SumizAI .com Your Second Brain, Powered by Intelligence.

https://sumizai.com/blog/how-the-table-of-contents-works.html

UNDER THE HOOD

How does the table of contents file your notes?

Every personal knowledge system dies the same death: the filing stops. Not the capture — capture is easy and even fun — **the filing**. SumizAI's answer is to make filing a decision the model makes on every single note, and to constrain that decision tightly enough that it stays consistent across thousands of them.

Updated: 2026-08-14 **8** questions 4 min read

Q. What shape is the table of contents?

A tree of chapters, up to six levels deep, numbered in a single running scheme: `1.` → `1.1` → `a)` → `a.1)` → `a.1.1`. Notes are the leaves and take their number from the same sequence, so a note has an address, not just a location.

Chapters have a name, a description and a position, and the labels are computed from the position in the tree rather than stored. Move a chapter and everything underneath renumbers itself.

It is a real structure the app maintains, not a view generated on the fly from tags. That is what makes it something the model can be asked to place a note into.

Q. How does a new note find its shelf?

The model is shown the numbered chapters and asked to **pick a number**. It can also ask for a new sub-chapter beneath an existing one. A new top-level branch appears only when nothing fits anywhere.

Picking rather than naming is the design decision that makes the whole thing work. An earlier approach asked the model to name the right chapter for each note, and it produced exactly what you would expect: *Vegetables*, *growing vegetables* and *Vegetable cultivation* as three sibling branches meaning one thing.

The model reads a compressed digest of the tree — around 4 000 characters — rather than the full outline, so the cost of filing does not grow with the size of the vault.

Q. What if the model files something in the wrong place?

You move it, and nothing about that is destructive. The tree is editable by hand: rename a chapter, reorder it, nest it deeper, pull it up a level, add one, delete one.

Deleting a chapter never orphans notes. Its contents move up to the parent, because the one invariant the app maintains is that **a note is never outside the table of contents**.

If a note ever does end up unplaced — an interrupted operation, say — it gets picked up and filed the next time the table of contents is read, rather than sitting invisible until you go looking for it.

Q. What does the rebuild do that ordinary filing does not?

It sees everything at once. Filing at creation time looks at one note against the existing tree, which is the right decision with the information available — and it can never fix a structure that grew wrong before this note existed.

The rebuild reads all the notes in batches of sixty and lays the chapters out again from scratch using the main model. If a note is skipped it stays where it is, and if the model's response fails the tree is left untouched. It cannot half-happen.

There is a deliberate constraint on the result: the tree must stay **flat — at most four levels, and no umbrella chapter** that contains everything. A guard removes the umbrella if the model produces one, because a root above the root adds a click and no information.

Q. When should I run a rebuild?

After a burst of work that changed the shape of what you know, not on a schedule.

The signal is a table of contents you scroll past rather than read: chapters that made sense at twenty notes and are now holding sixty, or three branches you keep confusing with each other.

It is safe to run repeatedly, and the tree is editable afterwards, so there is no reason to treat it as a big decision.

Q. Why cap it at four levels on rebuild when the tree allows six?

Because depth is a cost paid on every read and a benefit paid once at write time.

Six levels exist so that a structure that genuinely needs them is not blocked — some subjects really do nest that far. But left to itself a model will happily produce a beautiful five-level taxonomy of forty notes, and you will never use levels four and five except to click through them.

The rebuild therefore optimises for something specific: a tree you can scan in one screen and navigate in two clicks.

Q. What does the table of contents look like on disk?

As `base.md` in the vault directory: the outline with relative links to the files in `notes/`.

That is what makes an exported vault a usable document rather than a bag of files. Open `base.md` in Obsidian, VS Code or GitHub and you have a working index — the links resolve because they are relative paths between real files.

It is generated, not authored, so it stays in step with the tree in the app rather than becoming a second thing to maintain.

Q. Does it help with anything besides browsing?

It is the map the model gets. Every question you ask travels with the vault's table of contents, which is how the app decides which notes are worth pulling in as full text.

So the structure is not decoration for humans — a well-shaped tree makes the retrieval sharper, and a tree that has drifted makes it vaguer. That is the practical argument for the occasional rebuild.

It is also the order everything is exported in. A PDF or a PowerPoint deck follows the table of contents, so improving the tree improves the document you hand to somebody else.