

New Index Options

Overview

4D 6.5 includes several changes and new features that improve the performance of indexes, reduce the time it takes to build indexes, and give you more control over how your indexes are built. This chapter explains what indexes are, how they are organized in 4D, and how to understand and use the new 4D 6.5 indexing features.

Background

4D Indexes Contain Copies of Field Data

4D indexes are not part of 4D records. They are maintained independently as separate structures in the data file. A field index includes a *copy* of all the field data along with record numbers. The index improves query and sort performance in several ways:

- ✓ 4D can find the matching records without loading the individual records. This feature reduces the amount of data that needs to be read off disk, thereby directly affecting performance.
- ✓ Records are stored in random order on disk. Indexes sort and structure the field data for quick access.
- ✓ Indexes are very compact. Since indexes take less space than full records in memory, the 4D cache is more likely to be able to keep the indexes in memory. Reading data that is already in memory can be many times faster than reading data off disk.

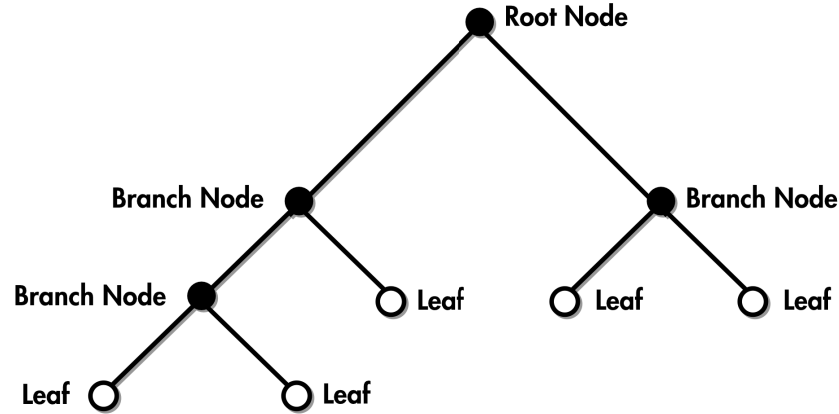
These benefits do not come without costs:

- ✓ Indexes take extra space to store, thus increasing the size of your data file.
- ✓ In order to be useful, indexes must be perfect at all times. Every time a record is added, modified, or deleted, all the affected field indexes need to be updated. This requirement adds some overhead to all record modification operations.

When you need to decide if the benefits of an index justify its costs, it helps to understand what indexes are and how they are implemented in 4D.

What Is an Index?

Field indexes are a structured representation of the data in the fields. Indexes are often called trees because they are organized with a root and any number of branches and leaves.



A tree structure.

Index Terminology

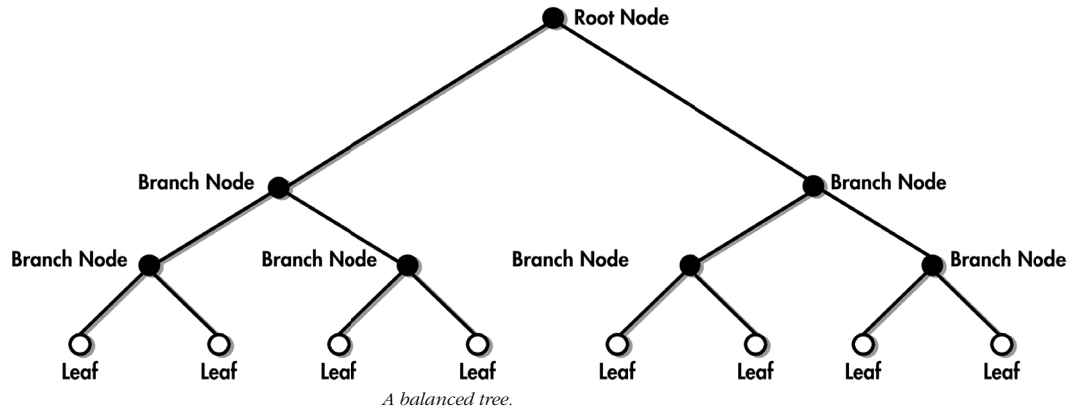
You will hear different terms for the different parts of a tree structure. This table explains some of the synonyms:

Term	Note
Node Page	Node is a general term for the parts of a tree that do not contain final values. In other words, a node is anything that is not a leaf. You may see 4D nodes referred to as nodes or pages.
Root node Root Top node Primary node Primary page	Trees start from a single node, normally called the root or top. 4D indexes include a primary index node (page) that includes the locations of branch (secondary) index nodes.
Branch Branch node Branch page Secondary page	Branches are nodes that are not the root node and not leaves. Described another way, branch nodes are nodes that have roots and leaves. 4D's secondary index pages can also be defined as branch nodes.
Leaf Terminal node Value	Trees terminate in leaves. 4D indexes include record numbers and field values as leaves.
Parent	The node immediately above the current node or leaf.
Child	The node immediately below the current node. Leaves, by definition, do not have children.

Kinds of Trees

There are many kinds of possible trees. One detail you may notice about the tree pictured above is that one side is deeper than the other. This is called an *unbalanced* tree. If you are parsing an algebraic expression into a tree to prepare it for evaluation, you are likely to end up with an unbalanced tree of this kind. 4D uses a *balanced* tree—a B-tree—for

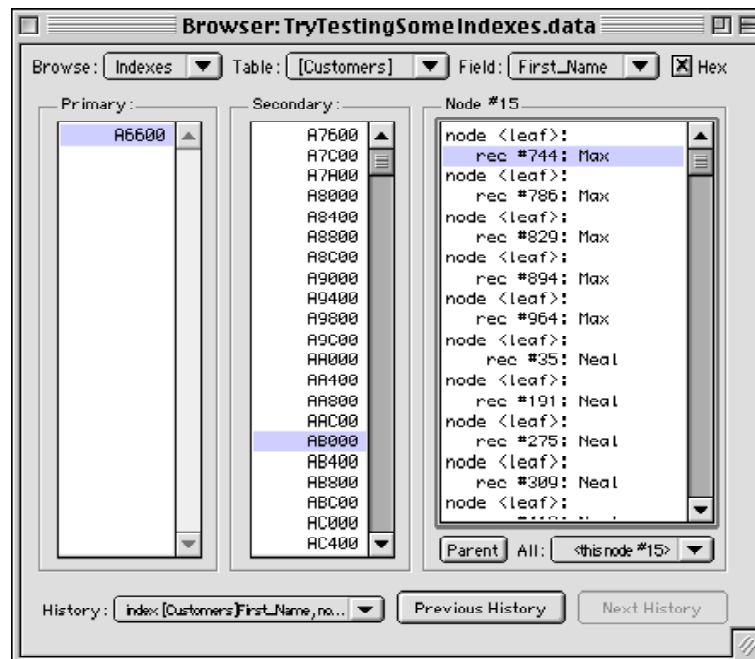
its indexes. Each branch of a balanced tree always has exactly the same depth. B-trees are a good general purpose tree structure, particularly well suited to indexing fields with few repeated values.



Inside a 4D Index

Viewing a 4D Index

4D, Inc. does not provide any tools that let you look inside an index to see how it is built. The third party tool DataCheck, available from <http://www.committedsoftware.com/>, includes a Browser that lets you inspect records and indexes directly. Here is how a 4D index appears in DataCheck.



Browsing a first name index in DataCheck.



The "4D Tools 6.5" chapter includes an overview of DataCheck, starting on page 44.

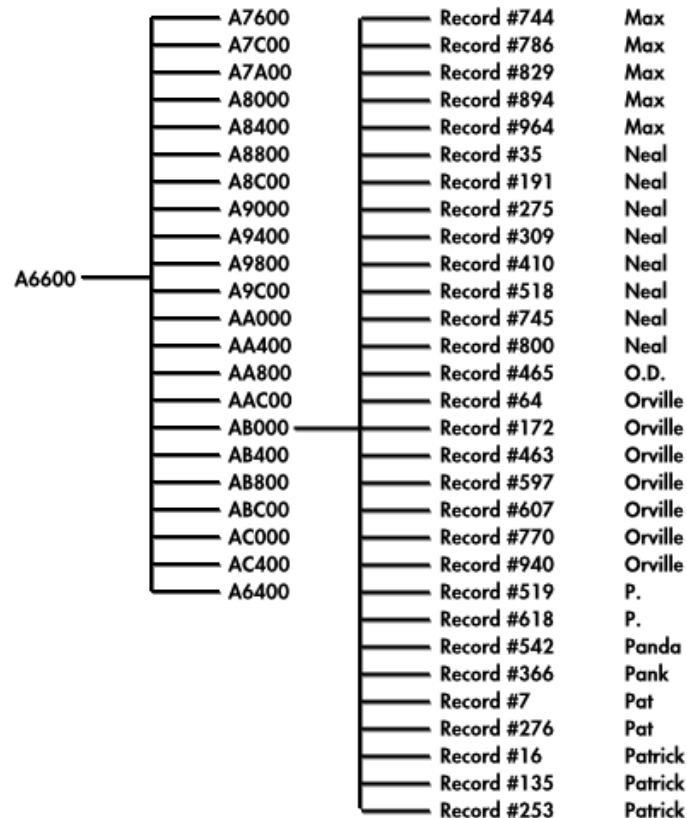
Chapter 28 - New Index Options

Here are some details you can learn by examining your indexes with DataCheck:

- ❖ Conceptually, there is a primary index node and a collection of secondary index nodes. (Internally, all nodes have the same structure.)
- ❖ The primary index node contains references to secondary index nodes.
- ❖ Secondary index nodes contain references to additional secondary index nodes or include record values. References and values are not mixed within a single node.
- ❖ The numbers you see for the primary and secondary nodes are offsets within the data file. These numbers allow 4D, 4D Tools, 4D Server, and DataCheck to locate index pages.
- ❖ Each record has one entry in the index for anything other than a subfield.
- ❖ Each record has one entry in the index for each subrecord. In other words, a record can have multiple entries in the index, corresponding to subrecord values. For this reason, subrecord queries are fast.
- ❖ Entries are sorted in ascending order by record number within an index page.
- ❖ Nodes within the same index can have different numbers of entries. The index does not have a fixed node size.
- ❖ Nodes may contain blank entries.

Schematic View

Let us look at the index again schematically:



Controlling the Entries per Index Node (Or: “What does that slider do?”)

The main index node A6600 is the parent of the twenty-two secondary index nodes shown. Each individual secondary node includes some number of node or record references. The exact number of references in the node has a direct impact on performance and is something you can configure with the 4D 6.5 index settings dialog.

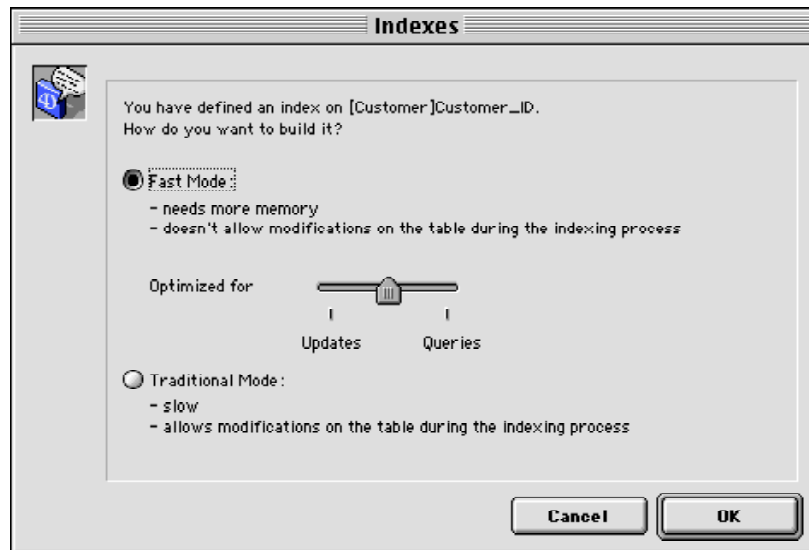
Controlling the Entries per Index Node (Or: “What does that slider do?”)

Overview

4D 6.5's index settings it is possible to control the number of entries placed in each index node as the index is built. This section explains why different node sizes optimize different operations and what the new fast mode index option actually does.

The Reindexing Dialog and SET INDEX

When you choose to index or reindex a field in a table with 1,000 or more records, 4D displays this dialog:



The 4D 6.5 index building dialog.

The **SET INDEX** command has been modified to provide programmatic access to the features of the 4D 6.5 index building dialog. The following line of code uses the fast mode to build an index optimized for queries:

```
SET INDEX([Customer]First_name;True;100)
```

The third parameter is a percentage that corresponds to the slider pictured above. 0% means updates and 100% means queries. Most programmer's first question when they see this dialog is: “How should I set the slider?” or “What does the slider change?” The slider changes the number of entries in each index node. Here is how the slider would look if it showed the node size:



Hypothetical node size slider.

Chapter 28 - New Index Options

4D uses a slider in the reindexing dialog, and percentages with the **SET INDEX** command, instead of allowing the developer to specify the number of nodes directly. This is a deliberate design choice. 4D is expected to change the underlying index structure in the future and does not want us to tie our code to exact node settings. The following table explains how the node size is changed by various percentages in 4D 6.5:

SET INDEX Percentages and Resulting Node Sizes

Percentage	Slider Position	Node Size
100	Queries	63
90		60
80		57
70		54
60		51
50	Default	48
40		44
30		41
20		38
10		35
0	Updates	33

If you want to build an index in the traditional mode, call **SET INDEX** without including the new optional percentage parameter:

```
SET INDEX((Customer)First_name;True)
```

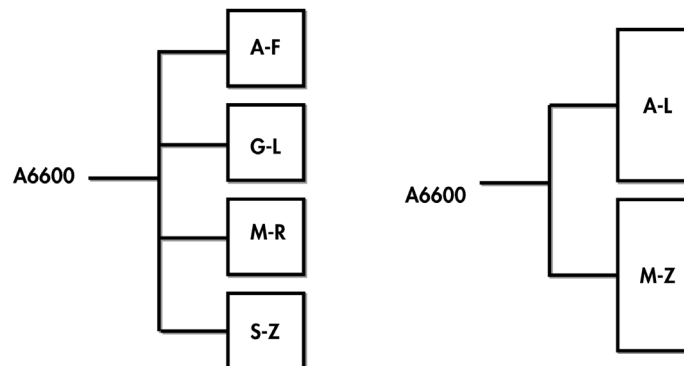
4D builds the index using the traditional mode and sets the average node size according to its own best guess.



*If you are converting a database, any existing calls to **SET INDEX** will not have the new percentage parameter. If you want to build indexes using the fast mode, you need to modify the routines that use **SET INDEX**.*

Node Size Example

The index building dialog sets the node size to 33 to optimize for updates, and to 63 to optimize for queries. Why are smaller nodes better for updates and larger nodes better for queries? We will compare operations on two versions of the same index. The version on the left has small pages (update optimized) and the version on the right has large pages (query optimized.)



Two versions of the same index.

Controlling the Entries per Index Node (Or: “What does that slider do?”)

Why Larger Pages Optimize Queries

Large pages are better for queries because 4D is likely to find matching values with fewer reads from disk. Imagine, for example, that you are looking for the value “Open” in the index pictured above. Here is what has to happen with each version of the index:

Small Pages	Large Pages
Read primary index node.	Read primary index node.
Read and examine index page with values between A and F.	Read and examine index page with values between A and L.
Read and examine index page with values between G and L.	Read and examine index page with values between M and Z.
Read and examine index page with values between M and R.	

This artificially simple example illustrates how larger pages can reduce the number of disk accesses required to locate a matching value. At the level of a database engine, reading data from the hard drive is very expensive. Many engine-level optimizations are nothing more than tricks to avoid reading and writing to disk. The cost of reading from disk depends on the hard drive’s speed, the computer’s speed, the speed of the computer-hard drive interface, and the operating system used. Windows NT, for example, has a higher performance file access system than the Mac OS.

This example also illustrates that larger pages can require more memory. In the example just listed the small page solution did not need to load the values between S-Z. The small differences between the number of entries in an index node become significant because they are repeated hundreds or thousands of times as the database is used.



Large pages are better optimized for queries also because 4D can skip more entries at a time when the page cannot contain a matching value. This subject is discussed in this chapter in the “Index Inspection” section, beginning on page 258.

Why Smaller Pages Optimize Updates

Smaller pages can be faster to maintain because 4D is less likely to have to split nodes as values are added, modified, and deleted. And, if nodes are split, they are smaller and quicker to build and move.

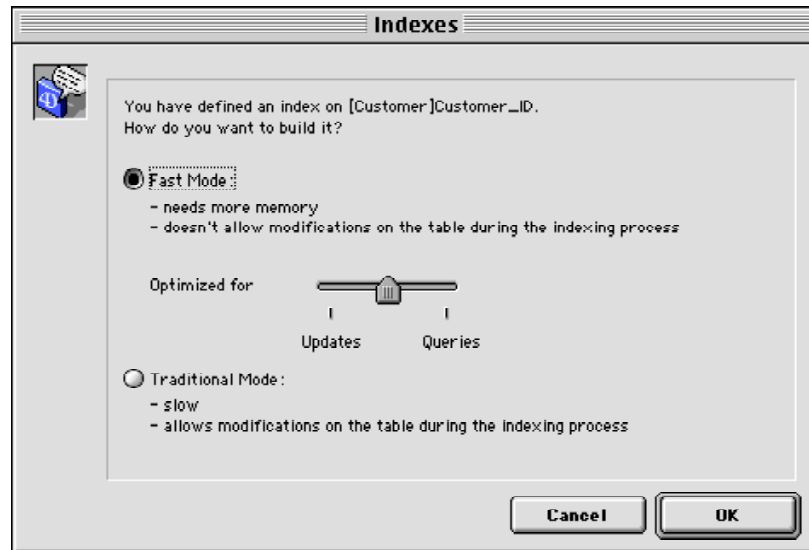
Can I Update a Query Optimized Index?

Developers often ask if selecting **Optimize for queries** means that they can no longer add, delete, or change records. It is always possible to add, delete, and change records, no matter which indexing options are selected. Once 4D builds the index, the indexing mode is forgotten. It is not saved with the index, and it has no effect on how updates are performed.

Fast Mode Versus Traditional Mode

Overview

The 4D 6.5 indexing system lets you choose **Fast Mode** or **Traditional Mode** while you are building or rebuilding an index:



The 4D 6.5 index building dialog.

This section explains the differences between these two modes.

The Indexing Changes, the Index Does Not

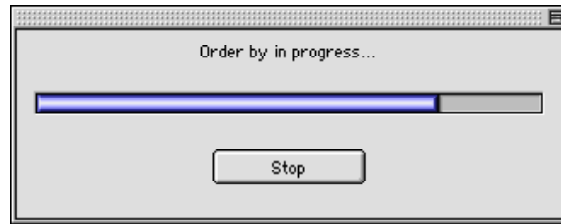
The fast mode indexing option changes how the indexing is performed internally, but not the contents of the resulting index. The fast mode option prompts 4D to run a different set of code while it is building the index, but this code does not produce a different kind of index from the traditional mode. Once the index is created, 4D does not know or remember how the index was originally built. All updates are performed using the traditional method, regardless of how the index was originally built. 4D has only one type of index, and it can be built in fast mode or traditional mode. So what is different about fast mode?

- ❖ Fast mode locks the index while it is being built. Accordingly, other processes cannot update the index while the index is being built or rebuilt. The advantage of this feature is that the index can be built very quickly, and the pages can be constructed in as ideal an order as possible.
- ❖ Fast mode gives you access to the slider that controls the number of entries put in each index node.

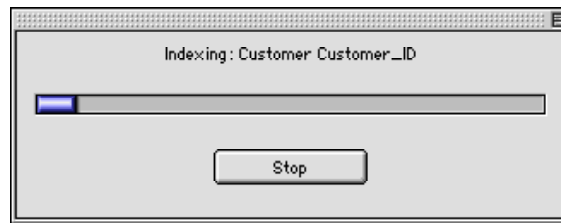
Fast mode is faster because it scans and orders field data before building the index. This is why you see 4D make three passes through your data when it is building an index in fast mode:



Fast mode step 1: Gathering Data.



Fast mode step 2: Ordering data.



Fast mode step 3: Indexing Data.

Traditional Mode

Traditional mode indexing builds the index one record at a time. Each record is added to the index individually. It is much faster to rebuild indexes for an entire table with fast mode.

Is Fast Mode Dangerous?

4D 6.5's fast mode indexing option included a highly specific bug until version 6.5.3. If you are using 4D/4D Server 6.5, either upgrade to 4D 6.5.4 or later, or reindex using traditional mode. There is no technical reason not to upgrade to 6.5.4 or later, and that is my recommendation. This description will help you determine if you might have encountered this bug:

- ❖ You are building an index on a field with non-unique values. Examples of fields likely to have valid duplicate values include Booleans and name fields.
- ❖ The individual records in the table are large (over 100-128K.) You will not have records this large unless you have large data types in the record: BLOB, text, or picture.
- ❖ When the index is built using fast mode, sometimes the entries in a node are sorted in descending order instead of the usual ascending order.
- ❖ When 4D updates the index, it assumes that the entries are sorted in ascending order, as they should be. When a missorted index node is updated, values can be put in the wrong place. This is a form of corruption.
- ❖ Corrupt indexes do not return the correct results when querying, sorting, using **Find index key**, or performing other index based operations.

Chapter 28 - New Index Options

This bug was a serious problem for some developers. It has now been corrected. If you have heard about this bug, note that:

- ❖ This is a highly specific bug.
- ❖ The bug has been corrected.

If you are using 4D 6.5.3 or later, I recommend using fast mode indexing when you want to tune the size of your index pages. If you do wish to rebuild your indexes using traditional mode, apply this code to each indexed field:

```
` Remove the existing index:  
SET INDEX([Table]Field;False)  
  
` Rebuild the index using traditional mode:  
SET INDEX($fieldPointer_p->;True)
```



Paul Carnine, the author of DataCheck and SanityCheck, diagnosed this indexing bug. His work helped 4D, Inc. correct the problem swiftly. DataCheck 2.0.2 and later check for this problem and report it. DataCheck is discussed in the "4D Tools 6.5" chapter, starting on page 44.

When 4D Automatically Uses Fast Mode

When 4D or 4D Tools rebuild indexes in tables with 1,000 records or more, they automatically use the fast mode index building mode. The exact number of entries per node is set according to internal algorithms. 4D has always adjusted the entries per node when building indexes. 4D 6.5's fast mode allows us to specify the node size.

Index Maintenance

Indexes are Always Maintained in Traditional Mode

The method you use to build the index has no effect on how the index is maintained. This is an important point that is easy to miss, so here it is again: **The method you use to build the index has no effect on how the index is maintained.** Indexes are always maintained using traditional mode. In other words, the index is updated one entry at a time as records are modified, added, and deleted. Over time the average number of entries in each index node may change. Rebuilding indexes from scratch occasionally may improve performance.

Tuning Indexes With Code

Data files start with no data. As your users add data, you may want to rebuild the indexes periodically. This process is easily done with code. This routine rebuilds all existing indexes to optimize them for queries:

```
C_LONGINT($tableNumber_l)  
C_LONGINT($fieldNumber_l)  
C_LONGINT($fieldDataType_l)  
C_LONGINT($stringFieldLength_l)  
  
C_BOOLEAN($fieldsIndexed_b)  
  
C_POINTER($fieldPointer_p)  
  
` All indexes will be built optimized for queries.  
$indexPercentage_l:=100  
  
For ($tableNumber_l;1;Count tables)  
  
  For ($fieldNumber_l;1;Count fields($tableNumber_l))  
  
    $fieldPointer_p:=Field($tableNumber_l;$fieldNumber_l)
```

```

GET FIELD PROPERTIES($fieldPointer_p;$fieldDataType_1;$stringFieldLength_1;$fieldsIndexed_b)

If ($fieldsIndexed_b)
  ` Clear the existing index.
  SET INDEX($fieldPointer_p->;False)

  ` Rebuild the index:
  SET INDEX($fieldPointer_p->;True;$indexPercentage_1)

End if ` ( $fieldsIndexed_b)

End for ` (($fieldNumber_1;1;Count fields($tableNumber_1))

End for ` ($tableNumber_1;1;Count tables)

```

You can make this code available to a database administrator or run it automatically at off-peak hours.

4D 6.5 Index Optimizations

Overview

4D 6.5 optimizes caching index pages and inspecting index pages during a query. These are automatic internal optimizations that you cannot control. Learning about them gives you insight into why 4D 6.5 is faster than earlier versions of 4D.

Caches

Caching is a fundamental optimization technique. Copies of objects are kept in faster memory to save the time spent fetching them from slower memory. The 4D/4D Server record and index cache is a well written cache. It is reasonably intelligent about what it keeps in memory and what it discards. 4D 6.5 refines how index pages are handled by the cache. Index pages are very valuable to the cache because they include a collection of values. The chances are relatively high that an index page which is already in the cache will be needed again shortly. When the cache is getting full, 4D/4D Server has to remove something. Because of their value, the 4D/4D Server 6.5 cache compresses index pages in memory rather than discarding them, if possible. Compressing and decompressing the pages in RAM is significantly faster than discarding the pages from memory and reloading them from the hard drive. This kind of small-scale optimization can deliver big performance differences when implemented in something as low-level as the record cache.



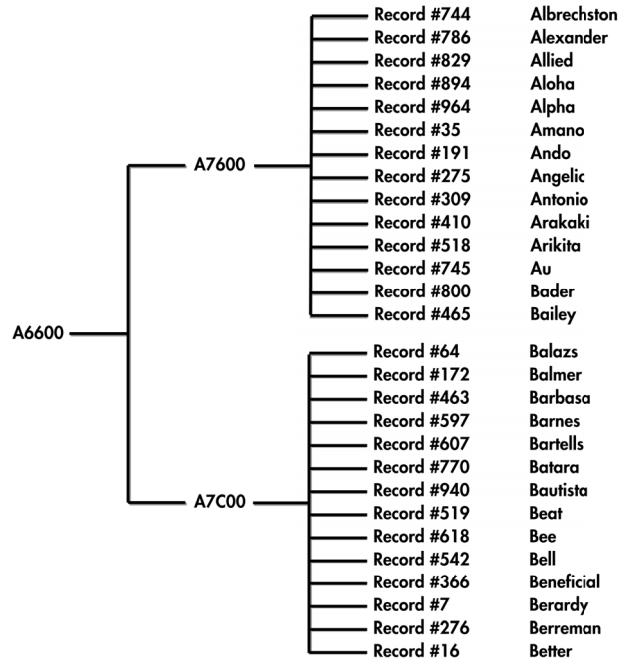
*The index page compacting feature can be turned on and off with **SET DATABASE PARAMETER**, as discussed in the “Tuning Database Parameters” chapter, starting on page 259.*



Programming 4th Dimension: The Ultimate Guide discusses caching extensively. Chapters include discussions of caches used by 4D, 4D Server, 4D Client, and several custom caching techniques you can use in your own systems.

Index Inspection

4D organizes entries within an index alphabetically and orders identical values by record number in ascending order. Look at the entries in this hypothetical index:



The entries per node in this theoretical index are set artificially low for the purposes of illustration.

Imagine that you are looking for the value “Barnes”. The common sense way to do this search quickly is to check the first and last entry on a page to see if the value might be on that page. In this way, people usually look for names in a telephone directory:

The first page starts at **Albrechston**

The first page ends at **Bailey**

Can **Barnes** be on the first page?

No:

Barnes comes after **Bailey** alphabetically

The second page starts at **Balazs**

The second page ends at **Better**

Can **Barnes** be on the second page?

Yes:

Barnes comes after **Balazs** alphabetically **and**

Barnes comes before **Better** alphabetically

The entire first page is checked with only two operations. If there are 33 entries in each page, 31 out of 32 operations are saved. If there are 63 entries on each page, 61 out of 63 operations are saved. An additional benefit of this optimization is that all query comparisons now perform at the same speed. In the past = and # comparisons could perform more quickly than ≥, ≤, > or < comparisons.