} B_{\text{eff}} \rfloor} \leq B_{\text{eff}}.$$

2. If $size(R) > \lfloor \frac{5}{8} B_{\text{eff}} \rfloor$, one or more entries can be moved from R to L. We know that

$$size(L \setminus \{e\}) = \underbrace{size(L)}_{\geq \lfloor \frac{3}{8} B_{\text{eff}} \rfloor} - \underbrace{size(e)}_{\leq \lfloor \frac{1}{8} B_{\text{eff}} \rfloor} \geq \lfloor \tfrac{2}{8} B_{\text{eff}} \rfloor.$$

Entries with a cumulated size $\geq size(e)$ must be moved from R to L. If $R = \{e_1, \ldots, e_n\}$, the set $S := \{e_1, \ldots, e_i\}$ such that

$$size(S) \geq size(e) \text{ but } size(S \setminus \{e_i\}) < size(e)$$

is the minimum set that must be moved from R to L to make L valid. In the worst case,

$$size(e) \leq \lfloor \tfrac{1}{8} B_{\text{eff}} \rfloor \quad \text{for all } e \in N \tag{1}$$
$$\lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor \leq size(N) \leq B_{\text{eff}} \quad \text{for all } N \text{ except the root} \tag{2}$$
$$0 \leq size(R) \leq B_{\text{eff}} \quad \text{for the root node } R \tag{3}$$

Figure 1: Node filling requirements

$S \setminus \{e_i\}$ is verly large but not large enough, i.e. $size(S \setminus \{e_i\}) + \varepsilon = size(e)$, and the last entry e_i is unnecessarily large $size(e_i) = \lfloor \frac{1}{8} B_{\text{eff}} \rfloor$. Hence, the size of S is bounded by

$$size(S) < 2 \cdot \lfloor \tfrac{1}{8} B_{\text{eff}} \rfloor < \lfloor \tfrac{2}{8} B_{\text{eff}} \rfloor.$$

Set $L' := (L \setminus \{e\}) \cup S$ and $R := R \setminus S$. Then the sizes are bounded as follows:

$$size(L') = \overbrace{size(L \setminus \{e\})}^{< \lfloor \frac{3}{8} B_{\text{eff}} \rfloor} + \overbrace{size(S)}^{\leq \lfloor \frac{2}{8} B_{\text{eff}} \rfloor} < \lfloor \tfrac{5}{8} B_{\text{eff}} \rfloor$$
$$size(L') = size(L \setminus \{e\}) + size(S) \overset{\text{def}}{\geq} \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor$$
$$size(R') < size(R) \leq B_{\text{eff}}$$
$$size(R') = \underbrace{size(R)}_{> \lfloor \frac{5}{8} B_{\text{eff}} \rfloor} - \underbrace{size(S)}_{\leq \lfloor \frac{2}{8} B_{\text{eff}} \rfloor} \geq \lfloor \tfrac{3}{8} B_{\text{eff}} \rfloor$$

By this, it directly follows that L' and R' fulfill req. 2.

□

Algorithm 4.6 (Optimal Split of One Node)**.** *Let $E = \{e_1, \ldots, e_m\}$ be the entries that should be distribed on two nodes with entry sets E_1, E_2. The goal is to calculate*

$$\arg\max_n d(n)$$

where $d(n) = \min_{F \in \{E_1, E_2\}} size(F)$ and $E_1 = \{e_1, \ldots, e_n\}$, $E_2 = \{e_{n+1}, \ldots, e_m\}$.

1. *$n := 1$*
2. *if $d(n+1) < d(n)$, $n := n + 1$*
3. *otherwise return n*
4. *go to 2*

Proof. Obvious. □

Algorithm 4.7 (Insertion)**.** *To insert a key-value-pair (k, v), in the first step the corresponding leaf node is searched. Then the node increase handling is performed.*

Algorithm 4.8 (Deletion)**.** *To delete a key k, in the first step the corresponding leaf node is searched. If such a leaf is found, the key-value-pair is removed and then the node decrease handling is performed.*

Algorithm 4.9 (Node Increase Handling)**.** *Input is a node $N = (M, E)$ to which an element was inserted.*

1: *Maybe the node is valid.*

2: *Try to shift the* $\min E$ *to* $left(N)$.

3: *Try to shift the* $\max E$ *to* $right(N)$.

4: *Split N into two.*

This order chosen to minimize the creation of new nodes.

These steps are in detail (the synchronization and redistributions are described below):

ad 1: Write N to its (old) place and synchronize with parent.

ad 2: Redistribute $left(N)$ and N (redistribution cares about synchronization).

ad 3: Redistribute N and $right(N)$ (redistribution cares about synchronization).

ad 4: Split N to two nodes L and R. R replaces N, while L will is a new node. Create a new root node with entries L and R if N was root. Insert L into parent and synchronize R with parent otherwise (insertion of L cares about its synchronization; note that the insertion of L into $parent(R)$ might have changed *$parent(R)$). This is the only moment when the* tree grows in height*!*

Algorithm 4.10 (Node Decrease Handling)**.** *Input is a node $N = (M, E)$ from which an element was deleted.*

1: *Maybe N is root and $deg(N) \leq 1$.*

2: *Try to merge N with $left(N)$.*

3: *Try to merge N with $right(N)$.*

4: *Maybe the node is valid.*

5: *Try to shift* $\min right(N)$ *to N.*

6: *Shift* $\max left(N)$ *to N.*

The order is chosen to maximize the removal of nodes (in the first place and disk accesses in the second place).

These steps are in detail (the synchronization and redistributions are described below):

ad 1: If $deg(N) = 0$, remove N and the tree is empty. If $deg(N) = 1$, remove N and make its single child new root.

ad 2: Split N to two nodes L and R. R replaces N, while L will is a new node. Create a new root node with entries L and R if N was root. Insert L into parent and synchronize R with parent otherwise (insertion of L cares about its synchronization).

ad 3+4: Let $L = left(N)$ and $R = N$ respectively $L = N$ and $R = right(N)$. Combine them to a single N'. Note that, in general, $parent(L) \neq parent(R)$. Synchronize R with $parent(R)$ and delete L from $parent(L)$ (deletion of L cares about its synchronization; note that the synchronization of L with $parent(L)$ might have changed *$parent(R)$). This is the only moment the* tree decreases in height*!*

ad 5: Redistribute N and $right(N)$ (redistribution cares about synchronization).

ad 6: Redistribute $left(N)$ and N (redistribution cares about synchronization).

Algorithm 4.11 (Node Redistribution)**.** *Let L and R be two nodes. They are combined two a single node N (which is overflown). This is split optimally to L' and R'. Note that either children move from L to R' or from R to L'. Synchronize L and R with $parent(L)$ and $parent(R)$ (note that the synchronization of L with $parent(L)$* might have changed *$parent(R)$).*

Algorithm 4.12 (Node Synchronization)**.** *Let $N = (M_N, E_N)$ be a node and $P = (M_P, E_P) = parent(N)$ its parent. Then let $e_i \in E_P$ with $child(e_i) = N$ be the entry of N in P. The main task of the synchronization is to*

- *set $key(e_i) := \max N$, and*
- *set $count(e_i) := \sum_{e \in E_N} count(e)$*

While this is quite simple, the update of $key(e_i)$ might have the importante effect that P is no more valid*:*

- *If $size(\max N) < size(key(e_i))$ (where $key(e_i)$ denotes the* old *key value), P might violate req. 2 after the update.*
- *If $size(\max N) > size(key(e_i))$ (where $key(e_i)$ denotes the* old *key value), P might violate req. 1 after the update.*

Hence, in the first case, a node decrease handling of P must be performed; in the second case, a node increase handling, respectively.

Note that these handlings might have effect on the neighbors of P, and also on their children and parents.

4.2 Implementation

The B$^+$-tree is implemented in Ada 95. The implementation makes heavy use of generics; in total the following parameters can be provided in form of actuals for formal generic parameters:

- the key and value type and functions that define an order on the key space,
- serialization procedures for key and value objects
- a block-IO package instance.

The block-IO package could simply open files and read and write $4k$ blocks from respectively to it. More sophisticated implementations could add a caching layer or distribute the data over network. Generally, one node is stored in one block. The serialization procedures care about bringing key and value objects to the block and back again. They could also implement some compression algorithm.

Nodes that are not needed anymore are arranged in a linked list. When a new block is allocated, one of the blocks in this list is taken if possible.

There are two constant block addresses: the first block always contains the root node. This root node always exists, even if the tree is empty. The second block is the head of the list of free blocks. This block cannot be allocated; it will always remain the header in the list. These two invariants allow us to store no more tree meta data. Furthermore, they allow instant access to the tree without much initialization.

4.3 Compression

The serialization procedures can implement compression of the data of a single node. Currently, there are three kinds serialization for string-keys implemented:

- no compression, just copy the data,
- prefix compression, and
- Levenshtein delta encoding.

Remark 4.13 (Compression Condition). Let

$$c : \{E : (M, E) \text{ is a node}\} \to \{(b_1, \ldots, b_n) : b_i \in \mathbb{B}^8\}$$

be a compression algorithm that maps the entries of a node to a sequence of bytes. Let $|c(E)| = n$ if $c(E) = (b_1, \ldots, b_n)$ denote the length of the sequence of bytes. Then the following condition must hold for all sets of entries E and arbitrary entries e:

$$|c(E \setminus \{e\})| \leq |c(E)| \leq |c(E \cup \{e\})|$$

The remark says that a node may not shrink (grow) in size on disk when an entry is added (removed).

References

[1] Rudolf Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[3] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), June 1970.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 18: B-Trees, pages 434–454. MIT Press, 2 edition, 2001.

[5] Donald E. Knuth. *The Art of Computer Programming*, volume 3: Searching and Sorting, chapter Multiway Trees, pages 481–491. Addison Wesley, 3 edition, 1997.

[6] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564, 2005.