Author: Hannah C. Tang (hctang@cs)
Last Update: May 2024

Introduction

This doc outlines the running **Payroll/Regist example** we've been using all quarter, as well as how it should be implemented in the various **NoSQL** options we'll be discussing. We will add a new table to the running example, ParkingTickets.

As a reminder, this is the schema and we've included some example values. Note that Frances doesn't have a car; Magda has two, and nobody employed at UW owns the Aston Martin.

Payroll

UserID	**Name**	**Job**	**ParkingPermit**	**SalaryHistory**
123	Leslie	TA	C15	50k
345	Frances	TA	NULL	60k, 50k
567	Magda	Prof	E18	120k, 110k, 100k
789	Quinn	Prof	NULL	100k

Regist

UserID	**Car**	**LicensePlate**
123	Charger	123 AAA
567	Civic	MMM 1234
567	Ferrari	MMM 5678
789	Picklemobile	PIK 1024
007	Aston Martin	XYZ 0007

ParkingTickets

LicensePlate	**ParkingLot**	**Date**	**Amount**
MMM 1234	C15	2022-11-20	$10
MMM 1234	E01	2022-11-21	$15
PIK 1024	E18	2022-11-22	$10
XYZ 0007	C19	2022-11-01	$10

MMM 1234	E01	2022-11-22	$20

Application

This is a parking enforcement app which supports the following methods:

1. [infrequent] Listing the permitted parkling lot and per-car tickets incurred by each user
    - method sig is `uid -> [ {car1, [{tix1}, {tix2}]}, {car2, []} ]`
2. [frequent] Counting how many tickets a license plate has ever had
    - method sig is `plate -> int` or it can be `plate -> [ {tix1}, {tix2} ]`
    - We use the output of this method to determine the citation amount.
3. [multiple times / sec] Determining whether a plate is allowed to be a specific lot
    - method sig is `(plate, lot) -> true/false` or it can be `plate -> lot`

Because NoSQL design is so intimately tied to its use-cases, there are three decisions which will need to be made for each system:

- **Method 3**: should we store the license plate and permitted lot as the key, with the value being true and the expectation that values which don't exist should default to false?
- **Method 2**: should we store a count of the tickets, or the actual tickets themselves? Note that the latter introduces the possibility of data anomalies (which may be deemed acceptable)
- How should we handle **cars without owners** (the Aston Martin) or **employees without cars** (Frances)? Is data loss acceptable or not?

Key/Value Store

Method 3 should just have the key be `(plate, lot)`; we can represent false implicitly by not having a row. So this keytype would have 3 k/v pairs (there are only 2 parking permits, but one of them is allowed to have their cars, plural, share a single permit).

Key	Value
123 AAA:C15	true
MMM 1234:E18	true
MMM 5678:E18	true

You could alternatively have the key and value be `plate->lot`, but that pushes more logic into the application (to verify that the parked lot equals the permitted lot):

Key	Value
123 AAA	C15

MMM 1234	E18
MMM 5678	E18

Method 2 is more elegantly implemented with a count rather than a list of tickets (less data to transfer/store):

Key	Value
MMM 1234	3
PIK 1024	1
XYZ 0007	1

But it's possible to store a list of entire tickets, too. Personally, I prefer to keep it as a count and use the data stored for method 1 as the "canonical copy" of all ticket information.

Method 1 depends on how you represent orphaned tickets (ie, tickets incurred by cars not owned by UW employees); if we want to keep this entirely within a K/V store, then we'd have to introduce a (possibly very large) key to hold all the orphans:

Key	Value
123	[C15]
345	[NOPERMIT]
567	[E01, {magda's civic, [its 3 tickets]]}, {magda's ferrari, []}]
789	[NOPERMIT, {car:picklemobile, []}]
UNOWNED	[NOPERMIT, {car:aston martin, [{parkinglot: C19, date: 2022-11-01, amount:$10}]}]

Personally, I'd use a different DB; I'd likely choose a RDMS like SQLServer. This allows me to store the many-to-many data without introducing anomalies; there would be very little performance penalty since method 1 is invoked so rarely, and tickets are issued merely "frequently".

Document Store

Note that we artificially constrain students to designing a *single document*, which necessarily means that there will be data loss: if Payroll is contained in Regist, we'll lose car-less Frances; if Regist is contained in Payroll, we'll lose owner-less Aston Martin. We could hack around this by having a special key to contain "orphans" (eg, `"NO_OWNER" -> {Frances}`), but in the real

world this is probably best solved by having multiple top-level datasets: Payroll, Regist, and a pre-computed join of the two tables. The rest of this section assumes we implement the application using a single document.

Since the two most common operations are keyed off of license plate, we select Regist as the top-level dataset and license plate as its (hopefully indexed) key:

```
[
  {“plate”: “123 AAA”,
    “car”: “Charger”,
    “userid”: 123,
    “payroll”: {“name”: “Leslie”, “job”: “TA”, “parkingpermit”: “C15”},
    “tickets”: []
  },
  {“plate”: “MMM 1234”,
    “car”: “Civic”,
    “userid”: 567,
    “payroll”: {“name”: “Magda”, “job”: “Prof”, “parkingpermit”: “E18”},
    “tickets”: [
      {“parkinglot”: “C15”, “date”: “2022-11-20”, “amount”: 10},
      {“parkinglot”: “E01”, “date”: “2022-11-21”, “amount”: 15},
      {“parkinglot”: “E01”, “date”: “2022-11-22”, “amount”: 20}
    ]
  },
  {“plate”: “MMM 5678”,
    “car”: “Ferrari”,
    “userid”: 567,
    “payroll”: {“name”: “Magda”, “job”: “Prof”, “parkingpermit”: “E18”},
    “tickets”: []
  },
  {“plate”: “PIK 1024”,
    “car”: “Picklemobile”,
    “userid”: 789,
    “tickets”: [
      {“parkinglot”: “E18”, “date”: “2022-11-22”, “amount”: 10}
    ]
  },
  {“plate”: “XYZ 0007”,
    “car”: “Aston Martin”,
    “tickets”: [
      {“parkinglot”: “C19”, “date”: “2022-11-01”, “amount”: 10}
    ]
  },
  {“plate”: “CAR LESS”,
    “payroll”: [{“userid”: 345, “name”: “Frances”, “job”: “TA”}]
  }
]
```

Method 3 is implemented as a `plate` lookup on the above dataset.

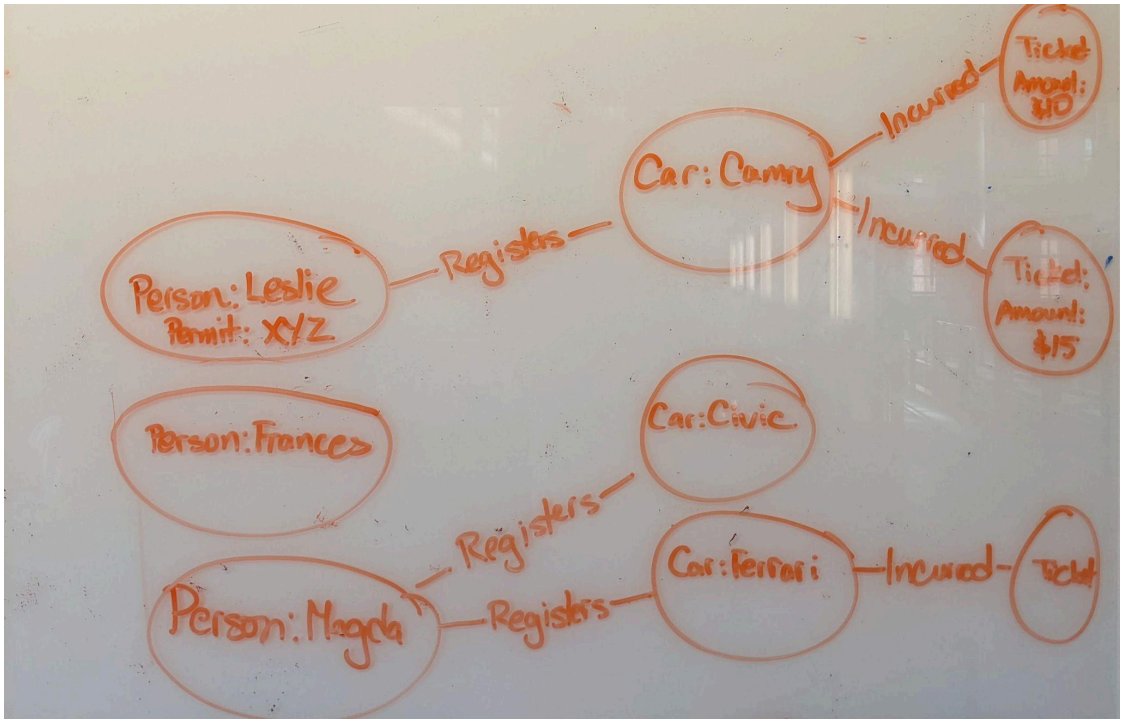
If you're willing to create multiple datasets, this app could benefit from a pre-computed `(plate, lot)->boolean` dataset; users change their license plate or parking permits maybe once a month (so the likelihood of anomalies are low), and the smaller dataset + slightly less computation could speed up queries.

Method 2 is also a `plate` lookup. Because it is less frequently called *and also* because its underlying data changes more frequently than method 3's, the argument for precomputing is less compelling.

Method 1 should be a full scan of the entire document to build a userid-keyed dictionary, since the document is currently keyed on the license plate.

Graph

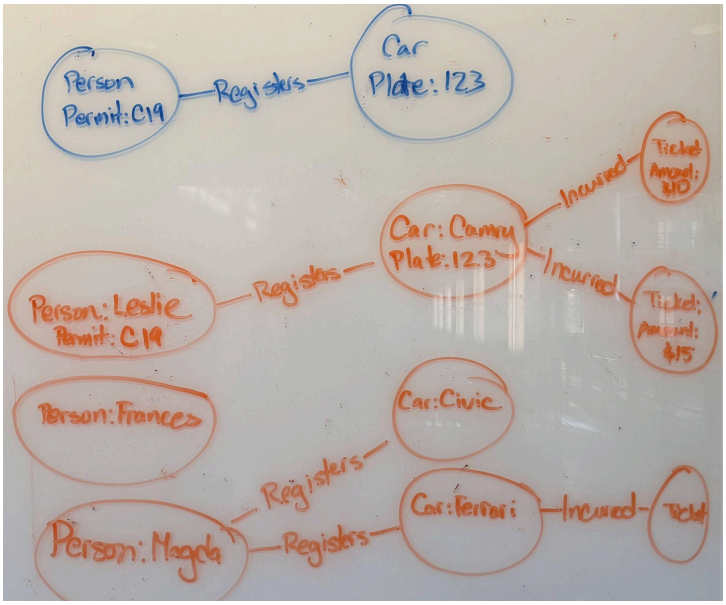
We will have 3 types of nodes: `EmployedPerson`, `RegisteredCar`, and `ParkingTicket`. They will be connected by 2 types of edges: `Registers` (linking `EmployedPerson`'s with `RegisteredCar`'s) and `Incurs` (linking `RegisteredCar`'s with `ParkingTicket`'s). An example graph might look like this:



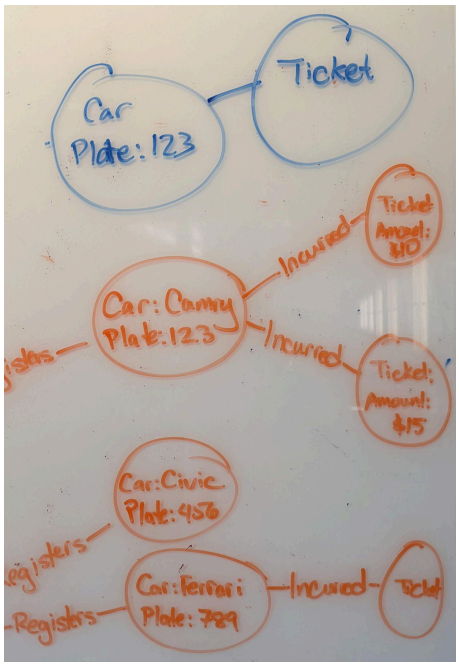
All three methods would be implemented by specifying a graph fragment to match the above graph against.

Method 3 is implemented as a match on the specified `plate` and `lot`. For example, if we wanted to find whether `LicensePlate=123` is allowed to park in `Lot=C19`, I would search for a graph fragment where a `Person` node would have `Permit=C19` and is connected via a

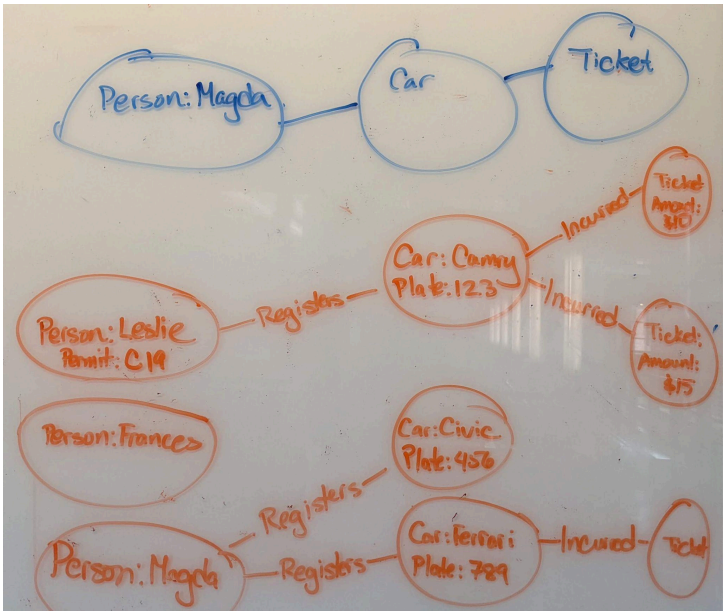
`Registers` edge to a `RegisteredCar` node with `LicensePlate=123`. Using the above example graph, this fragment would match Leslie and her Camry but not Magda and her Civic



Method 2 is implemented similarly. If the query language supports it, I would query for the number of matches rather than getting the actual matches and then performing the count manually. Note how this example fragment will match *two* tickets: the Camry's $10 ticket and its $15 ticket.



Method 3 likely requires two different graph fragments. The first returns cars that have incurred tickets (in this example, it would return Magda's Ferrari):



and a second fragment would query for cars that don't have tickets.

Wide Column

There are two access patterns: one for method 3 (to optimize lookup times) and one for methods 1 and 2 (since performance is not as critical), which indicates a need for two column families.

There are two possible schemas that would work for the rowkey.

Solution 1: Inspired by Key/Value

Because the Payroll/Registry/ParkingTicket application is so straightforward, it's possible to reuse the Key/Value schema. We would need to add a prefix to each rowkey, indicating the key's "type":

- (plate, lot) *(type=l)* *(mnemonic is "parking Lot")*
- plate *(type=c)* *(mnemonic is "Count")*
- userid *(type=u)* *(mnemonic is "User")*

Since there is exactly one value for each rowkey, we can hardcode any column name we want. For simplicity, we'll reuse the rowkey prefix as the column name.

	colfam=Lookups		**colfam=UserInfo**
	col="l"	**col="c"**	**col="u"**
l#123 AAA:C15	true		
l#MMM 1234:E18	true		

l#MMM 5678:E18	true		
c#MMM 1234		3	
c#PIK 1024		1	
c#XYZ 0007		1	
u#123			{ Leslie's payroll info, her parking permit, and empty list of tickets }
u#345			{ Frances' payroll info and empty list of tickets }
u#567			{ Magda's payroll info, her parking permit, and 3 tickets}
u#789			{ Quinn's payroll info and single ticket }

Note that there are **no rows** which have more than one column; wide column NoSQL databases handle sparse data elegantly.

Solution 2: Native WideCol

A more "native" wide column schema would notice that method 2 and method 3 are logically keyed off the same data: the license plate; the difference between the two methods' signature was an optimization for when we *also* knew the parking lot. Since wide column databases give us more tools to optimize our schema, we can reduce the key "types" to:

- plate *(type=p)* *(mnemonic is "Plate")*
- userid *(type=u)* *(mnemonic is "User")*

We consolidate the `(plate, lot)->boolean` and `plate->count` into a single row by using the parking lot *as the column name* (as with key/value, the lack of a cell is an implicit `false`).

	colfam=Lookups			**colfam=UserInfo**
	col="C15"	**col="E18"**	**col="c"**	**col="u"**
p#123 AAA	true		0	
p#MMM 1234		true	3	
p#MMM 5678		true	0	
p#PIK 1024			1	

p#XYZ 0007			1	
u#123				{ Leslie's payroll info, her parking permit, and empty list of tickets }
u#345				{ Frances' payroll info and empty list of tickets }
u#567				{ Magda's payroll info, her parking permit, and 3 tickets}
u#789				{ Quinn's payroll info and single ticket }