A. Noria In Simpler Terms

Hello, and welcome!

This section is written for anyone who wants to understand *roughly* what is going on in my thesis, without necessarily understanding all the fiddly technical bits. The running analogue I'll be using is one that I have, with various degrees of success, used to explain my work to non-technical people over the past five years. Hopefully, it'll be helpful to you as well!

Throughout the text, you'll find certain terms written in *italics*. Those are technical terms that are actually used in the thesis proper, and they will arm you with some signposts to connect what you are reading here with the thesis content.

The Library. Imagine a huge library that holds all the information for a website or app of your choice. This could be Facebook, Twitter, Instagram, TikTok, Reddit, you name it. It holds information about every user, every post, every like, every upvote, every comment, every picture, and every video. Every time you open said website or app, some representative has to go to the library to collect all the information relevant to whatever you are trying to view. If you are looking at your Facebook timeline, the representative has to figure out who your friends are, what they have posted recently, what comments there are on those posts, etc. Similarly, if you are looking at a Reddit post, the representative must gather the original post, but must also run around to find all the comments on that post, upvotes on those comments, etc. The representative may also need to collect additional information such as your name, and whether you have any new notifications or messages. It's an exhausting affair. The library is a *database*.

The Librarian. The representatives are not allowed to directly browse the library. Instead, the library has a librarian who knows the library really well, and who answers questions about the library's contents. When a representative wants to inquire about something, they ask the diligent librarian, who then scours the library to come up with the answer to the representative's *query*. This particular librarian is rather forgetful, and by the time the next representative steps up, they've already forgotten all about the previous interactions. But the representatives have grown to find this endearing. The librarian is a *database engine*, a word which is often used interchangeably with "database".

Answering Questions. As you might imagine, some questions are easy to answer, whereas others may take a very long time for the librarian to figure out the answer to. If someone asks "what is Jon's email address?", the librarian only has to look in the user directory (a *table*) for the entry for Jon, and all the information is right there. That is, of course, assuming that there is such a thing as a user directory which has information ordered by the user's name (an *index* whose *key* is the user's name). On the other hand, if someone asks "how many people have liked this post of Jon's?", the librarian has a bigger task in front of them. Even if there is a directory that lists likes by which post the like was for, the librarian still has to count how many there are, which could (hopefully) be a lot. Questions can even get so complicated that the librarian has to look through every single like in the library to get the answer! For example, imagine a representative asks "what is the post with the most likes?". To answer that question, the librarian must know how many likes **every** post has, which means they have to count the number of likes on every post. Ouch.

Writing Things Down. If we think back to the fact that these representatives are ultimately trying to bring content to users who are sitting there waiting for the page to load, it quickly becomes obvious that we need the librarian to answer questions **very** quickly. Now, this librarian is very speedy indeed, but if the questions get sufficiently complex, the answers still take time to find. So, one day, the librarian has an idea. They realize that a lot of representatives are asking the same few questions (the distribution of questions is *skewed*). For some reason, a lot of representatives want to know what Robert and Frans are up to (very few bother checking what posts Jon has made recently), and the librarian figures that if they can save themselves the repeated trips, it'll save quite a bit of time. So, the librarian decides to start **writing down** the answers they give out to representatives in a little notebook. When a representative asks a question, the librarian first checks their list of questions they've already found the answer to, and if the answer is there, they don't have to leave their comfy chair! The librarian has decided to *materialize*, or *cache*, query results.

Erasing Things. Sadly, the librarian's plan has a flaw. Over time, their lists grows so large that they're spending most of their time just reading through their list to look for whether they've heard a particular question before! The list is filled with questions no-one has asked in ages, which makes it hard to find the questions that are asked a lot. Worse yet, because of their inconvenient forgetfulness, the librarian doesn't actually remember which questions are asked frequently, and which are not. So, the librarian decides to simply erase a bunch of entries from their list at random. They figure that questions that are asked a lot will be asked again soon anyway, and then end up on the list again, whereas questions that aren't being asked much, well, are just going to stay off the list. This is *eviction*, and specifically *randomized* eviction. Other eviction strategies exist (like "least recently used"), but the work in this thesis uses randomized eviction.

Productive Humans. Unfortunately for the librarian, the library is ever-changing. Every day, representatives bring in piles and piles of new records that users have produced. Likes, comments, photos, and more are coming by the boatload. Worse yet, the representatives expect that those records immediately start showing up in the answers to the questions that other representatives ask! If Jon posts something, they expect his mother to then see that post almost immediately. In the past, this wasn't too much of a problem — true, the librarian had to file away the records that came in, which took some time, but at least by the time they were looking for answers for the next representative, they would also come across those new records and take them into account. But now that the librarian is using their notebook, they're often not even looking at the records, and so risk giving inaccurate answers! In other words, the answers in the notebook grow *stale*, and are no longer *consistent* with the data in the library.

Work-Work Balance. After thinking about their problem late at night when they have some downtime, the librarian realizes that they need to be updating, or *maintaining* their notebook when they are filing away new records that arrive. They figure that while this will make filing take longer, so many more representatives ask questions than bring new records, that on balance the notebook should still allow them to serve more representatives per day overall. The library *workload* is *read-heavy*. The librarian still has to decide how to split their time between servicing representatives that bring new records and those that want to ask questions, but as long as both lines are shrinking, all is good.

Throwing Away the Notebook. The first thing the librarian thinks of is to simply throw away (*invalidate*) the notebook any time a representative brings new files. This works, but the librarian quickly discovers that this doesn't save much time over not keeping a notebook at all. Since there is always a steady stream of new files, the notebook barely gets a few entries in it before it has to be thrown away! The librarian thinks there might be a way to only erase entries in the notebook that are related to the newly filed records, but quickly eliminates that option — the new records that come in frequently affect the questions that most people have (new likes for Robert's newest post alongside questions for how many likes Robert's newest post has). If the solution was to throw those entries away, the librarian would still spend all their time counting how many likes Robert's newest post has.

Updating the Notebook. While erasing a notebook entry one day, the librarian catches themselves thinking that what they're doing makes little sense. A representative brought in a single new record that was a like on Frans' latest post, and the librarian's eraser currently hovers over an entry that says that the current number of likes on that very same post is 42,006. The librarian erases the number, grins, and before moving on to erasing the next entry puts down 42,007. This is genius! The next time someone asks for the number of likes on that post, there's no need to go count all those likes from scratch again — the entry in the notebook will still be there **and**

it will be correct. The librarian maintained the notebook *incrementally*, and thereby saved having to do a bunch of redundant work later.

More. More! More!! Now that the librarian has discovered this little trick, they start looking for other entries that can be updated in the same way. Unfortunately, while the procedure is simple for some answers in the notebook, it is very tricky for others. It's all well and good to add two numbers, but if Jon, who wasn't following Robert before, starts following him, all of Robert's past posts now have to be placed at the correct position in the list given as an answer to "what posts have recently been made by people that Jon follows?". The librarian takes a break and ponders if there's a good way to solve this problem.

A Hierarchy of Notebooks. The next day, the librarian comes in with a plan, and pulls out a large sheet of drawing paper. Each time a question comes in, the librarian maps out a flow-chart for how they figured out the answer to that question. What directories they had to browse through, in what order, and how the files from those directories were combined. Then, the librarian keeps a separate notebook for each step in that flow chart. Count the number of likes in this directory under the entry for a particular post? Great, write that number down in one notebook. Look for all the people that both user A and user B follow? Great, write down which were in both in another notebook. Then, do this all the way down to the final answer for each question. If two questions require similar steps, the librarian re-uses the overlapping parts of the flow chart, and uses the same notebooks for the same steps. The librarian is mapping out the *dataflow* of the questions — how the entries in the notebook relate to the data that's stored in the library.

Using the Flow-Chart. The librarian's insight only becomes apparent once the next batch of new records are brought in by a representative. Now, instead of looking through all the notebook entries, the librarian looks at where each new record would be filed. Then, the librarian consults the giant flow-chart, and looks for the step in the chart that indicates to look something up in that same place. If a new like comes in, then the librarian looks in the flow-chart for a step that reads "go to the directory that holds likes". Then, the librarian follows the edges of the flowchart from that step. For each step, the librarian finds the entry in that step's notebook that matches the new record, updates that entry to include the new record, and then moves on to the next step. If a step is followed by multiple parallel steps, the librarian does all of them, one after the other. By the time there are no more parts of the flow-chart to follow, the librarian has updated all the notebook entries that can possibly depend on the new record. And crucially, without looking at anything unnecessarily! This is *incremental view maintenance using dataflow*, and is what *Noria* provides.

Common Knowledge. The librarian is now pretty happy — while it takes a little while to update the notebooks to reflect new records that come in, it's not too bad, and a large majority of all the questions that representatives come in with already have up-to-date answers in the notebooks. Life is pretty good. But every now and again, the librarian still has to answer questions whose answer does not appear in a notebook. This is especially frustrating in cases where the librarian is pretty sure that they found the answer some time in the past, but has since erased the relevant entry to save space in the notebook. It feels like there should be a way to cobble the answer together from related tidbits in other notebooks that may still hold parts of the answer, rather than having to go all the way back to the library shelves and do all that tedious manual counting. If Robert's post's like count is still in **some** notebook, then the librarian shouldn't have to count the likes again just because there isn't an entry for the specific question the representative asked.

Suspiciously Similar Flow-Charts. After going through the steps of answering some of these questions where it feels like at least part of the answer lies in a notebook somewhere, the librarian starts to notice a pattern. When drawing out the flow-chart steps for the new question, there's nearly always overlap with steps from some other questions. And while writing into the notebooks for those shared steps, there's almost always an entry with the exact same answer already present. The librarian realizes eventually that this is not actually so surprising — if two questions both ultimately require that the librarian count the number of likes for one of Robert's post, then they will both share the prior steps related to that question in the flow-chart. And the result will in both cases end up in the same entry in the relevant notebook — the one for that post.

Flow-Charts in Reverse. Following this observation, the librarian decides to try something. The next time a question comes in for which the notebooks do not have an answer, the librarian maps out the flow-chart for answering that question as usual. But instead of then following the flow-chart from the top ("start by opening the likes directory"), the librarian follows the flow-chart **in reverse**. The librarian first looks in the answer notebook at the end of the question's flow-chart, and if the answer isn't written down there, then goes up to the notebook for the second-to-last step of the flow-chart. If the relevant entry is written down there, then the librarian can just take what's written there, do the last flow-chart step, and then have the answer for the representative's new question! If that notebook also has no relevant information, the librarian continues "up" the flow-chart until they either find an entry, or the flow-chart says to look in one of the data directories, where the relevant data is guaranteed to reside. What the librarian is doing is an *upquery* — a reverse lookup in the dataflow for information that isn't **quite** refined enough to answer the question, but is better than having to consult the entire data directory. Upqueries are particularly attractive because they allow the librarian to keep only a single flow-chart, rather than keep multiple "what to do if this notebook doesn't have the information?" flow-charts.

This Thesis. This is as far as this analogue will go. It's not perfect, but it should give you sufficient working knowledge of the problem area that this thesis tackles. In particular, the contributions of this thesis is the notion of "upqueries", as exemplified by the last paragraph. All the techniques from the preceding paragraphs already exist in past related work.

Partial State. You may wonder about the lack of the word "upquery" in the thesis title, and what "partial state" means. This is one place where the library analogue starts to break down. An approximate explanation is that partial state is what enables the librarian to erase entries from notebooks, and upqueries are an important part of how to make having such erased entries practical. And while the ability to erase things may seem trivial, it turns out that the librarian's flow-chart approach gets tricky when information may be missing from notebooks part-way through. Especially since most databases ("libraries") have multiple concurrently executing "librarians", which all share notebooks and need to ensure that they do not step on each other's toes or overwrite each other's work!

The Real World. To understand why this work **matters**, we need to tie back all we've explored above to real-world concepts. The notebooks represent computer memory, which in practice is both limited and expensive, so you can't just go around writing everything down. This is why it is so important to keep the notebooks small. The librarian's work must be done by a real computer somewhere, and every second of work costs money. If the librarian does half as much work, that's a bill, and often a significant one, cut in half somewhere! Overall, you can think of the strategies we've explored in one of two ways. Either, think of it as letting you do more with limited resources — the are only so many notebooks, and the librarian only works eight hours a day. Or, think of it is as letting you do the same with fewer resources — you can now serve the same number of representatives while using fewer notebooks and letting the librarian leave early.

I hope that was helpful. Thank you for reading!