

Indexing

Presented by: **Atanas Atanasov**

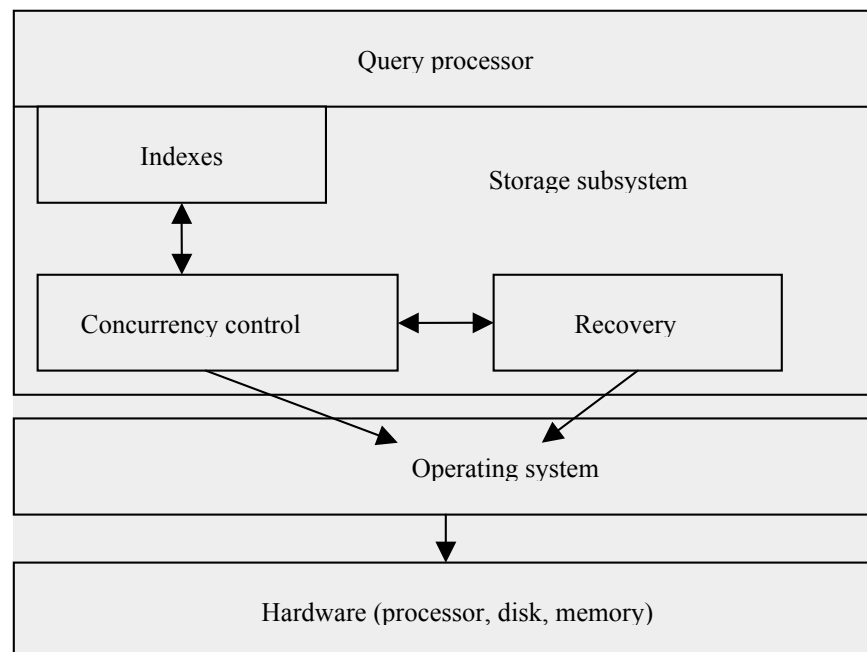
INTRODUCTION

An index for a table is data organization that enables certain queries to access one or more records of that table fast. Proper using of indexes is therefore essential to high performance. Improper selection of indexes can lead to the following mishaps.

- Indexes that are maintained but never used
- Fields that are scanned in order to return a single record
- Multitable joins that run on for hours because the wrong indexes are present

When a query is initiated without an index, 4D starts at the beginning of the table being queried and examines each record sequentially, one by one. When querying with an index, the index structure is typically traversed first – potentially skipping many records – then 4D returns the matching records.

The following figure shows the place of indexes in the architecture of a typical database system.



Indexes are provided from 4D. 4D organizes the access to data in memory and, for clustering indexes, also organizes the layout of data on the disk. Indexes are tightly integrated with the concurrency control mechanisms. They are heavily used by the query processor during the query optimization.

Index is a data structure built on certain architecture. The most common index architectures are “Clustered” and “Non-Clustered”.

A non-clustered index contains references or points to a blocks that contains the row data for which the index has been constructed. Depending of the size of the rows, it may hold several other rows. The logical order of the index

does not match the physical stored order of the row on disk. The leaf nodes contain the row of the index. The clustered index architecture is based on “Clustering” or re-orders the data block in the same order as the index. Therefore a table can have only one clustered index.

BENEFITS AND LIMITATIONS OF USING INDEXES

The main benefit of using indexes is speeding up the process of retrieving data from a database.

For example:

Let say we have a table named “People” with two fields for “First” and “Last” name and we are searching for “John Smith”.

Query ([People]; [People]Last=“Smith”;*)

Query ([People]; &; [People]First=“John”)

In this query the database will start looking into the “Last” field in every row on the table (This is called “Full table scan”). Then it will do the same within the new selection and will search sequentially every record inside the selection for first name equals “John”. If there is a B-tree index setup on “Last” field, the database engine simply follows the B-tree structure to find the requested value.

Example:

If we add one more field named “Email” in our “People” table, and setup a B-tree index on this field. Consider the following expression.

Query ([People]; [People]Email=“@yahoo.com”)

This query will return all people which email address ends up with “yahoo.com”, but even the fact that the “Email” field has B-tree index setup; the database will perform sequential search or full table scan. This is because the index keys are built with assumption that words go from left to right. With a wildcard at the beginning of the search, the database will be unable to traverse the B-tree index.

Other limitation of the indexes is when too many indexes are added to the database or new records are added to the database. The size of the index file grows. This can result in indexes taking up quite a bit of space. Using too many indexes can actually slow your database; each time a record is updated or removed, the index also has to be updated. This is one of the trade-offs between performance and record maintenance.

Indexes key types

A key of an index is a set or sequence of attributes. Index search use values on those attributes to access records. Many records in the Many table can have the same key values. There are three types of index keys:

- A sequential key; this when the last record inserted into the table has the highest value.
- A nonsequential key; the value of this type of key is not related to the table order. Let's say SSN of the last person added to a table with indexed field SSN will not be the highest SSN.
- An unique key; the value of this key is unique for the indexed field. Unique keys are used in One table in One to Many and One to One relations.

When is not acceptable to use indexes

If the indexes are not use in proper way it may lead to poor database performance, problems with the structure and the data file, quickly run out from disk space and crashing. In order to avoid these pitfalls, the developer has to do some analyses of the database and the reason to apply indexes on the table's fields. One good effort will be to optimize the query without using an index or use the right index for different type of queries. There is some guidance when to avoid using indexes;

- Problems with the structure file: When you have structure problems with your database, try to fix the problems prior adding index to a field otherwise the problems will persist and speed up your queries won't solve them. Most of the time adding indexes to database with structure problems will make the problems worst.
- Fragmented data file: Prior adding index to a table compact your data file and indexes. In this way the index will be checked and compacted. Applying index on fragmented data file will results in creation of new index pages, half full index pages and poor performance. This also will affect the update and maintenance of the data file.
- Not enough disk and memory space: Adding indexes to a database increase the physical size of the database. Also, every call to the indexed field will load the index pages into the memory. As a result this might slow overall performance of the system.
- Frequent changes of the database structure during the development process: During the development process 4D developer has to focus on building the database. Later on, when testing the structure, the developer should try to find which queries are slow and try to increase the speed by using indexes.
- Redundant fields: If you apply indexes to redundant field (two fields on the same table of different tables with the same data), this will affect the performance of the machine. One possible solution is to normalize the database (Normal form 2).
- Avoid applying indexes on small tables.

Index optimization tips

- Limit the number of indexes per table. Indexes increase the time it takes to perform Insert, Update and Delete, so try to limit the number of indexes. If you have read-only table, the number of indexes can increase.
- Keep the indexes as narrow as possible. This will reduce the size of the index files by compressing the index files. At this point the index architecture will be rebuilt. In this way you reduce the number of reads required to read the index and frees you stack memory

- Try creating indexes on integer field. Numeric field are search faster then text fields. For example: instead perform query on city field, do the same query on Zip Code. The search will be faster and result will be more accurate (Some cities have more than one zip code for different areas).
- The order of the index fields in the composite index is very important. Fields with high selectivity are query after the fields with low selectivity. Cluster index first
- Cluster index is more desirable if you need to select by the range of values or you need to sort results set with **ORDER BY** or working with sets. On the other hand, regular btree index is better for field where you need to perform some math operations like **SUM**, **MAX**, . Etc. You need to add or compare any single value in that field.

WHAT IS NEW IN 4D V11 SQL

This section covers some of the new implementation in Version 11 related to indexes. These changes were made because new indexes have been introduced in v 11, and the way how 4D manages those new indexes. The number of index keys per table is increased from 16 Mln. to 128 Bln.

With 4D v11 SQL, the index structure and index fields are separated from the database file in two different files:

[database_name].4DINDY is the structure file

[database_name].4DINDEX is the data index

All database indexes are now stored in these two external files which are automatically placed next to the structure. They must not be renamed or moved; otherwise, 4D will have to create them again. One of the main benefits is if the index becomes corrupted, it is possible to physically remove the files. Next time when 4D starts, it will create both files. Once these files are created the database engine takes care of updating the index structure. Index keys are no longer loaded with the record they belong to unless referenced by 4D. This new approach 4D v11 SQL uses specific temporary memory that is devoted for indexing and sorting operation. This temporary memory is different than the memory used from the database. Indexes are stored in cache memory. 4D sets the “cache priority” for different objects in the memory likes, records, indexes, index address tables and so on. The weight allocated for an object is increased in proportion to the times the object is accessed by the code thus the priority can change. In v11, and index has higher priority than a record. Every time when this record is accessed add weight to the record’s priority and this may lead to having higher priority than the little accessed index. At this point the index could be removed from the cache before the record.

Some commands were depreciated in Version 11 and some were changed to reflect the new index implementation. Commands like: **SEARCH BY INDEX** and **SORT BY INDEX** are depreciated in v11. There are commands like **QUERY BY FORMULA**, **QUERY SELECTION BY FORMULA** and **APPLY TO SELECTION** which were changed to support the new technology. With v 11, 4D developer can manage indexes dynamically by using the commands **CREATE INDEX**, **DELETE INDEX** and **SET INDEX**. With help of the MCI API now, it is very easy to check the integrity of index files.

All of this will increase the overall performance on your database; also the database will be immunized from index related errors.

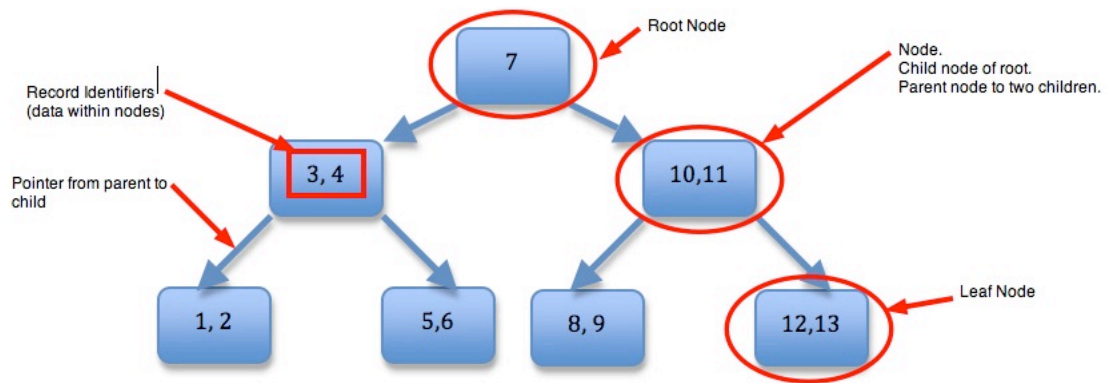
AVAILABLE INDEXES IN 4D V11 SQL

In previous version of 4D only the B-tree was available as an option to speed up the performance of a database. In v11 were introduced three new types of indexes. Cluster pointing to an array or a bitmap, Composite and Keyword index:

B-tree index

The first indexing type is B-tree index. A B-tree is a balanced whose leafs contains sequence of key-pointer pairs. The keys are stored by values. The architecture of this index is as follows. Record Identifiers, such as an ID field, populate and ordered set of elements, called nodes. These nodes are connected by pointers to each other in such a way that there is a single node that has no pointers to it - called the *root* node – and each

subsequent node underneath the root either contains key values associated with the field that is indexed and pointers to more nodes, or just the key values with no pointers. The image below illustrates this concept:



Definitions:

Node/Page: The elements that the tree is built with. A node contains keys and pointer to a child node.

Subtree: A subset of nodes within the main tree that contains a node and all the nodes underneath it. In the picture above, the node that contains keys 3 and 4 can be seen as the root of a subtree that has 2 child nodes.

Root: The top-most node. It is the only node without a parent.

Child node: Every node has 0 or more child nodes, which are below it in the tree.

Leaf node: Nodes at the bottom most level of the tree. They don't have any children.

Parent node: A node that has children. A child node has at most one parent.

Height: This is the length of the longest downward path from a node to a leaf.

Height of the tree: The height of the root. Every tree has height h . In the example, $h = 3$.

Depth of the node/Root path: length of the path from the node to its root.

Inner node/Subpage: Any node from the tree that has children and is not the root node.

Key: An element inside a node that contains the record's unique

When you create an index, the database engine allocates a single page or node. This page represents the root node and remains empty until you insert data in the table. When the root node becomes full, the optimizer creates leaf nodes (Leaf nodes are with one more than elements in the parent node) and moves the entries from the root node to the newly created leaf nodes and puts pointers to those leaf nodes in the root node. The indexes are updated immediately; create a record with an indexed field and the index is updated right away.

When all leaf nodes are at the same depth, the tree is said to be balanced. Every time a record is added or removed, the tree re-balances itself. This is a property of the B-Tree structure that helps maintain efficiency. The height will increase as elements are added to the tree, as seen in the coming examples.

The root node and the branch nodes contain key values and pointers to other branches or leaf nodes. Only leaf nodes contain information about record numbers in the table. So if you were to query a record whose key was stored in an inner node, that node would point to a leaf that contains information about the record in question.

Index Selectivity:

Index selectivity is a term used to describe the ratio of the number of unique values in a field in the table versus the total number of records. The B-tree index is very efficient when the index selectivity is equal to one. So for example, if you have a field, [Cars]ModelID, that has 4 distinct values (BMW, Porsche, Mercedes and VW), and the table contains 8 records total:

$$\text{Index Selectivity} = \frac{\text{Distinct_Records}}{\text{Total_Records}} = \frac{4}{8} = 0.5$$

You should create B-Tree indexes on tables where the majority of queries are most likely to return less than 15% of all rows. Beyond that point, a sequential search is more effective. Once the search engine matches the search value with the key in a node, it can use the pointer to efficiently fetch the corresponding rows from the table.

Properties of the B-tree

Let's define a B-Tree with a root, $root[T]$.

Let $n[x]$ be the number of keys stored in node 'x'. $n[x]$ is always stored in non-decreasing order:

$$n_1 \leq n_2 \leq \dots \leq n_{n[x]}$$

There are upper and lower bounds of the number of keys a node can contain. These bounds depend on the **minimum degree** of the B-tree, t .

t determines how many different keys are allowed in one node as follows:

Example:

$$t = 2$$

Minimum number of keys per node: 1

Maximum number of children for this node: 2

Upper limits:

Every node can contain at most $2t-1$ keys. We say the node with $2t-1$ keys is full. This node will have $2t$ children.

Example:

$$t = 2$$

maximum number of keys per node: 3

Maximum number of children for this node: 4

le:

$$t = 2$$

Minimum number of keys per node: 1

Maximum number of children for this node: 2

Upper limits:

Every node can contain at most $2t-1$ keys. We say the node with $2t-1$ keys is full. This node will have $2t$ children.

Example:

$$t = 2$$

maximum number of keys per node: 3

Maximum number of children for this node: 4

Every internal node contains $n/x + 1$ pointers $p_1, p_2, \dots, p_{n/x+1}$ to its children. Leaf nodes have no children so their pointers (p_i) fields are undefined.

Every leaf has the same depth, which is the tree height h . In other words, the maximum number of access operations required to reach any leaf is equal to the number of access operations for all other leaves.

Here is an example of how a B-Tree structure is built, assuming $t = 2$:

$t = 2$

minimum number of keys per node: 1

maximum number of children with 1 key: 2.

maximum number of keys per node: 3

maximum number of children with 3 keys: 4

Every internal node contains $n/x + 1$ pointers $p_1, p_2, \dots, p_{n/x+1}$ to its children. Leaf nodes have no children so their pointers (p_i) fields are undefined.

Every leaf has the same depth, which is the tree height h . In other words, the maximum number of access operations required to reach any leaf is equal to the number of access operations for all other leaves.

Here is an example of how a B-Tree structure is built, assuming $t = 2$:

$t = 2$

minimum number of keys per node: 1

maximum number of children with 1 key: 2.

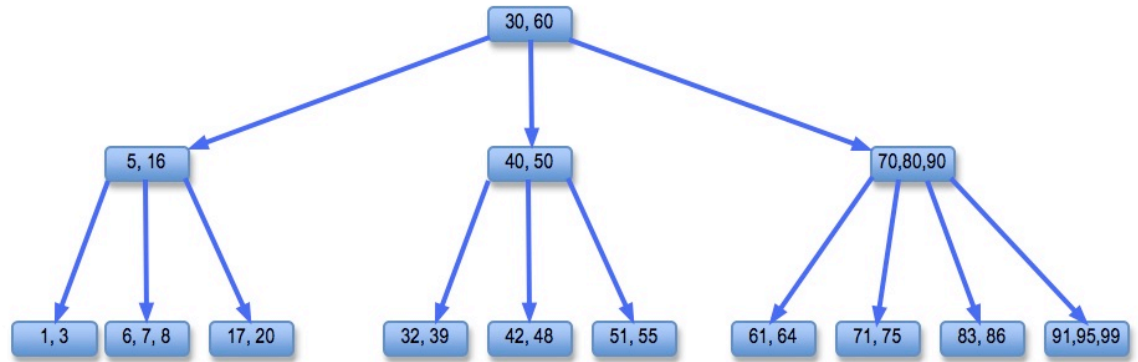
maximum number of keys per node: 3

maximum number of children with 3 keys: 4

Let say we have the following thirty-one keys:

1,3,5,6,7,8,16, 17, 20, 30, 32, 39, 40, 42, 48, 50, 51, 55, 60, 61, 64, 70, 71, 75, 80, 83, 86, 90, 91, 95 and 99

This is what the tree will look like after all the keys have been inserted:

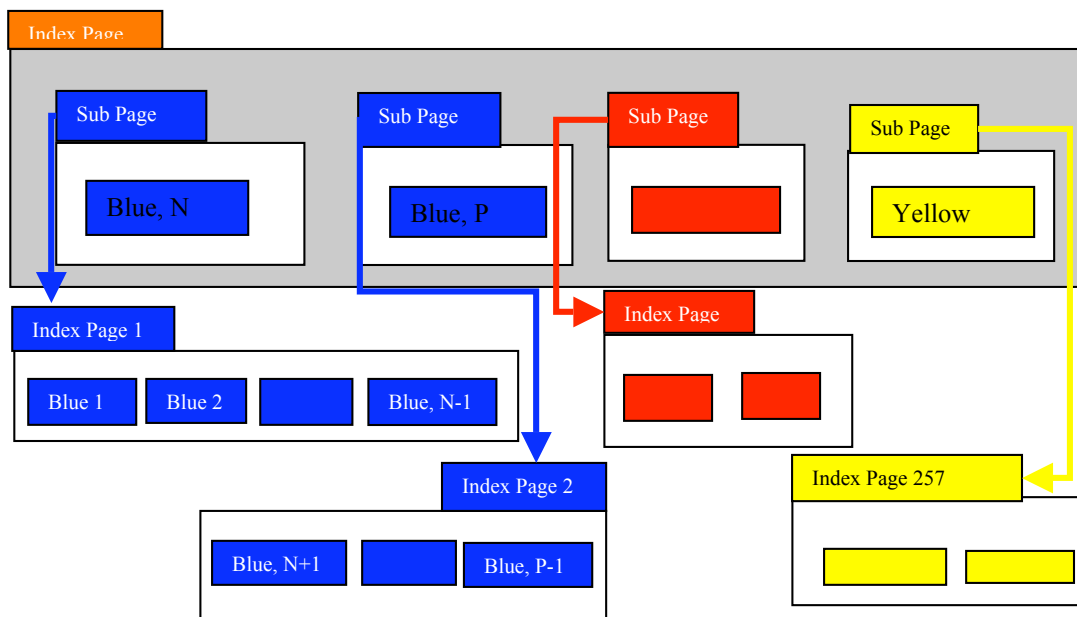


B-Trees have substantial advantages over other methods when access time to the node exceeds access time within nodes. This usually occurs when most nodes are in the secondary storage. For example, if you have nodes with a fairly small number of keys loaded into memory, the search inside these nodes will be very fast. However, each time these nodes reference a record stored on disk, the disk will be accessed. It is more desirable to send one request to the disk for a lot of data instead of many requests to the disk for a little data at a time. This is accomplished by having many child nodes per node; in other words increase the size of t . In this situation, maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often, and the efficiency increases. For most indexing types in 4D, the value of t is equal to 256. For Alpha and composite indexes, $t = 128$. B-tree index support many different queries types. If range queries (e.g. find something between X and Y) or extreme queries (e.g. Min or Max) occurs frequently, then this type of index is useful.

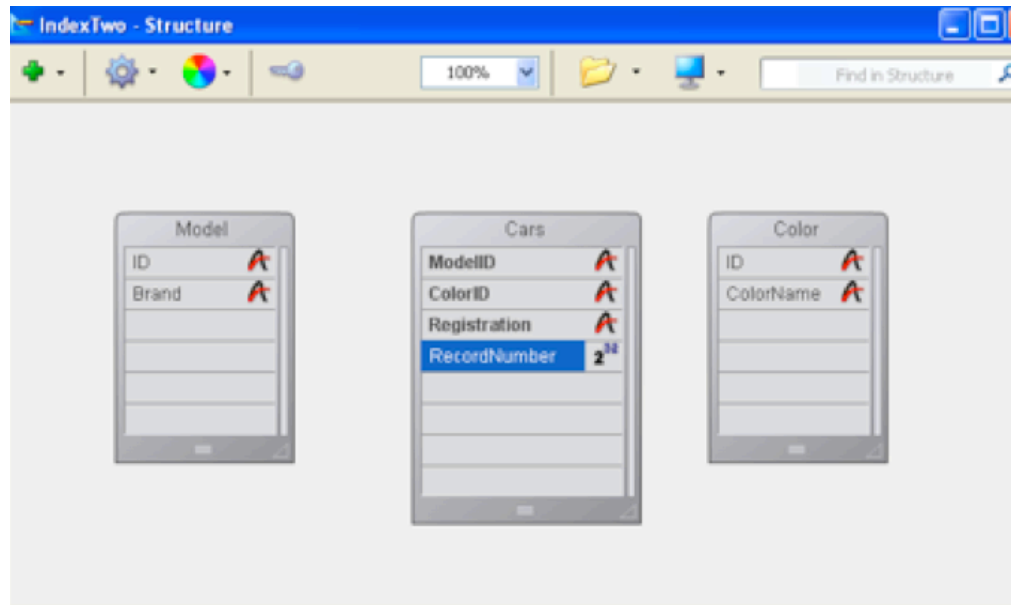
In 4D, a B-tree index has 256 keys per page. In other words, $t=256$. In our example where we have 5 million records, that leaves us with 19532 pages.

$$\frac{5,000,000}{256} = 19532$$

Will this number of pages slow down our query?



Now that we have seen how a B-Tree structure is built, let's see how it behaves in 4D. Here is an image of the database structure that will be used in these examples.



Query Cars[ModelID] Without an Index

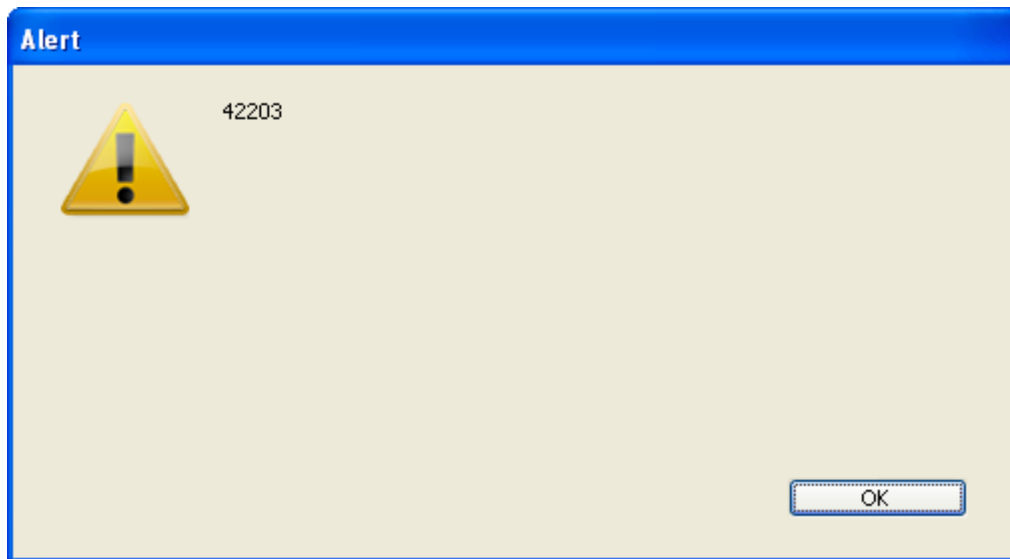
First we will execute a query on one field, [Cars]ModelID. We will not use an index on it for our first run. The Cars table has 5,000,000 records in it with four distinct values: "Mercedes", "BMW", "VW" and "Porsche". Let's look for just BMWs using the following code:

Our results return 833,333 records:

```
`The alert will return how many milliseconds it takes to perform the
`Query
t:=Milliseconds
Query ([Cars]; [Cars]ModelID="BMW")
Alert (String (Millieconds-t))
```

IndexTwo - Cars: 833333 of 5000000			
RecordNumber	ModelID :	ColorID :	Registration :
3	BMW	Red	CA
11	BMW	Black	CA
15	BMW	Yellow	CA
23	BMW	Black	CA
27	BMW	Black	CA
35	BMW	Black	CA
39	BMW	Blue	CA
47	BMW	Red	CA
51	BMW	Red	CA
59	BMW	Red	CA
63	BMW	Black	CA
71	BMW	Black	CA
75	BMW	Red	CA
83	BMW	Red	CA
87	BMW	Black	CA
95	BMW	Red	CA

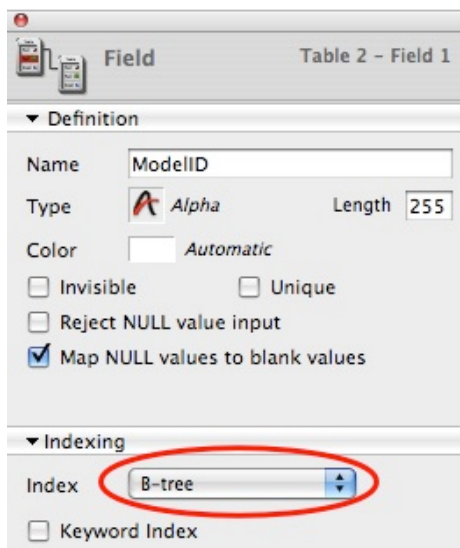
And our alert reads that it takes over 42 seconds to find them!



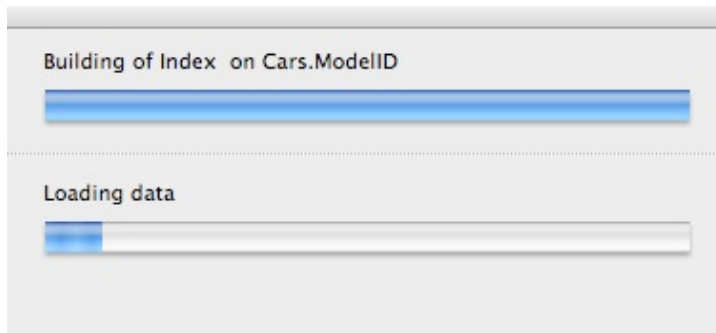
This long search time is not completely unexpected; 4D searched each record –one by one- in the Cars table for records that matched our query condition.

Query Cars[ModelID] With a B-Tree Index

Now we are going to repeat the same query but this time uses a B-tree index. In order to enable the index, go to the field in the Structure view and select the appropriate indexing type in the Index section:



The field name goes bold in the structure to indicate that there is an index on the field. Also note that 4D builds the index file, which may take some time if the table is large. For this particular table, building the index file took a few seconds. The following progress bar is displayed when the index is being built.



Before executing the query, let's step back to make an educated guess on the results. The success of this B-tree index depends on the index selectivity. Remember, *selectivity* is defined as the number of unique records divided by the total number of records. In the case of the ModelID field, that is:

$$4 / 5 \text{ Million} = 8\text{E-}6 \text{ (or } 0.00000008 \text{)}$$

The B-Tree index performs optimally when the selectivity is close to 1, which we are quite far from. How do you think the query performance will behave now? Once the index is created, we execute the same query from the previous section with the following results:



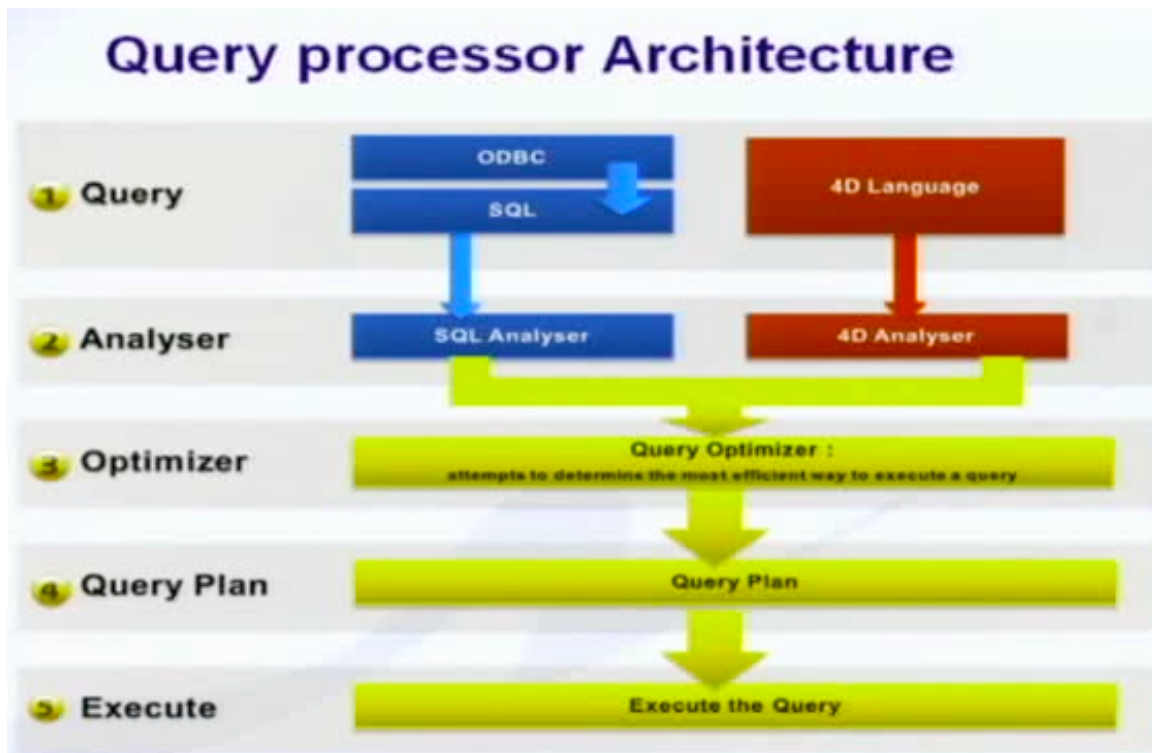
833,333 records found in 268 milliseconds – a huge improvement over 40 seconds! However, as mentioned earlier, this has come at the cost of an increased database size: The index file, INDX, is now 172 MB:

Name	Date Modified	Size
Backup Journal[001].txt	Yesterday, 2:16 PM	4 KB
IndexTwo.4DB	Today, 3:14 PM	516 KB
IndexTwo.4DD	Today, 3:14 PM	660.2 MB
IndexTwo.4DIdx	Today, 3:14 PM	172.4 MB
IndexTwo.4DIndy	Today, 3:14 PM	196 KB
IndexTwo.journal	Yesterday, 2:16 PM	4 KB
► Preferences	Feb 29, 2008, 1:15 PM	--
► Resources	Feb 29, 2008, 1:15 PM	--
► rollback.tmp	Mar 11, 2008, 12:16 PM	--
► temporary files	Today, 3:08 PM	--

Furthermore, our selectivity was much lower than 1, despite the performance gain. This may lead you to believe that there still may be a better way to index this field. We will get back to that in a bit.

The Query Path and Query Plan

Before we move on, let's take a side step to examine exactly what happened during the query.



Following the diagram above shows how a query lives and dies from inception to execution.

In the first step, the query is written in the 4D Method editor. It is then passed to the appropriate query analyzer (4D syntax is passed to the 4D analyzer, ODBC and SQL syntax is passed to the SQL analyzer). The query is optimized and the plan is created. Last, the query executes.

There are two pieces of information that developers can glean from this: The **query path** and the **query plan**. A query's plan and path are analogous to what a road trip might be like. Most of the time, a query's

path will be identical to its plan. However, 4D might implement dynamic optimizations during the actual query while it is being executed to improve performance. This is typically caused when there are few enough records so that performing a sequential search is faster than using the index, even when the optimizer initially plans to use the index.

There are three commands available for use to inspect the query path and plan:

GET LAST QUERY PLAN – This command returns the plan that the optimizer has created for the query being analyzed.

GET LAST QUERY PATH – This command returns the actual query path taken.

DESCRIBES QUERY EXECUTION – This turns query analysis mode on and off. When query analysis mode is on, you can use the above two commands to return the path and plan. Query analysis mode can potentially slow down your application, so be sure to only leave it on when debugging. Try not to include this command in compiled database as well.

The query that we used in the previous example is inserted into the method below:

```
C_INTEGER($docref)
`Turn on query analysis mode.
DESCRIBE QUERY EXECUTION(True)
`Our query for BMW cars
t:=Milliseconds
QUERY([Cars];[Cars]ModelID="BMW")
ALERT(String(Milliseconds-t))
`Save the query plan and path to text variables and then write their
`values to a document.
$tQPlan:=Get Last Query Plan(Description in Text Format )
$tQPath:=Get Last Query Path(Description in Text Format )
$docref:=Append document("test.txt")
SEND PACKET($docref;"Query Plan"+Char(13))
SEND PACKET($docref;$tQPlan)
SEND PACKET($docref;"Query Path"+Char(13))
SEND PACKET($docref;$tQPath)
SEND PACKET($docref;Char(13)) CLOSE DOCUMENT($docref)

`Turn off query analysis mode.
DESCRIBE QUERY EXECUTION(False)
```

The results of using the commands GET LAST QUERY PATH and GET LAST QUERY PLAN are printed below. Though printed within this document, they look almost identical when written to a text file. The instructor will demonstrate how this works. The important thing to know is what information is returned to the developer.

With the index turned off, this is what the query plan and path look like in the text file. Notice that the number of records and the search time are printed out.

Query Plan

Cars.ModelID = BMW

Query Path

Cars.ModelID = BMW (833333 records found in 55400 ms)

With the index turned on, this is the query plan and path:

Query Plan

[index : Cars.ModelID] = BMW

Query Path

[index : Cars.ModelID] = BMW (833333 records found in 254 ms)

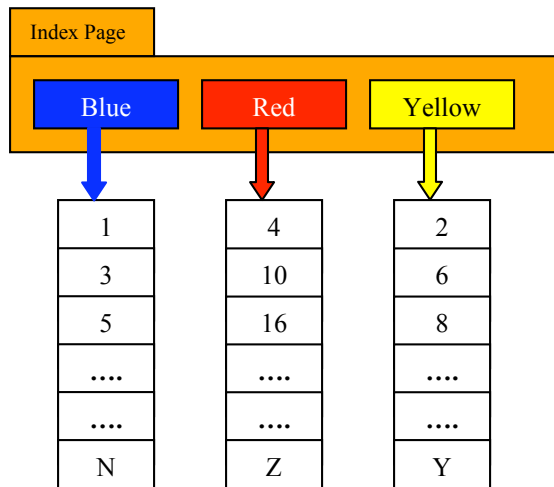
Note that the use of the index is denoted by the “index” that proceeds the table and field name in both the path and the plan. Whether you use an index or not, the total number of records and the time it took to find the records is also printed.

CLUSTER B-TREE

The Cluster B-tree index can improve query performance in many cases where B-tree indexes are ineffective. Specifically, this index is best suited for columns that contain many records and relatively few unique values. A field that contains a short list such as a color is good examples of when the Cluster B-tree index outperforms the B-tree search. Structurally, the heart of the Cluster B-Tree is identical to a B-tree: A root node exists that contains information. If there are more keys than t, other nodes are created. However, this is where the anatomy begins to differ. Whereas the B-tree could contain many nodes with pointers to more nodes and eventually leaves; the Cluster B-tree contains nodes that point to arrays or bitmaps. These arrays and bitmaps are the two different kinds of cluster indexes in 4D.

Cluster B-tree using Arrays

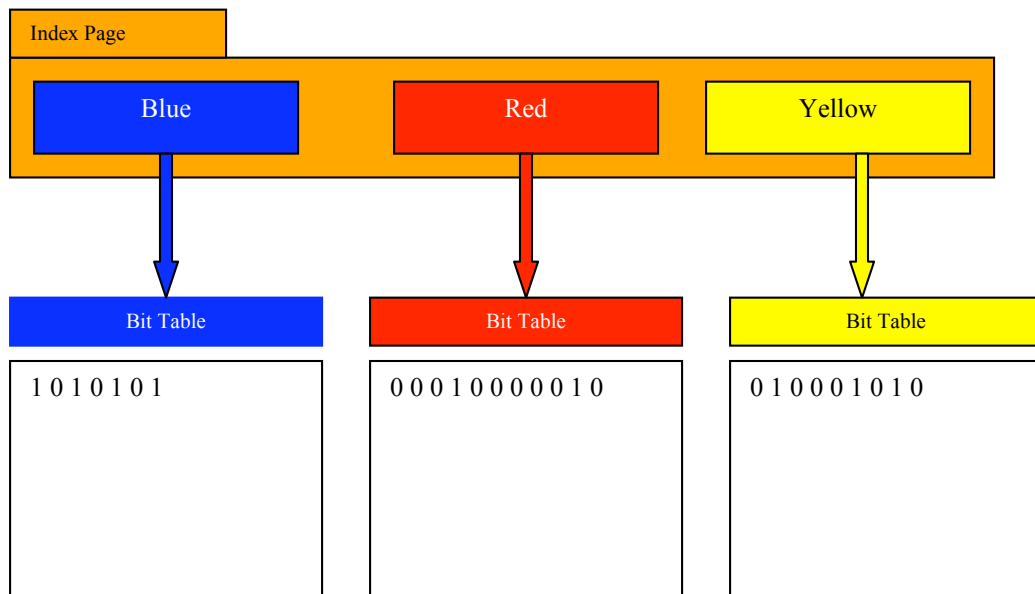
Keys are unique and sorted. Cluster B- Tree indexes contain nodes with search key values and a pointer to an array of long integers which contain keys that match that record number. The following diagram illustrates this:



Unlike the index structure of a B-Tree, the search keys are stored in the arrays instead of the nodes. In this example, there are three unique, sorted keys: Blue, Red and Yellow. In addition to the keys, there are pointers to long integer arrays containing the record numbers that match the key. For example, Records 1, 3 and 5 in the table contain the value “Blue”.

Cluster B-tree using a Bitmap

The second way is through the use of bitmaps instead of arrays. The size of the map – the number of bits in a map – is equal to the total number of rows in the table. Each bit in the map corresponds with a row in table. If the bit is set, in other words “equals 1”, the search key value occurs in that row. When the bit is not set, the queried value does not exist on that row. The following diagram illustrates this:



Using this kind of Cluster B-Tree indexing, you can notice that every single record is represented in each of the bitmaps. Looking at the Blue bit table, bits 1, 3, 5 and 7 are set so those contain “Blue” in their records. In the Red table, Bits 4 and 10 are set, so those contain “Red” in their records.

Use of either a Bitmap or Array

So how does 4D decide between using arrays and using a bitmap? A calculation is performed when the index is built on the field being indexed and 4D chooses the one that requires less space. Here is how it is calculated.

An element in an array of longints takes up 4 bytes, or 32 bits, of space. A bitmap will be n bits long, where n is the total number of records in the database.

For example, say you have a table with 1 million records, and only 50 of them contain the value “Yellow.”

To calculate the space required if using an array of longints:

[Number of Matches] x 4 bytes = $50 \times 4 = \underline{200 \text{ bytes}}$

To calculate the space required if using a bitmap:

[Total records] / 8 bits per byte = $1,000,000 / 8 = \underline{125,000 \text{ bytes}}$

So if you have 1 million records, and only 50 of them contain the value “Yellow”, it is less space-expensive to use an array of longints than it is to use a bitmap. As you can see, it will take quite a few more than 50 records before the size of the array eclipses the size required to create a bitmap. Specifically, it would have taken more than 31,250 records that contained “Yellow” to make 4D use a bitmap. Following the calculation for the space required using an array. For example, let’s say we have one more record (31,251) that contain yellow:

To calculate the space required if using an array of longints:

[Number of Matches] x 4 bytes = $31,251 \times 4 = \underline{125,004 \text{ bytes}}$

This value and anything above it would cause the space requirement for the array to exceed that of the bitmap. In this case, 4D would use a bitmap. Note that this is for a per-unique-value basis. The example demonstrated what would happen to the Yellow values. If Blue or Red contained more than 31,250 matches in the database, they would use a bitmap – in other words, bitmaps and arrays can be used together in the same index.

Query Cars[ModelID] With a Cluster B-Tree Index

Remember how, in the previous section, it took about 42 seconds to query a field that only had a few unique values? That test was followed up with a B-tree indexed search that returned the same results in 1 second.

Although that was a vast performance gain, we also noted that the selectivity of the table was 0.0000008. Optimal performance occurs in a B-tree index query when selectivity equals 1. This indicated that there may be a more optimal way to query this field despite the reduced query time that the index provided.

Now let’s put the Cluster B-tree to the test. Again, we are going to perform the same query – find all BMW cars using the following code:

The same 833,333 records are found, but notice the time that that it took – 15 milliseconds, which is an even more impressive gain than before.

Furthermore, the size of the INDX file is less than 3 megabytes; far less than the 180 megabyte index file using the B-tree index. Remember this is using the same number of records!

Name	Date Modified	Size
Backup Journal[001].txt	Mar 17, 2008, 2:16 PM	4 KB
IndexTwo.4DB	Yesterday, 5:17 PM	516 KB
IndexTwo.4DD	Yesterday, 5:17 PM	669.2 MB
IndexTwo.4DIndx	Yesterday, 5:17 PM	2.6 MB
IndexTwo.4DIndy	Yesterday, 5:17 PM	196 KB
IndexTwo.journal	Mar 17, 2008, 2:16 PM	4 KB
▶ Preferences	Feb 29, 2008, 1:15 PM	--
▶ Resources	Feb 29, 2008, 1:15 PM	--
▶ rollback.tmp	Mar 11, 2008, 12:16 PM	--
▶ temporary files	Yesterday, 5:16 PM	--

The query path and plan are identical in this query, as well:

Query Plan

[index : Cars.ModelID] = BMW

Query Path

[index : Cars.ModelID] = BMW (833333 records found in 0 ms)

Zero milliseconds? Why the discrepancy between this and the alert time of 15 milliseconds? Here is the query printed again:

Remember, GET LAST QUERY PATH returns the time needed for every query criteria to be found. Compare this to our alert statement, which takes the time acquired right after the query then subtracts the time right before the query and you can see how there is probably a few millisecond delay setting the variable, *t*, to accept the time as well as calling an Alert out.

Let this be a lesson that choosing the right index can dramatically improve the performance of the database. Between the B-Tree and Cluster B-Tree, selectivity is a good indicator of which index you should use. If you have a field that contains many unique values such that selectivity is close to 1, use the B-Tree index. Otherwise, if you only have a field that contains few unique values, use the Cluster B-Tree.

B-trees are still useful!

Despite being outclassed in the previous section, B-trees can perform better than Cluster B-trees under certain situations. Here is a query that demonstrates this.

RecordNumber has a selectivity of 1. In other words, each value is unique per record.

QUERY ([Cars [; [Cars] RecordNumber < 4000000)

Here is the query path and plan using a B-Tree index on the field:

Query Plan

[index : Cars.RecordNumber] < 4000000

Query Path

[index : Cars.RecordNumber] < 4000000 (3999999 records found in 216 ms)

And here is the query path and plan using a Cluster B-tree index:

Query Plan

[index : Cars.RecordNumber] < 4000000

Query Path

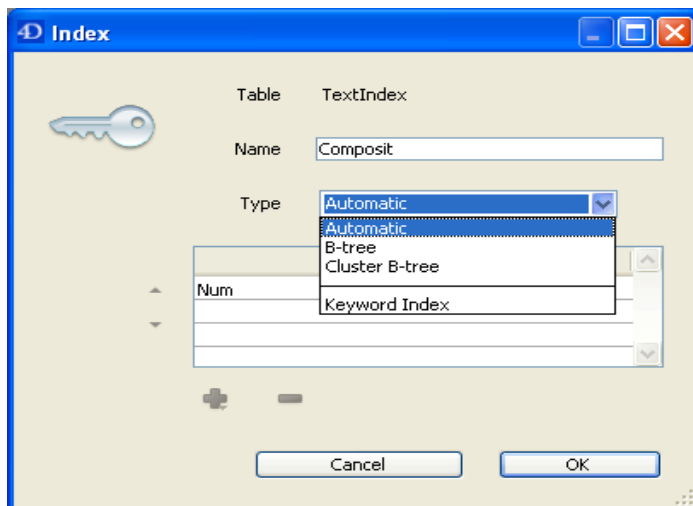
[index : Cars.RecordNumber] < 4000000 (3999999 records found in 270 ms)

The B-Tree performs better in this situation.

COMPOSITE INDEX

Composite indexes combine several fields together and index their values in either a B-tree or Cluster B-tree structure. Composite indexes are good for sorting especially large selections. It is also good for queries on several fields where only a small number of records are returned. For example, if you want to do a query on two fields, you have a choice to do an index search on the first field, then do an index search on the second field and join the two queries. So let's say the first search returns one million records, the second search returns five hundred thousand records and the join between them returns only ten records. With a Composite index, internally the index will go through the pages which contain the exact match of the keys. If you make a query on multiple fields, regardless of the field order in the query, 4D will try to find if there are fields associated with a composite index and will give the composite index priority.

The Composite index reduces the number of queries at the database level and frees you from maintaining compound fields.



First a query using two separate indexes

In our example first will index "model" field with btree index and "color" field with cluster index. The query is for blue BMWs as follows:

```
QUERY ([Cars]; [Cars]ModelID="BMW"; *)  
QUERY ([Cars]; & ; [Cars]ColorID="Blue")
```

And here is the path and plan:

Query Plan

```
[index : Cars.ModelID ] = BMW  
And  
[index : Cars.ColorID ] = Blue
```

Query Path

```
AND  
[index : Cars.ModelID ] = BMW (833333 records found in 146 ms)  
[index : Cars.ColorID ] = Blue (69267 records found in 0 ms)  
--> 69267 records found in 158 ms
```

First 4D will find all Model equal BMW and join this result will return all colors equal Blue.

Query With a Composite Index

Now we will see the composite index in action. A composite index of type B-tree is placed on the two fields, ModelID and ColorID. The query stays the same, but here is the path and plan:

Query Plan

```
[index : ModelID, ColorID ] LIKE BMW , Blue
```

Query Path

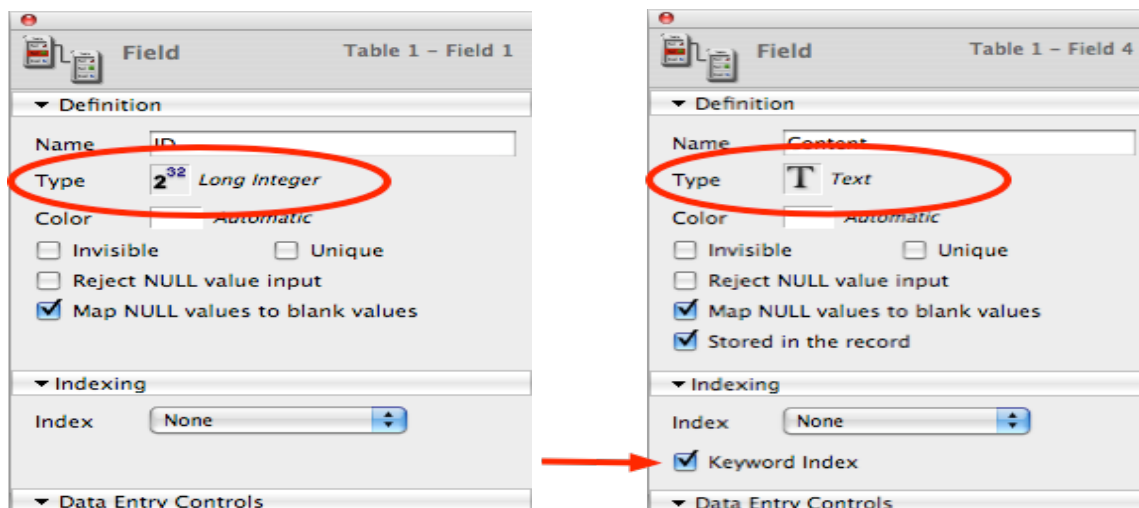
```
[index : ModelID, ColorID ] LIKE BMW , Blue (69267 records found in 4 ms)
```

The same query took only 4 ms!

You can have composite index based on b-tree or based on cluster, both are available in v 11. Composite index is good for sorting especially in large selection. Also is good for queries on several fields where as a result return small selection of records.

KEYWORD INDEX

The last indexing type available for use is the keyword index. This type of index is only available for alpha or text fields. The option in the Inspector is not available when looking at a field of other types.



When an Alpha or Text field is indexed using the Keyword Index, every word inside the field is indexed. It is built using clusters. This includes short and single-letter words such as “a” and “the”. This obviously impacts the size of the index file, but the performance gain in queries provides a decent counterpoint to that argument. More on this in a bit.

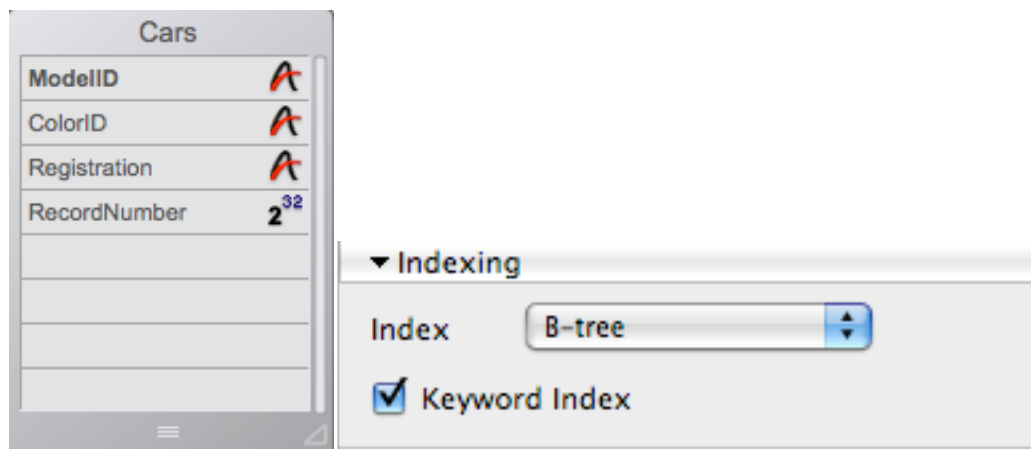
You can perform a search on the entire text or alpha field for keywords. This query will be applied to full words only, and not just a sequence of characters such as “-“, “/”, etc. Note that it is possible to perform a keyword query without an index, but the query executes much slower when it is performed on a field without the keyword index.

Because of the new keyword search, a new operator, %, has been added that can be used in Query commands.

```
QUERY ({aTable}{; queryArgument{; *}})
QUERY BY FORMULA ({aTable}{; }{queryFormula})
QUERY SELECTION BY FORMULA (aTable{; queryFormula})
```

If you use Keyword index together with other available indexes the optimizer will make decision based on the operator you use.

Here is an example:



As we have seen, the ModelID field contains only a few unique values. We determined that the Cluster B-tree index was better than the B-tree index.

The field has both the Keyword index and B-Tree index enabled. The size of the INDX file is 172MB.

Using the “=” Operator:

In the method editor, the following query is executed.

```
QUERY ([Cars]; [Cars]ModelID="Porche")
```

How do we know that the B-tree index was used instead of the Keyword index? The following section shows what it looks like when the Keyword index is used.

Using the “%” Operator:

Now let us execute the same query but use the keyword operator instead:

```
QUERY ([Cars]; [Cars]ModelID%"Porche")
```

Use of the “=” sign forces the query forces the B-Tree index to be used, as seen here in the query’s path and plan:

Query Plan

[index : Cars.ModelID] = “Porsche”

Query Path

[index : Cars.ModelID] = “Porsche” (1666667 records found in 450 ms)

This will force the keyword index to be used as shown here:

Query Plan

[index : FullText : Cars.ModelID] = “Porsche”

Query Path

[index : FullText : Cars.ModelID] = Porsche (1666667 records found in 0 ms)

As you can see, when an Keyword index is used, the path and plan show “index : FullText :”. When any other index is used, the path and plan just show “index :”. Also note that the seek time using the Keyword index is much faster than using a B-tree index.

Using the “%” Operator on an un-indexed field:

As mentioned earlier, it is still possible to do a keyword search on an Alpha or Text field that does not have the Keyword index enabled. When taking the index off in the example above and executing the same query, this is the resulting path and plan:

Query Plan

Cars.ModelID contains Porsche

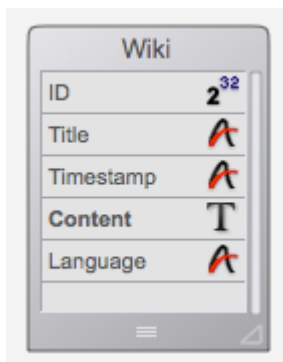
Query Path

Cars.ModelID contains Porsche (1666667 records found in 71582 ms)

As you can see, using the index saved us over a minute of processing time. Also note that the other index was not used, even though it was available. In other words, the keyword search operator, %, can only be used with the Keyword index or no index at all. So as we have seen, Keyword indexes can speed up searches on Alpha and Text fields similar to cluster indexes. This was demonstrated above. But let’s take a look at how this behaves on a grander scale – remember that Keyword indexes really shine when searching within a block of text.

Example:

The database contains only a single table, [Wiki].



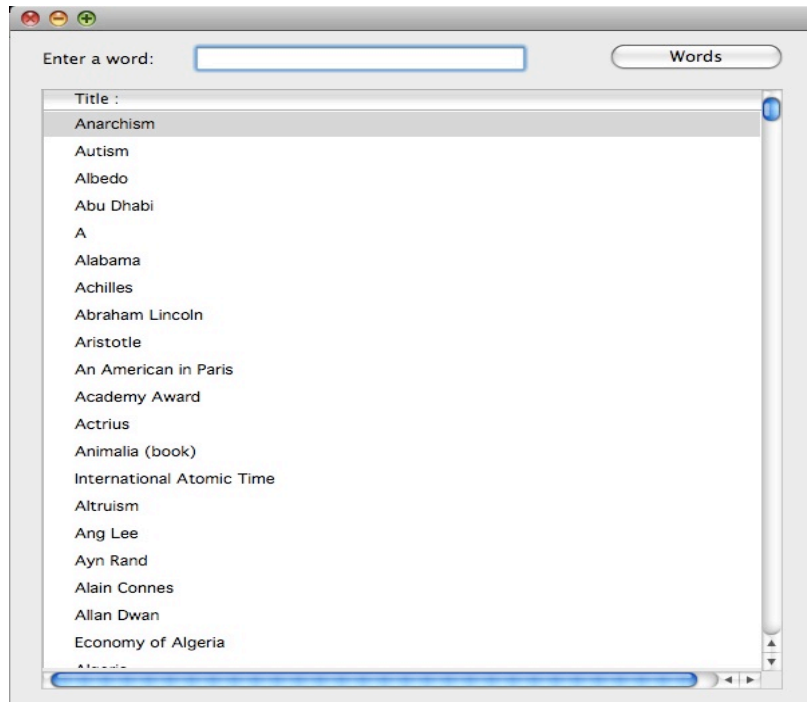
Wiki	
ID	2 ³²
Title	A
Timestamp	A
Content	T
Language	A

4 Wiki_Fulltext EN - Wiki: 2278408 of 2278408		
	Timestamp :	C
	2007-05-28T15:09:05Z	,
	2007-05-28T09:47:13Z	

This table contains the full collection of Wikipedia articles as of May 2007, which amounts to over 2.2 million records. This number itself is hardly stretching the limits of what 4D is capable of holding, but when you consider that this is 2.2 million complete Wikipedia articles, the magnitude of it all is impressive; it takes a good amount of disk space to hold this database. The datafile and index alone take up 22 gigabytes of storage.

Name	Date Modified	Size	Kind
Logs	Yesterday, 12:06 PM	--	Folde
Preferences	May 9, 2007, 7:47 AM	--	Folde
Resources	May 8, 2007, 8:05 AM	--	Folde
temporary files	Today, 9:51 AM	--	Folde
Wiki_Fulltext_Repair_log.html	May 23, 2007, 1:31 AM	16 KB	HTM
Wiki_Fulltext_Repair_Log.xml	May 23, 2007, 1:31 AM	4 KB	Word
Wiki_Fulltext.4DB	Today, 9:52 AM	588 KB	4D Ir
Wiki_Fulltext.4DD	Today, 9:19 AM	12.09 GB	4D D
Wiki_Fulltext.4DIndx	Today, 9:20 AM	10.01 GB	Docu
Wiki_Fulltext.4Dindy	Today, 9:52 AM	196 KB	Docu
Wiki_Fulltext.journal	Jun 9, 2007, 6:54 AM	4 KB	Plain

Perhaps even more impressive is the following demonstration. Inspecting the table, you can notice that the Content field contains the Keyword index. There are no other indexes in this database. The Content field contains the body of the Wikipedia articles.



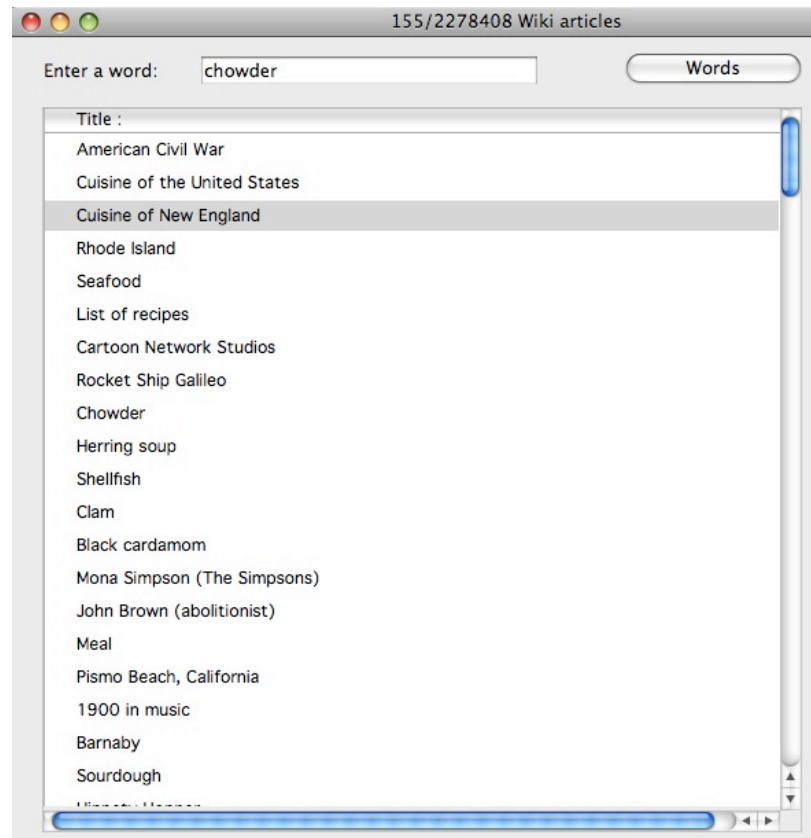
On the main form above, there is one field for users to enter a search. The space below that field is used to display the article titles for records that contain this word. Here is the code that is executed whenever a character is entered into this field, marked “Enter a word.”:

```

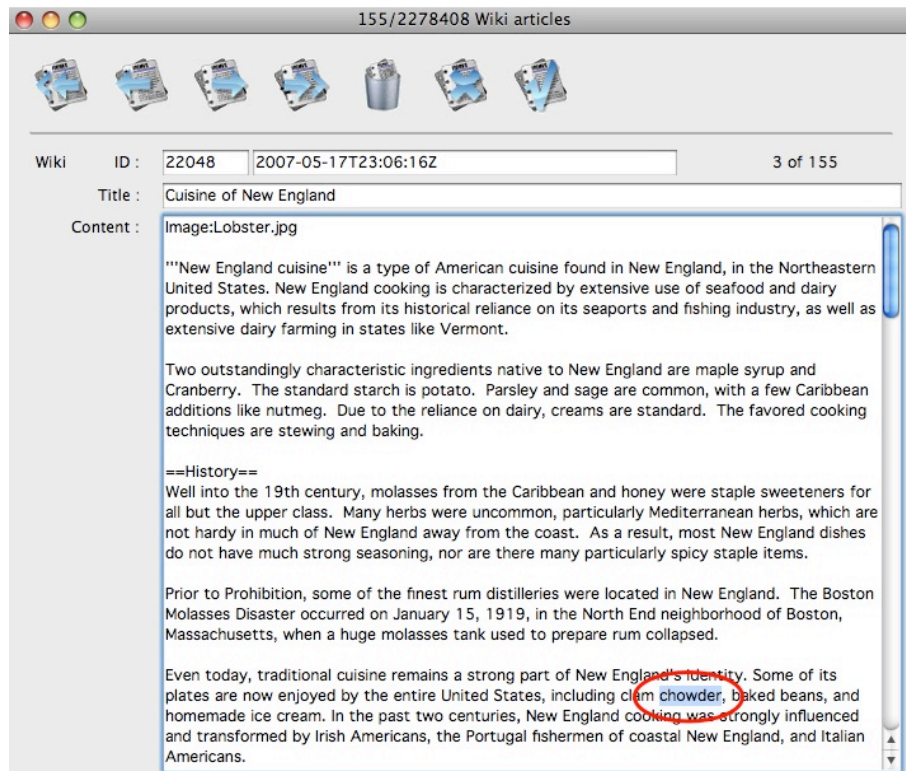
If (Form event=On After Keystroke )
    $word:=Get edited text
    If (Length($word)>2)
        If ($word#"")
            $word:=$word+"@"
            QUERY([Wiki];[Wiki]Content%$word)
        Else
            ALL RECORDS([Wiki])
        End if
        $text:=String(Records in selection([Wiki]))+"/"+String(Records in
table([Wiki]))+" Wiki articles"
        SET WINDOW TITLE($text)
    Else
        REDUCE SELECTION([Wiki];0)
        $text:="Wiki articles"
        SET WINDOW TITLE($text)
    End if
End if

```


Every time a letter is entered into the field, 4D performs a query. Once the index is loaded into cache, the type-ahead search happens very quickly. The titles in the window space are updated on the fly.



The list of articles in the lower portion of the window all contain the word that was entered! In this example, the word “chowder” was entered, and double clicking one of the results – Cuisine of New England – brings us to the body of the article, with the first instance of the word “chowder” highlighted for us:



By pressing the “Words” button, 4D will take a few seconds to build an array using DISTINCT VALUES with ALL words contained in the articles returned in the query! Again, this act takes only a few seconds to finish executing. This demonstration should be ample proof that the Keyword index in 4D v11 SQL provides a practical solution for querying large bodies of text that would have been painstakingly sluggish in 4D 2004.

CONCLUSION

Generally speaking, B-tree indexes work best when selectivity is high. Cluster B-tree indexes work best when selectivity is low. Keyword indexes work best when searching a Text or Alpha field, and are based on Clusters. Last, composite indexes index values from multiple fields and can be built on either B-tree or Cluster B-tree structures. In any case, a thorough developer can check which indexes are being used and which indexes perform faster on a field using DESCRIBE QUERY EXECUTION and its associated commands. Using these simple facts and the details in this document as a sample, you should now have a comfortable grasp on the new indexing schemes and how to put them to use in 4D v11 SQL. Happy, and swift, querying! In the new version 11 there is a new option called “Rebuild indexes after import” which will saves you time during import of large amount of data.