WHITEPAPER

INGRES

UNDERSTANDING AND MANAGING BTREE TABLES IN INGRES

BY PETER GALE, INGRES CORPORATION

TABLE OF CONTENTS

3	Preface
3	Overview
3	"What is a BTREE?"
5	Why Use a BTREE (and Why Not?)
6	Accessing Rows in a BTREE
9	Adding Rows to a BTREE Table
17	Space Usage and Fillfactors
22	Analyzing Fragmentation & Overflow
23	Leaf Page Overflow
24	Table Partitioning
24	Summary

About the Author

This paper was written by Peter Gale, a Senior Consultant with Premium Services at Ingres. Peter has been using the Ingres product since 1991. Chip Nickolett assisted with the development of this paper. Chip is the Director of Consulting Services for the Americas for Ingres, and he has been using the Ingres product since 1986. Both are very experienced with managing large, mission-critical environments, and maintaining the high levels of performance and availability required for those environments.

UNDERSTANDING AND MANAGING BTREE TABLES IN INGRES

Preface

The BTREE table structure in Ingres is touted as the best default choice. However, its inner working and performance implications are usually not fully understood. This white paper explains the inner working of a BTREE and provides advice on how to obtain the best performance from them. Although this information may seem superfluous to some, it is beneficial when optimizing your schema for maximum performance and efficiency in environments where every unnecessary I/O matters.

Overview

Most Database Administrators (DBAs) choose the BTREE storage structure as a matter of default. It's flexible, accommodates growth, and supports all types of queries - but at what price? Did you know that on average a BTREE takes 100% more I/Os than a HASH and 33-50% more I/Os than an ISAM (assuming full use of the key)? Did you know that BTREE tables can have overflow that never shows up in the 'HELP TABLE' output? Did you know that table scanning performance (range searches, pattern matching) gets progressively worse as more and more rows are added (albeit highly dependent on the leading-edge granularity of the key used)? Do you know how to prolong the time between modifies of a BTREE? If you answered 'No' to any of those questions then please read on. Whilst the Ingres BTREE structure works very well for many things, it should not be taken for granted since other storage structures could be more appropriate. When you do use it you should take some time to make sure you are configuring for the best growth and sustained performance. This paper will tell you how this method will be as effective for you as it has been for us.

"What is a BTREE?"

BTREE" literally stands for Balanced Multiway Tree. A BTREE table is made up of a multi-level dynamic index structure pointing to the data pages. The lowest levels of the index, known as the leaf pages, are a dense index. This means there is a separate entry on a leaf page for every row in the table. The higher level index pages point to the leaf pages or the next layer of index pages. These index pages are a sparse index where there is only one entry pointing to a page at the next level of the index. This entry shows the range of values held on the index page at the next level down. The lowest levels of this sparse index structure are known as sprig pages.

Fig. 1 shows the typical page structure of a BTREE table.

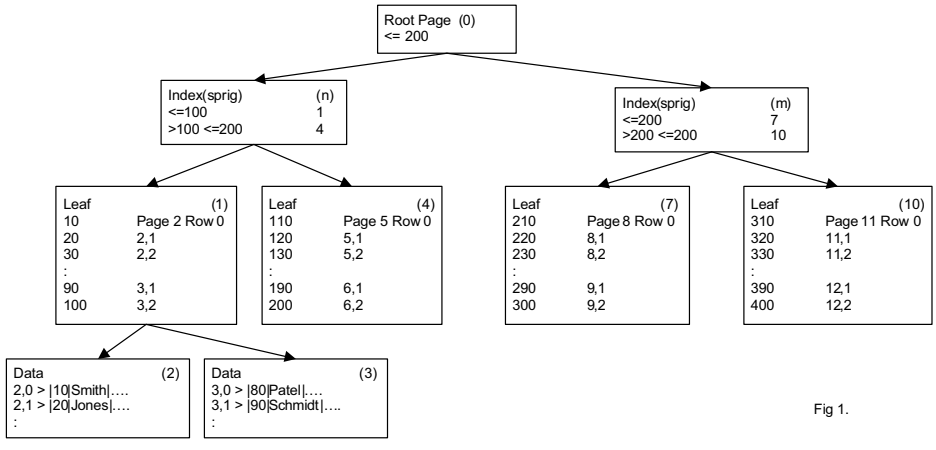


Fig 1.

(The data pages have been omitted for some of the leaf pages.)
In small to medium sized BTREE tables there won't be any index/sprig pages. These are only needed when the number of leaf pages exceeds the maximum number of entries that will fit on the Root page.

When rows are added to a BTREE table the index structure automatically expands to accommodate them. New leaf pages are added, existing pages are split in half and index levels inserted as required. As will be explained later the index structure can become unbalanced if new rows are added at the end of the existing key sequence.

The BTREE structure exists to provide a fast access mechanism for "volatile" (dynamic or changing) data. Generally it provides good performance and accommodates growth very well without degrading transaction performance. However, there is a price to pay in terms of database size, and potentially concurrency as well.

This paper will explore ways to organise BTREE tables in order to reduce the required frequency of modifies, and also minimize on disc space usage and I/Os. We will also look at how the dynamic index restructuring can be controlled in order to reduce contention problems.

Why Use a BTREE (and why not?)

The BTREE structure is often the 'default' structure that is used when creating a table. However, BTREE is the slowest of the three keyed table structures available in Ingres, on average – in an optimized environment, because it always requires more I/Os to perform a keyed read than ISAM or HASH structures. HASH tables normally require just one I/O, and ISAM generally only one but occasionally two if an index page needs to be read. BTREE often requires two because the leaf page usually needs to be read into cache as well as the data page. At first glance that may not seem terribly significant, but think again. Two is 100% more than one. If, in the extreme, all your tables were BTREE when they could be HASH, you would have double the number of I/Os taking place for all your keyed accesses.

You need to have a good reason too use BTREE as opposed to a good reason not to. These are the possible reasons why you might choose to use BTREE:

1. A table is growing rapidly in a way that would result in unmanageable overflow in ISAM or HASH.
2. The base table structure is only being used to physically cluster data, not provide an access method.
3. Partial key searches are common and the data is dynamic.

Subsequent whitepapers will look at the ISAM and HASH structures, but suffice to say in very many cases these two storage structures can and should be used in preference to BTREE. However, it is important that your decision to utilise a particular storage structure - be it BTREE, ISAM, HASH, or HEAP - is an informed decision.

You may be thinking that some of the other features of BTREE would sway your decision, such as:

- Range Searching
- Pattern Matching
- Partial (left most) key access

However ISAM also has these features, and if the data is relatively static there could be a cost savings. It is recommended to monitor query execution plans (QEPs) to ensure that additional sorts are not being added, which could negate the benefit derived by using ISAM.

If you do not require any of these features, then consider using the HASH structure. Both HASH and ISAM can be managed in a way that will allow considerable growth between each reorganisation of the table, but that is the subject of another paper.

Accessing Rows in a BTREE

Before we look at the factors that affect the performance of a BTREE table we need to understand what happens in terms of locking and I/Os when rows are accessed. Later we will see how this will change as the table grows. As for now we are looking at the locks and I/Os that occur under normal circumstances on a recently modified table with no new rows added. Ingres takes many internal locks and reads a number of pages for every access. These pages include the root page, and the free header & free map pages. Generally, these pages will not incur I/O after the first read and the locks are only held momentarily. Any variation on this will be pointed out.

Keyed Select

A basic select using the full key of the table needs to do 2 things.

- Read the leaf page that points to the data page containing the row
- Read the data page

In so doing it may well traverse a number of higher level index pages. Below is a typical lock and IO trace of a keyed select.

```
SELECT * FROM one_col_key
WHERE key_col='80000'
Executing . . .

+-------------+-------------------------------------------------------------+
|key_col      |ock_1                                                        |
+-------------+-------------------------------------------------------------+

*****************************************************************
    LOCK:   TABLE PHYS Mode: IS  Timeout:    0 Key: (btreewp,one_col_key)
    LOCK:   PAGE  PHYS Mode: IX  Timeout:    0 Key: (btreewp,one_col_key,5967)
    UNLOCK: PAGE  Key: (btreewp,one_col_key,5967)
    LOCK:   PAGE  PHYS Mode: IS  Timeout:    0 Key: (btreewp,one_col_key,5968)
    UNLOCK: PAGE  Key: (btreewp,one_col_key,5968)
    LOCK:   PAGE  PHYS Mode: S   Timeout:    0 Key: (btreewp,one_col_key,0)
    LOCK:   PAGE  PHYS Mode: S   Timeout:    0 Key: (btreewp,one_col_key,4332)
        I/O  READ  File: aaaaabaa.t00 (btreewp,one_col_key,4332) Count: 1
    UNLOCK: PAGE  Key: (btreewp,one_col_key,0)
    UNLOCK: PAGE  Key: (btreewp,one_col_key,4332)
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,4636)
        I/O  READ  File: aaaaabaa.t00 (btreewp,one_col_key,4636) Count: 1
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,4640)
        I/O  READ  File: aaaaabaa.t00 (btreewp,one_col_key,4640) Count: 1

*****************************************************************
|80000      |Row 80000                                                    |
+-------------+-------------------------------------------------------------+
(1 row)
```

Each LOCK/UNLOCK line shows the key of the page being locked in () brackets. The database is 'btreewp' and the table is 'one_col_key' and the number is the page number.

Each I/O line shows the key of the page being read in () brackets. The database is 'btreewp' and the table is 'one_col_key' and the number is the page number. From the I/O trace you can see that each read is for a single page (Count: 1).

Page 0 is the root page, the top of the index. You will also note that there are no I/Os for this page as it is already in the cache.

Pages 5967 and 5968 are the Free Header and Free Map pages. Physical (PHYS) locks are only taken on these pages to ensure that the page is being read in a consistent state. These locks are immediately released.

Page 4332 is a sprig page and permanent or static locks are not needed for a read. There was an I/O for the sprig page but this does not always happen as the page is likely to be in cache[1] .

The leaf page for row 80000 is page 4636 and the data page is 4640. These pages are locked and read.

From the I/O trace you can see that each read is for a single page (Count: 1).

Range Searching

Specifying a range of key values or querying a range using the left most part of the key works as follows.

- Find the first leaf page for the range
- Find the data pages for the leaf page
- If the end of the range has not been reached find the next leaf page and its data pages and repeat.

[1] Pages are retained in cache on the basis of their cache priority and the frequency of use. Root and index pages have higher priority than leaf pages (and data pages) and will also be accessed more frequently.

Tracing for a range spanning a single leaf page looks like this:

```
SELECT * FROM one_col_key
WHERE key_col like '7006%'
Executing . . .

key_col    ock_1
*****************************************************************
<snipped>
    LOCK:   PAGE  PHYS Mode: S   Timeout:    0 Key: (btreewp,one_col_key,3851)
        I/O  READ  File: aaaaabaa.t00 (btreewp,one_col_key,3851) Count: 1
    UNLOCK: PAGE  Key: (btreewp,one_col_key,0)
    UNLOCK: PAGE  Key: (btreewp,one_col_key,3851)
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,3980)
        I/O  READ  File: aaaaabaa.t00 (btreewp,one_col_key,3980) Count: 1
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,3982)
        I/O  READ  File: aaaaabaa.t00 (btreewp,one_col_key,3982) Count: 8
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,3983)
*****************************************************************
<snipped>
700690     Row 700690

(11 rows)
```

In this instance, the sprig page is 3851 and is locked and unlocked. Once leaf page 3980 is read a lock is taken on the first data page (3982) and multi-page I/O or group read takes place. Eight pages are read in, but only those containing relevant rows are locked (3982 and 3983). So, although we have retrieved 11 rows and read two data pages, we have still done the same number of I/Os as a keyed read.

Now let's look at a trace for a range spanning two leaf pages:

```
SELECT * FROM one_col_key
WHERE key_col like '7017%'
Executing . . .

key_col    ock_1
*****************************************************************
<snipped>
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,3985)
        I/O  READ  File: aaaaabaa.t00 (btreewp,one_col_key,3985) Count: 1
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,3989)
        I/O  READ  File: aaaaabaa.t00 (btreewp,one_col_key,3989) Count: 8
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,3990)
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,3991)
*****************************************************************
70170      Row 70170
701700     Row 701700
701710     Row 701710
<snipped>
701790     Row 701790

(11 rows)
```

The first leaf page is 3985 which shows that the range starts on page 3989. A group read takes place which reads pages 3989 to 3996 (Count: 8). The second leaf page is 3990 which has been picked up by the group read as has its first data page, 3991 (you start to see the benefits of group reading now). In this instance we have the same number of I/Os as before, but one extra lock for the additional leaf page.

The wider the range the more this process repeats. Ingres will continue to perform group reads until it reaches the end of the range, taking page locks as it goes.

If the Ingres query optimizer believes that a query will access more than 10% of the table it will automatically take a table lock at the start. This will have positive impact on the performance of this query but is bad for concurrency. Table level locking will also occur when the query exceeds the maximum number of page locks allowed per SQL statement (MAXLOCKS – See the SET sql command) or when it exceeds the maximum number of all types of locks for a transaction. Please note that this behaviour can be changed (and improved) by increasing the number of locks per transaction, increasing maxlocks, and setting "set lockmode where level=page" in ING_SET.

Full Table Scans

A full table scan will always take a table level lock and will perform group reads throughout the scan. This is true regardless of the storage structure being used.

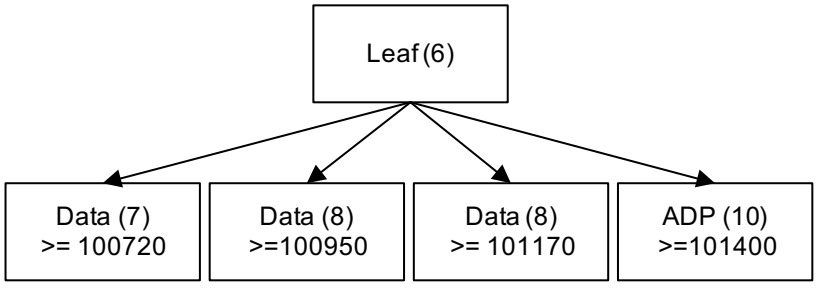
Adding rows to a BTREE table

Adding rows to a BTREE table is commonly considered its great strength. The index structure will expand to accommodate new rows and new data pages are added without generating overflow chains. Let's first look at exactly what happens when a row is inserted.

2 Overflow is a condition where a row is targeted to a specific data page by its key, but the data page has no space to accommodate it. An extra data page is created and chained off the target page. Subsequent accesses to this row have to traverse the entire overflow chain to find the page the row is on. All pages in the chain are locked during this traverse. This only happens in HASH and ISAM tables in Ingres.

As you have seen from Fig 1, each leaf page can be associated with one or more data pages. The last data page for a leaf page is known as the Associated Data Page (ADP). The ADP is the ONLY page that new rows will be added to even if the other data pages have free space. Immediately after a modify new rows will find space on their appropriate leaf page and its associated ADP. However this immediately causes a slight problem because the new rows will not be physically in order.

Lets work through and example where we have a leaf page pointing to four data pages. The range of the key values covered by the leaf page is 100720 – 101470. The key is unique and there are gaps in the sequence.



Simple BTREE
Page numbers in ()

If I now insert a row with a key value of 100721 the following locks are taken.

```
LOCK:    PAGE  STAT Mode: X   Timeout:     0 Key: (btreewp,one_col_key,6)
LOCK:    PAGE  STAT Mode: X   Timeout:     0 Key: (btreewp,one_col_key,10)
```

What has happened is that we immediately have a bit of 'Fragmentation'. The new row is on page 10 (the ADP) but rows either side of the new row in the key sequence are on page 7. For a keyed access this makes no difference as the entry on the leaf page for 100721 will point directly to page 10. But if we do a range search for all rows starting with 10072 this is what happens.

```
SELECT * FROM one_col_key
WHERE key_col like '10072%'

LOCK:    PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,6)
         I/O  READ  File: aaaaabab.t00 (btreewp,one_col_key,6) Count: 1
LOCK:    PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,7)
         I/O  READ  File: aaaaabab.t00 (btreewp,one_col_key,7) Count: 8
LOCK:    PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,10)
```

We have not incurred extra I/Os but we have locked three pages instead of the two that we would have locked for the same search prior to inserting the row, as shown on the following page.

```
LOCK:    PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,6)
         I/O  READ  File: aaaaabac.t00 (btreewp,one_col_key,6) Count: 1
LOCK:    PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,7)
         I/O  READ  File: aaaaabac.t00 (btreewp,one_col_key,7) Count: 8
```

If the leaf page had more than eight pages associated with it we would also have incurred an extra I/O to read the ADP. So, by just inserting one row we have increased the number of locks taken by 50% and possibly the number of I/Os by the same amount for this particular query. There is nothing that can be done about this. This is the way BTREE tables' work. New rows go on the ADP. The wider the row the more data pages there are per leaf page and therefore the further away the ADP is on average from any given data page.

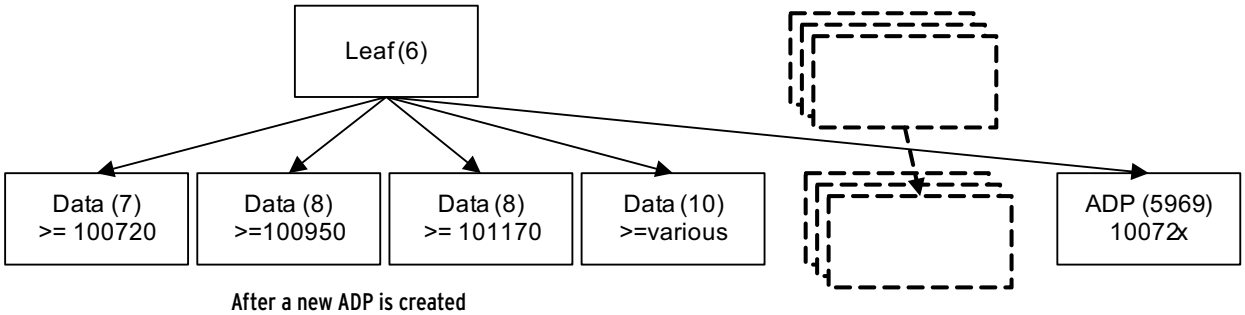
When the ADP fills up

At some point the ADP will become full. In our example this will happen when the 22nd row is added anywhere in the range covered by the leaf page. At that point a new page is created to become the ADP for leaf page 6. The rows on the old ADP (10) contain a complete mixture of rows from the range. Here is how the locking and I/O happen when the new ADP is created.

```
LOCK:   PAGE  STAT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,6)
LOCK:   PAGE  STAT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,10)
LOCK:   PAGE  PHYS Mode: IX  Timeout:    0 Key: (btreewp,one_col_key,5967)
LOCK:   PAGE  PHYS Mode: IS  Timeout:    0 Key: (btreewp,one_col_key,5968)
LOCK:   PAGE  NOWT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,5969)
UNLOCK: PAGE  Key: (btreewp,one_col_key,5967)
UNLOCK: PAGE  Key: (btreewp,one_col_key,5968)
LOCK:   PAGE  STAT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,5969)
        I/O  READ  File: aaaaabab.t00 (btreewp,one_col_key,5969) Count: 1
```

Pages 6 and 10 are locked as expected. 5967/8 are the Free Hdr/map. Ingres attempts to grab a "No Wait" lock on the first unused page that is available. What this means is that if it fails to get the lock it won't wait, it will simply go on to grab the next unused page. Once it has the page it updates the Free Hdr/Map and releases their locks. It then locks the new ADP properly and adds the row.

This whole event happens infrequently but is costly. There are 8 locking operations compared to the normal 2 for a BTREE insert. In this case the only I/O was for the new ADP. The leaf page and the old ADP were already in cache as might be expected. The relevant section of this table now looks like this.



After a new ADP is created

Having a new ADP does not affect keyed accesses, but as you can no doubt guess it does affect range searching.

```
SELECT * FROM one_col_key
WHERE key_col like '10072%'

    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,6)
        I/O  READ  File: aaaaabac.t00 (btreewp,one_col_key,6) Count: 1
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,7)
        I/O  READ  File: aaaaabac.t00 (btreewp,one_col_key,7) Count: 8
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,10)
    LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,5969)
        I/O  READ  File: aaaaabac.t00 (btreewp,one_col_key,5969) Count: 1
```

Now the benefits of group reading are much reduced. We have 50% more I/Os and we are now up to four page locks simply to retrieve eight rows in key sequence. This is real fragmentation and it only gets worse as more and more rows are added. All scanning operations will be affected with increased I/O and locking, and degraded performance. To remedy this it is necessary to perform a "MODIFY" to rebuild the index and layout the rows sequentially. Ironically, the performance of the modify is also going to be impacted by the fragmentation.

Fortunately, we can do something about this type of fragmentation. It is unlikely we would ever prevent it entirely, but we can reduce the likelihood of it occurring by building the table in such a way that the ADP will, on average, have sufficient space for the growth that will take place between modifies. After we have looked at what happens when a leaf page fills up we will look at fillfactors and how they can be used to control the free space on the ADP.

When a Leaf Page Fills Up

When a leaf page fills up Ingres performs a Leaf Split operation. The way this works has changed a number of times but from Ingres 2.6 onwards, but it is a relatively simple if costly operation. This is what happens:

- The entries on the leaf page are split into 2 groups halfway through the range of values on the leaf page.
- A new leaf page is created and the higher key value group is moved to the new page.
- A new ADP is allocated for the new leaf page.
- The sprig page for the leaf pages is updated.

The end result is we have two leaf pages that are 50% empty since the split almost always happens midway through the range of key values. Often this is part way through the range of key values on a particular data page, so the two new leaf pages will both have entries pointing to the same data page. Both leaf pages may also point to any of the data pages that are now the ADP or have been the ADP since the last modify. The end result is table scans will involve quite a bit of jumping around.

Let's look at what happened when the leaf page split:

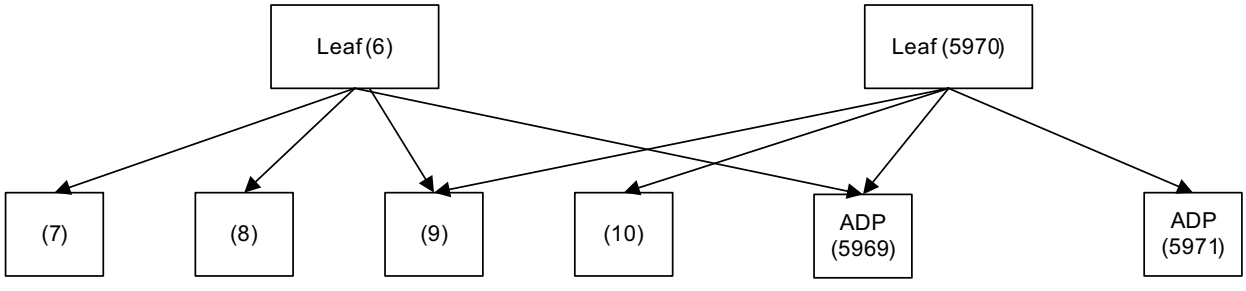
```
LOCK:   PAGE  STAT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,6)
LOCK:   PAGE  PHYS Mode: X   Timeout:    0 Key: (btreewp,one_col_key,0)
LOCK:   PAGE  PHYS Mode: X   Timeout:    0 Key: (btreewp,one_col_key,484)
UNLOCK: PAGE  Key: (btreewp,one_col_key,0)
LOCK:   PAGE  STAT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,6)
LOCK:   PAGE  PHYS Mode: IX  Timeout:    0 Key: (btreewp,one_col_key,5967)
LOCK:   PAGE  PHYS Mode: IS  Timeout:    0 Key: (btreewp,one_col_key,5968)
LOCK:   PAGE  NOWT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,5970)
UNLOCK: PAGE  Key: (btreewp,one_col_key,5967)
UNLOCK: PAGE  Key: (btreewp,one_col_key,5968)
LOCK:   PAGE  STAT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,5970)
LOCK:   PAGE  PHYS Mode: IX  Timeout:    0 Key: (btreewp,one_col_key,5967)
LOCK:   PAGE  PHYS Mode: IS  Timeout:    0 Key: (btreewp,one_col_key,5968)
LOCK:   PAGE  NOWT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,5971)
UNLOCK: PAGE  Key: (btreewp,one_col_key,5967)
UNLOCK: PAGE  Key: (btreewp,one_col_key,5968)
LOCK:   PAGE  STAT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,5971)
UNLOCK: PAGE  Key: (btreewp,one_col_key,484)
LOCK:   PAGE  STAT Mode: X   Timeout:    0 Key: (btreewp,one_col_key,5969)
```

Plenty going on here as pages are locked and unlocked, including multiple locks on the Free Hdr/Map pages as the new pages are allocated. You can see the new leaf page (5970) and its ADP (5971) are being "No Wait" locked and statically locked as they are allocated. The only I/Os in this instance were on the new pages.

Once again, the only real impact is on scanning operations. A select of all rows starting with 1009 would previously have been on the original leaf page (6). This is the result now the page has split.

```
LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,6)
    I/O  READ  File: aaaaabac.t00 (btreewp,one_col_key,6) Count: 1
LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,7)
    I/O  READ  File: aaaaabac.t00 (btreewp,one_col_key,7) Count: 8
LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,5970)
    I/O  READ  File: aaaaabac.t00 (btreewp,one_col_key,5970) Count: 2
LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,8)
LOCK:   PAGE  STAT Mode: S   Timeout:    0 Key: (btreewp,one_col_key,5971)
```

So now we are up to five page locks and three I/Os just to retrieve 12 rows that could all fit on the same page. Pictorially our table now looks like this:



Leaf page pointers after a leaf split

Again, this is a simplified case in that the original leaf page has less than eight data pages, but it is easy to see how this retrieval could become quite complex.

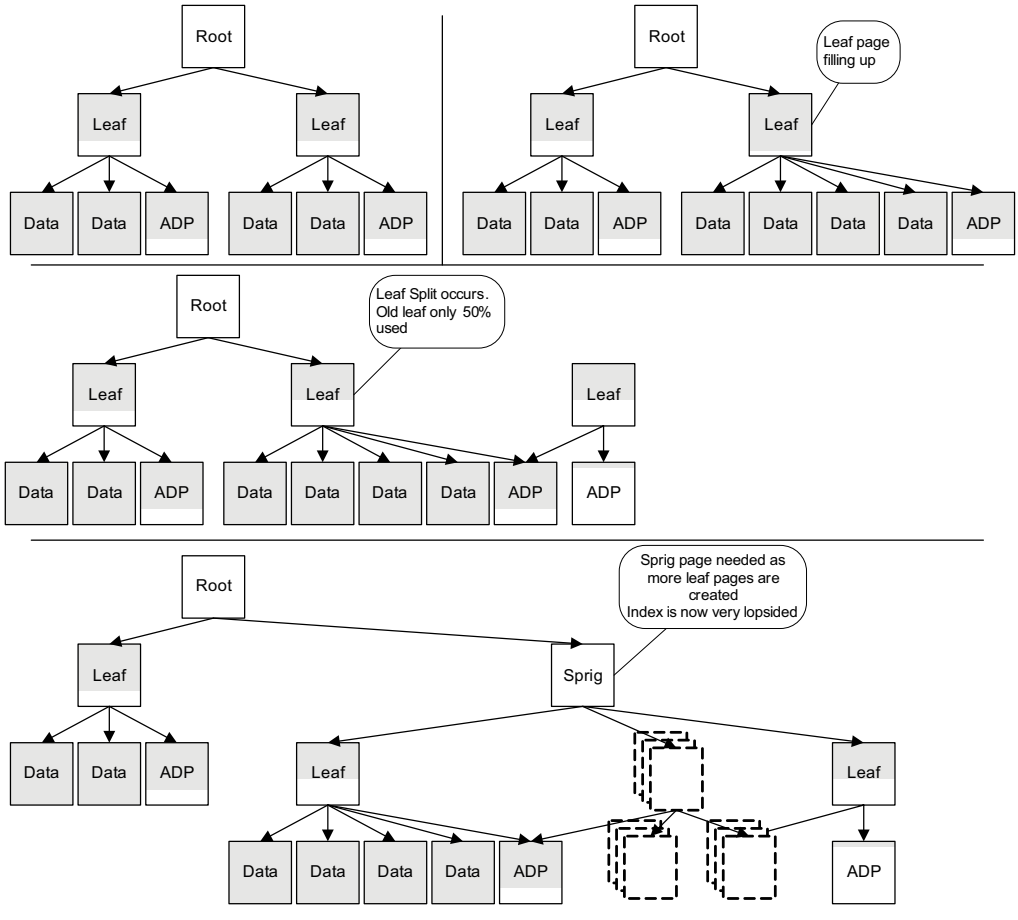
So as with fragmentation, we want to keep leaf splits to a minimum if we are going to keep the best level of performance and prolong the period between modifies. The next section deals with how space on pages is allocated and used and how fillfactors can be used to maximize the performance of a table.

Appending Rows

A common design issue is to use sequential keys on a table. It seems to be a simple and logical choice when a surrogate key is going to be used in place of complex multi-part key. Much has and could be written about making such a design decision, but that is outside the scope of this discussion. Accepting the fact that these kind of tables exist we need to know how to deal with them in our physical database.

BTREE tables are ideally suited to accepting new sequential key values. As ADPs are filled new ones are created in logical sequence alongside the previous one. The leaf pages will split and the resulting two pages will generally only share one common page. Leaf page splits are the only real issue with physical table structure in this situation.

At each split the result will be two half-empty leaf pages. Only the second leaf page and its ADP will be written to again (because rows are only being added to the end of the key sequence) and that leaf page will eventually split as well. Now we have three half-empty leaf pages, only one of which will fill up, and 3 ADPs, only one of which will have rows added. As more and more leaf splits occur, more and more space on leaf pages and ADPs becomes wasted. Also we start to develop more index and sprig pages at the end of the index structure to cope with all the new leaf pages. What we end up with is a lop side BTREE structure as shown below:



Space Usage and Fillfactors

In addition to defining the key columns and the storage structure, the MODIFY command also allows the user to specify the percentage of space that will be used on each page. These percentages are known as the Fillfactors. With BTREE tables it is possible to specify a different fillfactor for each of the 3 page types, index, leaf and data. These are specified in the MODIFY statement as follows.

```
MODIFY ….. TO B-TREE UNIQUE ON ……
WHERE indexfill=80,
            leaffill=70,
            fillfactor=80
```

The values shown above are the defaults. The fillfactor for data pages (fillfactor) is something of an anomaly because it should almost always be set to 100%. As explained earlier new rows are only written to the ADP and because of this any free space on the other data pages is simply wasted. In general, there will always be some free space left on every ADP even with a fillfactor of 100% (described later). The benefits of using a 100% fillfactor on BTREE tables all stem from the fact your tables and therefore your entire database is smaller. Any operations that scan or partially scan a table will benefit. These operations include:

- Checkpoints
- Optimisation
- Modifies
- Non or partial keyed accesses
- Unloads and Copies

Of course, the obvious question is how do we make free space on the ADP if fillfactor is 100%? There are two answers to that.

It could be argued that it is not necessary to leave free space on an ADP. It is likely that not all ADPs will get new rows anyway so the spare space is just wasted. Also the very first insert will create a new ADP and therefore the absolute maximum amount of free space will be available. However as we have seen this approach could have a detrimental effect on range search type queries because the new ADPs will now be physically out of sequence. Therefore this is a good reason to not allow this to happen. It is assumed that you are using BTREE to accommodate growth but also to allow range searching and/or pattern matching, so you want optimum performance for these operations. If all you accesses are keyed reads why are you using BTREE?

In practice the ADP will rarely be full after a modify, regardless of the fillfactor. The total number of rows on all the data pages linked to any given leaf page is determined by the number of keys that can fit on the leaf page. Taking a simple example using default fillfactors lets assume we have a table which has the following number of maximum items per 2K page.

Leaf Pages	100 Keys
Data Pages	40 Rows

(Note: These values can be obtained from the iitables system catalog, they are keys_per_leaf and tups_per_page respectively).

If the table is modified with the default fillfactors of 70% for leaf pages and 80% for data pages the space is used up as follows:

Leaf Page	100 * 70% = 70 keys
Data Page	40 * 80% = 32 rows
Number of data pages per leaf page	leaf entries / Rows per page 70/32 = 3 (rounded up)
Number of full data pages	2 (1 less than the total)
Number of rows on the ADP	leaf entries –rows on full data pages 70 – (32 * 2) = 6

(Note: The actual calculations are slightly more complex than this and are documented in the Ingres DBA Guide, but this is a fairly accurate approximation.)

Thus, the ADP is 85% empty and can accommodate another 34 rows before a new ADP is needed. This is true for all the ADPs in the table (except the last one which could contain anything from one to six rows).

Now if we do the same calculations using 100%, we get this:

Leaf Page	100 * 70% = 70 keys
Data Page	40 * 100% = 40 rows
Number of data pages per leaf page	leaf entries / Rows per page 70/40 = 2 (rounded up)
Number of full data pages	1 (1 less than the total)
Number of rows on the ADP	leaf entries –rows on full data pages 70 – (40 * 1) = 30

In this instance we only have space for 10 new rows on the ADP, but that still represents 25% of the page even though fillfactor was set at 100%. This may not be enough space to accommodate the expected growth between modifies so we will need to use the other fillfactors to adjust this. For the purposes of this paper we will assume that we are going to use 100% data page fillfactor and use the other fillfactors to create sufficient space for growth.

Adjusting Free Space on the ADP

This is actually quite a simple exercise. We can use the calculations above (or the more accurate ones from the DBA Guide) to work out the number of free rows we will have on our Leaf Pages and ADPs. Let's assume our target is, on average, to accommodate all new rows on the existing ADPs. We are assuming an even distribution of activity across the table. Not always the case but and ideal design goal.

Option 1 – Larger page size

This seems like an obvious approach, make the buckets bigger and you can fit more in. Using the above example we can see how the same table when placed on 8k pages yields the following numbers:

Max Keys per Leaf Page	423
Max Rows per Page	106
Leaf Page	423 * 70% = 296 Entries
Data Page	106 * 100% = 106 rows
Number of data pages per leaf page	leaf entries / Rows per page 296/106 = 3 (rounded up)
Number of full data pages	2 (1 less than the total)
Number of rows on the ADP	leaf entries –rows on full data pages 296 – (106 * 2) = 84

Now we have 22 free rows on the ADP plus all the benefits of using a larger page size but that is not the end of the story. Our table has 10,000 rows and on 2K Pages we had 143 ADPs with room for 10 rows each. Thus, we had space overall for 1430 new rows. On the 8K pages we only have 34 leaf pages so we actually only have space for 748 new rows, a reduction of 48%.

This is all controlled by the row and key widths so you will need to do the calculations and see which page size yields the most free space.

(Note: There is a lot more in deciding which page size to use than just the free space on a ADP. Larger page sizes can increase database contention if not managed properly. See our white paper on larger pages sizes)

Option 2 - Adjusting the Leaf Page Fillfactor (leaffill)

As can be seen from the calculations above, the number of rows on an ADP is a determined by the number of keys on a leaf page. The number of rows on the ADP is the result of subtracting the total number of rows on the other pages from the total number of keys on the leaf page. The number of rows per data page remains constant so we can adjust the number of rows on the ADP by adjusting the number of keys on the leaf page.

So it seems obvious that you simply reduce leaffill until you have enough spare rows on the ADP.

Let's say our target is to have space for 4000 new rows on the ADPs between modifies. In the above 2K example we have room for 1430 rows. We need to reduce the number of rows on the ADP to accommodate these new rows. That would mean making space for an extra 18 rows per ADP if the number of ADPs stayed the same. But it won't! As we decrease the number of keys per leaf page, and thereby the number of rows on an ADP, extra Leaf pages and ADPs will be added to accommodate the displaced rows. The easiest way to do this is to take a guess, do the math and adjust accordingly. The answer is that we set leaffill to 57%.

Let's do the math:

Leaf Page	100 * 57% = 57 keys
Data Page	40 * 100% = 40 rows
Number of data pages	leaf page leaf entries / rows per page 57/40 = 2 (rounded up)
Number of full data pages	1 (1 less than the total)
Number of rows on the ADP	leaf entries –rows on full data pages 57 – (40 * 1) = 17, leaving 23 free

We have displaced 13 keys from each leaf page in the original table

13 * 143 = 1430 / 57 keys per leaf = 33 New ADPs

So we now have 176 ADPs each with 23 rows free, a grand total of 4048 rows. We also get two more data pages for each new ADP = 99 more pages in total.

The total number of pages jumps to by 23% to 533, but this is still 7% smaller than the table would be with 80% data page fillfactor.

If a table is static in volume then these fillfactors can be used every time the table is modified. If a table is continually growing it will in due course develop enough capacity using the default 70% leaffill as more leaf pages and ADPs will exist after each modify. The option then becomes available to increase leaffill to keep the table size down to the minimum required.

The above example was based on a relatively small table of 10,000 rows. If that table continued to grow it would develop enough ADPs to handle 4000 new rows when the table size reached 28,000 rows. At 50,000 rows the leaffill can be increased to 74%, a saving of 6% on the total size of the table. One final point on this, you need to make sure that the number of free keys on the leaf page is always greater than or equal to the number of free rows on the ADP. Otherwise you will get a leaf split and negate all the benefits.

Option 3 – Adjusting the Data Page Fillfactor

"Hang on a minute", I hear you say, "I thought we were sticking to a 100%?" It is possible in extreme cases that reducing leaffill to get the desired capacity on the ADPs will actually make the table bigger than it was when the data page fillfactor was 80%. In this situation it may be desirable to put the fillfactor back to 80% or some value between 80 and 100 to gain the desired capacity.

Of course there are endless permutations of adjusting fillfactor and leaffill at the same time. Find out what suits you best by doing the calculations but remember the target is the smallest possible table size with the desired ADP and leaf page capacity. **Remember, there will come a point of diminishing returns, so beware of that.**

Analyzing Fragmentation & Overflow

Fragmentation will occur as a BTREE table grows unless all new rows are being appended to the key sequence. Leaf page overflow can occur when duplicate keys are allowed. It is important to regularly modify BTREE tables to remove fragmentation and reclaim free space created by deleting rows. However leaf page overflow cannot be removed by modifying. With the trend towards 24x7 operations, it is important to minimise the time spent performing modifies. To achieve this we need be able to determine how much fragmentation has occurred, and then adjust the fillfactors to minimize that in the future.

Fragmentation

A simple scan of the table which includes the key columns and page numbers (tid/512) will show when rows are being retrieved from pages that are out of sequence.

```
SELECT key_col,tid/512
FROM one_col_key
Executing . . .
```

```
+-------------+-------------+
|key_col      |col2         |
+-------------+-------------+
|            0|            2|
|           10|            2|
|           11|            8|
|           12|            8|
|           13|            8|
|           14|            8|
|           15|            8|
|           16|            8|
|           17|            8|
|           18|            8|
|           19|            8|
|           20|            2|
|           21|            8|
|           22|            8|
|           30|            2|
.
:
|          850|            5|
|          860|            5|
|          870|            5|
|          871|           49|
|          880|            6|
|          881|           49|
|          890|            6|
```

The rows on page eight have all been added since the last modify and are on the ADP for leaf page one. These rows are already slightly physically displaced from their adjacent rows. When the number of data pages per leaf page is high the ADP can be a long way from its logically adjacent data pages. The two rows on page 49 (at the end of the table) are on a new ADP that was created once page eight filled up. This is a test bed example, but the same principle applies on any BTREE. On large table the new ADP could be many thousands of pages away.

Leaf Page Overflow

Duplicate keys can cause leaf page overflow and also alter what happens when leaf pages split. In a unique table a split is done half way through the range of key values on the leaf page. With duplicate keys the split will not normally result in a 50/50 split of the rows between the two leaf pages because all the duplicates have to be kept on the leaf same page.

Thus, as new duplicate rows are added, splits occur more and more often until all the keys on the page are the same. At that point duplicate keys will cause leaf page overflow. This increases I/O. This is an example of what happens:

Start point				
Maximum 10 keys per leaf page	2 Rows with key 20 added. Page splits	6 more rows with key 20 added	1 more row with key 20 added	1 more row with key 20 added. Overflow occurs
Leaf page 1	Leaf Page 1	Leaf page 1	Leaf page 1	Leaf page 1
10	10	10	20	20
20	20	20	20	20
30	20	20	20	20
40	20	20	20	20
50	30	20	20	20
60		20	20	20
70	Leaf Page 2	20	20	20
80	40	20	20	20
	50	20	20	20
	60	20	20	20
	70			
	80	Leaf Page 2	Leaf Page 2	Leaf Page 2
		40	40	40
		50	50	50
		60	60	60
		70	70	70
		80	80	80
		Leaf Page 3	Leaf Page 3	Leaf Page 3
		30	30	30
			Leaf Page 4	Leaf Page 4
			10	10
				Oflow Leaf Page
				20

So progressively fewer and fewer key values are removed from the original leaf page, and the consecutive key values are now jumbled up across five pages that are probably physically displaced from each other. Modifying the table will eliminate the fragmentation but not the overflow. Using larger page sizes will allow greater numbers of duplicate keys on the same leaf page, but the real answer is to examine the design and eliminate the duplication.

Leaf page overflow can be identified using SQL. To do this you will need to work out the maximum number of keys that can be held on a leaf page using the formulas in the DBA Guide. Feed this value into the SQL below. Any rows that are returned indicate duplicate key values that are in leaf overflow.

```
SELECT key_col_1, key_col_2……, count(*)
FROM table
GROUP BY key_col_1, key_col_2……,
HAVING count(*) > max_keys_per_leaf_p
```

Table Partitioning

The ability to specify value based distribution of rows across disc partitions has added a whole new complex dimension to table performance. An in-depth study of this will be the subject of another white paper, but at this stage it is clear that partitioning radically alters the I/O characteristics of all storage structures. In particular it is likely with AUTOMATIC or HASH partitioning that performance of range searching and pattern matching is going to be degraded. Consecutive values of the partitioning columns will NOT be on the same or adjacent pages so the benefit of group reads is likely to be lost.

Summary

Optimum performance from a BTREE is obtained by keeping the fragmentation as low as possible. Sequential keys are bad news for the storage structure, and regular modifies will be needed (or, use HASH if possible). For random insertions, look to increase the free space on the ADPs to accommodate the expected number of new rows between modifies. Whenever possible, use 100% fillfactor and adjust the leaffill to increase the free space on the ADPs. Remember, that ISAM tables can handle new rows, and, up to a point, they do it better than BTREE tables.

NOTES

NOTES

About Ingres Corporation

Ingres Corporation is a leading provider of open source database management software. Built on over 25 years of technology investment, Ingres is a leader in software and service innovation, providing the enterprise with proven reliability combined with the value and flexibility of open source. The company's partnerships with leading open source providers further enhance the Ingres value proposition. Ingres has major development, sales and support centers throughout the world, supporting thousands of customers in the United States and internationally.

INGRES CORPORATION : 500 ARGUELLO STREET : SUITE 200 : REDWOOD CITY, CALIFORNIA 94063
PHONE +1.650.587.5500 : FAX +1.650.587.5550 : www.ingres.com : For more information, contact info@ingres.com

INGRES