Topics

- How to look for a job
- Database systems

Content

How to look for a job

- Right now it's tough, last week there were ~30k employees laid off from various tech companies
- Want to touch on how one goes about finding opportunities, especially in a tough environment.
- PEY
    - What's the current situation? How many jobs are still posted? Are you getting interviews?
    - With PEY and other job applications, it's all a numbers game. Your goal is to get a job, and there are a series of steps which lead to that. In marketing, that's called a "conversion funnel". With every step, you have some percentage of success, and to have an expectation of landing that job (reaching your final goal), you basically have to multiply all the percentages to figure out how many jobs you need to apply to on top
- Conversion Funnel
    - For example, if the funnel has the following steps
        - Resume review (50%)
        - Pre screen (30%)
        - Interview (10%)
    - Then your probability of success is 0.5*0.3*0.1 = 0.015 which is ~1.5% chance of getting a job. On average, you need to apply to 67 jobs (67 * 0.015 = 1) to expect 1 offer.
    - Therefore then, your job is two-fold
        - Apply to as many jobs as you possibly can, so you maximize the chances of hitting an offer
        - Optimize the funnel and step to step conversion rate and get it as high as you possibly can.
    - First part is easy, just log in every day and submit your resume to every job out there
    - Second part is much harder, typically breaks down to:
        - How good is your resume? Is it scannable? How unique is it from your friends and peers? Would you hire yourself if you see the resume?
        - How good are your fundamentals? Are you able to pretty easily crack a reasonable portion of leetcode medium problems?

        - How's your interview performance? Are you so nervous that you can't speak properly and think correctly? Are you able to communicate well and apply what you know to the interview?
    - All of the above takes time to perfect
- Resume
    - Keep it to one page, if I can write all of my work experience on one page, you can too :)
    - Make it scannable, think about myself for example. When i was reviewing resumes for Uber, I had a stack of 200 resumes to do in about 1 hour, which breaks down to about 18s per resume.
    - Set an alarm for 10-20s, see how much info you can gather in about that amount of time, and use it as a benchmark. Your job is to not stuff as much info into the resume as you can, but to take as much of it out as possible and only focus on the important things.
    - A good way to do it is to write a 3 page resume and cut it ruthlessly down to 1 page.
- Prescreen
    - This can take a lot of form, all the way from a project which might take a few hours to do or a coding assessment which can take anywhere from 30 mins to 1 hour.
    - **This is where you shine**. Your resume has been selected, and you are far from the pressures of a live interview. This is where the core of your preparation will pay off.
    - If it's a coding assessment, don't jump into it right away. Do some leetcode problems to warm up, make sure you can do 100% of leetcode easys and 40-50% of leetcode mediums w/o any hints.
    - If it's a project, don't just bang it out and submit, ask a friend to help review your code and make sure you put your best foot forward (however don't get someone else to write it for you, you will be caught).
    - I was in the unfortunate position of catching someone cheating the previous term working at Uber on a coding assessment. We ended up reporting them to the university. There is a fine line getting advice/help on these things and straight up having someone do it for you. We are looking for consistency throughout the process, and if someone does not perform according to one part of the assessment, we will take a closer look.
- Interviews
    - When you go to your first interview, **it should not be your first interview**. You should've gotten at least 10 practice rounds before you do your first actual live interview.
    - How do you get those practice rounds in? How many people here are looking for a job? Okay after class, if you are looking for a job, find 10 other people and set aside a 2.5 hour stretch with each of them. For the first hour, you interview the other person, for the second hour, the other person interviews you. For the last

30 mins, compare notes, give each other feedback. There, now you have 10 actual live interview practice under your belt. You are much better prepared :)
- It's super valuable and eye opening to sit on the other side of the interview table. Pay attention to the feeling you get as the other person is hard at work, what are you thinking? Pay attention to the other person, how are you evaluating them? What do you like/not like about various aspects?

How to treat lectures

- Note, It's super important for you to go to tutorials, that's where most of the tactical materials will be covered for this course. For example, previous tutorial we talked about REST apis and next one we will talk about more specifically Neo4j databases
- The goal of the lecture is to tell you more generally the spirit of what's going on and give you context around what's important to understand
- Think of yourself as a tree of knowledge, and the trunk is you foundational knowledge, everything that you are learning here at the Unviersity. The branches are the context and every time you gain a piece of new information, you will use your foundation, the context and place it on one of the branches. As most knowledge does not exist in isolation, it's my job to introduce you to the context so you can solve bigger and more complex problems in the future.

Storage systems

- One of my favorite treatments of this topic is building it from bottom up, so we'll do some of that today
- Basic database
    - Consider the following simple "database":

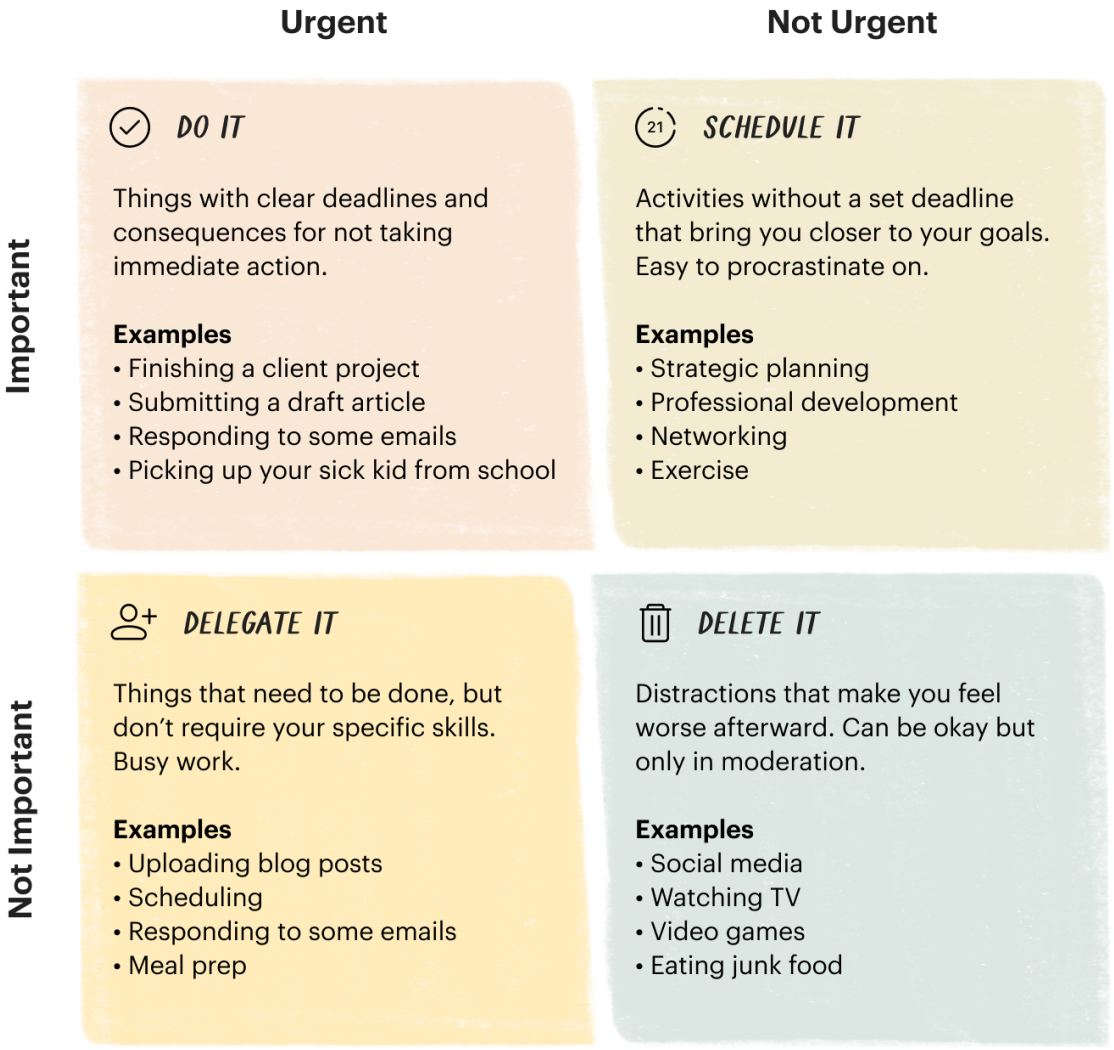
```
1 #!/bin/bash
2
3 db_set () {
4     echo "$1,$2" >> database
5 }
6
7 db_get () {
8     grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
9 }
~
```

    - I believe everyone here has taken command lines course from before? Can someone tell me what this does?
    - What are the performance characteristics of this database? Does it do anything particularly well?
    - You can claim that this is a complete database, and everything else (including both from SQL and nosql world) is just an optimization for different purposes
    - How might you go about and optimizing this database?

- Let's resolve the issues with this DB's $O(n)$ time for retrieval
    - Again, I just want to have a central theme running through this course, that is everything is a **trade off**, as you will see throughout this discussion, nothing comes for free. Typically every time you improve something, something else becomes slightly worse. Your job as an engineer is to make sure the positives outweigh the negatives. If someone promises you free lunch, you should probably look harder.
    - Let's build a simple index!
    - What's a simple index we can have that resolves the linear read problem?
    - How about, an in memory dictionary structure that stores the key and latest offset in the file.
    - Now we have O(1) reads and writes, what's the trade off?
        - All keys must fit in RAM
        - Upon crash, index must be reconstructed, which can take a long time for large DBs
        - **Increased complexity** - this is key, a lot of software engineering is managing complexity. Typically to resolve a drawback for a particular solution, you have to increase complexity. More structures, more auxiliary systems, more points of failure.
        - Risk of the db on disk goes out of sync with the index
- More will follow in future lectures
- SQL
    - Relational model: data is organized into relations (tables) and each relation is an unordered tuple (rows)
    - SQL is declarative, likely the only declarative language that most programmers have exposure to (other declarative language include prolog)
    - You are describing the results of what you want to achieve from the database, and it gets the data to you, and typically you are not exposed to how the data actually gets to you.
- NoSQL
    - Originally intended as a Twitter hashtag, but evolved to encompass a bunch of technologies and goals:
        - Greater scalability than relational databases can easily achieve, incl very large datasets and high write throughput (ex. Key-value stores)
        - Preference for free and open source vs commercial databases
        - Specialized queries not well supported by relational (ex. graph)
        - Frustration with relational schemas, and a desire to have more dynamic and/or expressive data models (ex. document)
- Mongo
    - Schemaless? You can't really have no schema and just insert data in arbitrary ways and expect the client to make sense of the data. More accurately can be described as "Schema-on-read".
    - Traditionally you have schema on write, where when you are writing data to the database, database enforces a schema upon the data being written

    - One way you can think about schema on write is like static type checking in a language like Java or Haskell, and schema on read like dynamic programming and duck typing.
    - For instance, if you have a “name” field and wishes to split into first_name and last_name. In document databases, you would handle it in application code with a if/else statement
    - However in SQL databases you would do a schema migration, aka write some SQL that creates the new columns and migrates the old data over to teh new data.
    - Document database are great if you are consuming data from a third party and it’s not practical to create or enforce schemas, and you are happy dealing with the differences when the data is consumed.
    - Or if you have data that’s naturally document like with minimal interactions with other data sources outside of itself. For example if you are building a resume hosting site.
- Neo4j
    - Graph database, this is another type of interesting nonrelational DB you may consider.
    - Many to many relations are fairly commonplace in the world, and relational DB can handle simple forms of many to many relations.
    - However, once it gets complex, or you have multiple hops between data and other data.
    - If you imagine you are trying to create Facebook, with the near infinite ways where two humans can be connected to each other. You may want to ask it a simple question, how many degrees (hops) do i have to do to go between me and another person. It’s fairly easy to express that in Neo4j but harder to do in SQL. Typically you would have to resort to recursive queries and it’s quite long. You will explore this idea in assignment 1.
- Dependency Injection
    - Very simply, instead of taking dependencies for granted, you are injecting the dependency into a class at runtime
    - An example: instead of hard coding in a database connection, you instead would pass it through in the constructor of the class. Thereby allowing easier testing and you are not locked into a certain class in the beginning.
    - As an example, imagine you are writing a class to connect to a database. Instead of hard coding the database access class in your application, you might want to take in as a parameter. When you are writing automated test suites, you can substitute a mock object and be able to test things in isolation.
- Prioritization:
    - How does one prioritize? Any ideas?
    - Urgency VS importance
    - One modification I would give to urgent vs important is the “urgent but not important”, oftentimes it’s not important “right now”. However, if you wait long enough, they will become important.

- In other cases, namely certain bugs you may get from time to time. Sometimes the bugs are fairly ephemeral and if you wait long enough, they will become not urgent and not important.



- Estimation:
  - One important skill every software engineer must develop is a good sense of how long certain things take.
  - However, it's a tough skill to learn and develop. Typically the knowledge you need is split into a few categories: Known Knowns, Known Unknowns, Unknow Unknowns.
  - Most projects of an interesting size are filled with the two types of unknowns. If you have not encountered them yet, it most likely means you haven't thought deeply about it.

    - Typically when you are a junior engineer, you are not responsible for estimating the project. Often you will start by estimating the smaller features/tickets that you get, and it's a good opportunity to start thinking about how you might go about doing it.
    - There are various models for doing these estimations. I find the most reliable to be very simple, **when in doubt or don't know where to start, break down the project into smaller pieces.**
    - For example, you might be tasked with estimating an ambiguous problem, or looking into an area you are not super familiar with. You might start by thinking through carefully step by step
        - What knowledge you need to know
        - What experiment(s) you might need to run to understand the problem better
        - Rough ballpark of how much effort this will be (weeks/months) after doing some investigation
        - If the project/task is more than 1 week, you need to start breaking it down, gradually into the next level of granularity. Create a project schedule in whatever tool you like, and then specify what needs to be done week over week until delivery, who is going to do them etc
        - Finally, work with the project team to break it down further, ideally into 1-2 days granularity and then create tasks in task tracking system
-
- REST APIs
    - Standard way of communicating between client and server, over HTTP, is stateless (primary factors)
    - Being perfectly honest, it's not a protocol like SOAP or HTTP is, but a set of guidelines that most people understand more or less intuitively
    - Most of what you see/hear about REST are more to do with HTTP rather than REST itself. For example, it uses HTTP methods (GET/POST/PUT/DELETE).
    - Resources to reference:
        - https://en.wikipedia.org/wiki/Representational_state_transfer
        - https://developer.mozilla.org/en-US/docs/Glossary/REST
- Designing good REST APIs
    - Use HTTP methods correctly. GET should not change state, aka it can be retried as many times as needed to get the results back.
    - POST is meant to modify resources. Hence if you retry a POST request, a browser would typically warn you that it may modify things on the server
    - PUT/DELETE is seldomly used, but PUT is meant to create new records and DELETE is meant to signal a record should be deleted
    - Make sure URLs are semantic. For example, typically follows this format: https://api.example.com/<version>/info-hierarchy
        - https://restfulapi.net/resource-naming/ good article on this part