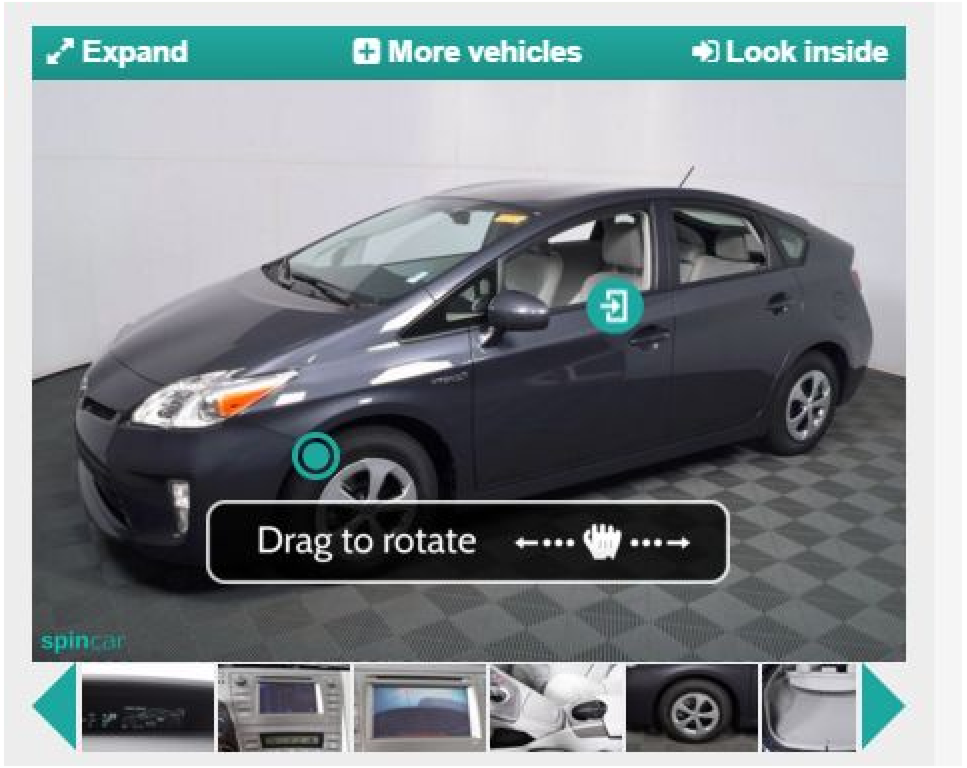
Unlimited Scalability in the Cloud
A Case Study of Migration to Amazon DynamoDB

Steve Saporta
CTO, SpinCar

Mar 19, 2016

SpinCar

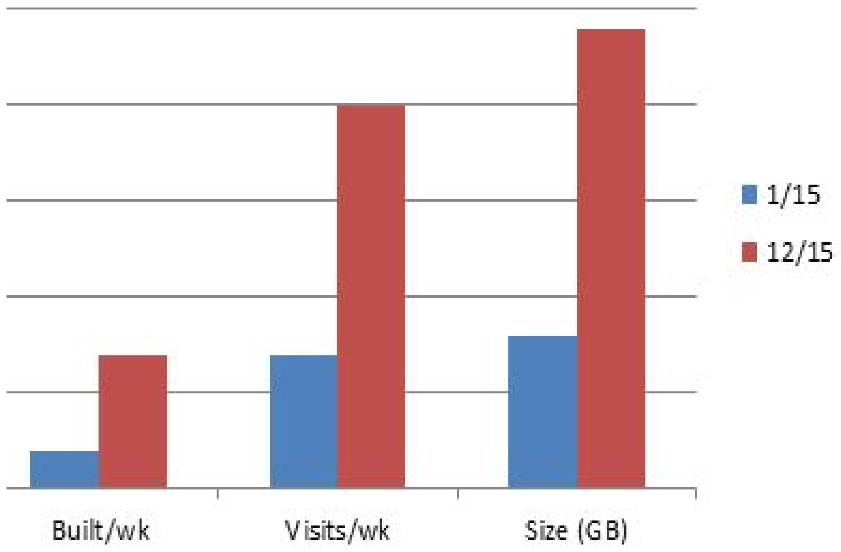


When a web-based business grows...

- More customers = more transactions
- More transactions = more database requests
- More database requests = TROUBLE

SpinCar’s growth

- WalkArounds built per week
  - Jan 2015: xxxx
  - Dec 2015: xxxx
- Page visits per week
  - Jan 2015: xxxx
  - Dec 2015: xxxx
- SQL database size
  - Jan 2015: 8 GB
  - Dec 2015: 24 GB



Headroom calculation

- “Headroom”: how much additional traffic can be handled
- We simulated increased traffic by replaying typical transactions on a large number of computers
    - “Bees with machine guns”
- When traffic reached about 5 times Dec 2015 levels, some systems failed catastrophically

Should we scale up our SQL database?

- Scaling read capacity is kind of hard
- Scaling write capacity is really hard
- Some common ways to scale SQL databases
    - Bigger servers
    - Read replicas
    - Sharding

Bigger servers

- Bigger means more expensive
  - RDS PostgreSQL Multi-AZ on-demand price
    - db.m4.large = 2 virtual CPUs, 8 GB RAM = $0.364/hr
    - db.m4.xlarge = 4 virtual CPUs, 16 GB RAM = $0.730/hr = extra $267/month

Bigger servers

- There’s only so big you can get
  - db.r3.8xlarge = 32 virtual CPUs, 244 GB RAM = $7.960/hr
  - Amazon doesn’t offer anything bigger

Read replicas

- Each read replica costs as much as a regular database server
- Synchronization isn’t trivial to set up, and it can fail
- They’re called “read replicas” because they don’t help with writes!

Sharding

- Divide the database into several smaller ones
  - e.g. customer names starting with A-I, J-R, S-Z
- Sharding can help with both reads and writes
- Application needs to direct each request to the appropriate database server
- Queries across shards (e.g. a company-wide report) can be challenging to implement

An alternative: DynamoDB

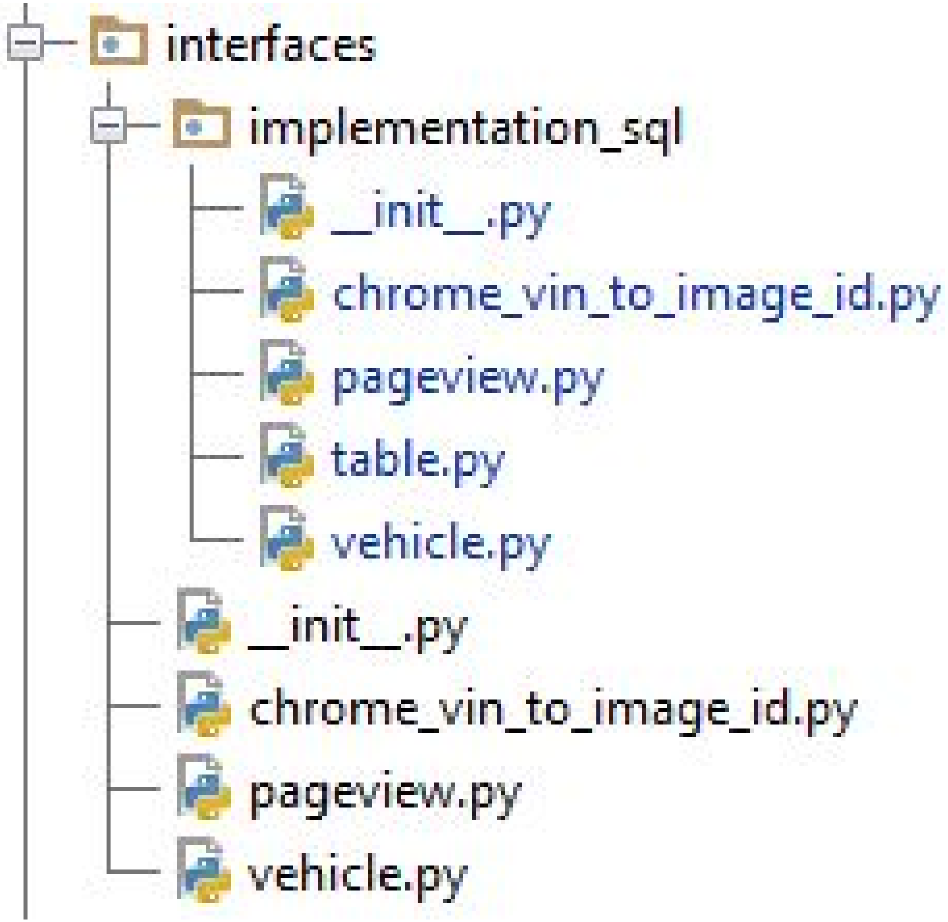
- A NoSQL database in the cloud
- Is NoSQL more scalable than SQL? Maybe.
  - http://programmers.stackexchange.com/questions/194340/why-are-nosql-databases-more-scalable-than-sql
- The real scalability gain lies in Amazon's architecture for DynamoDB
  - *There is no theoretical limit on the maximum throughput you can achieve. DynamoDB automatically divides your table across multiple partitions, where each partition is an independent parallel computation unit. DynamoDB can achieve increasingly high throughput rates by adding more partitions.* -- https://aws.amazon.com/dynamodb/faqs/#scale_anchor

Migrating from SQL to DynamoDB

- Starting point: PostgreSQL 9.x database on Amazon RDS db.m3.large Multi-AZ instance
- Goals
    - Move several high-traffic tables from the SQL database to DynamoDB
    - No downtime
    - No loss of data
- More than one way to accomplish these goals. Case study of how SpinCar did it.

Migration: Step 1

- Replace code that reads and writes the SQL tables with an interface
- Create an **implementation** of the interface that reads and writes the same SQL tables
- No change in functionality yet, just introducing a layer of abstraction



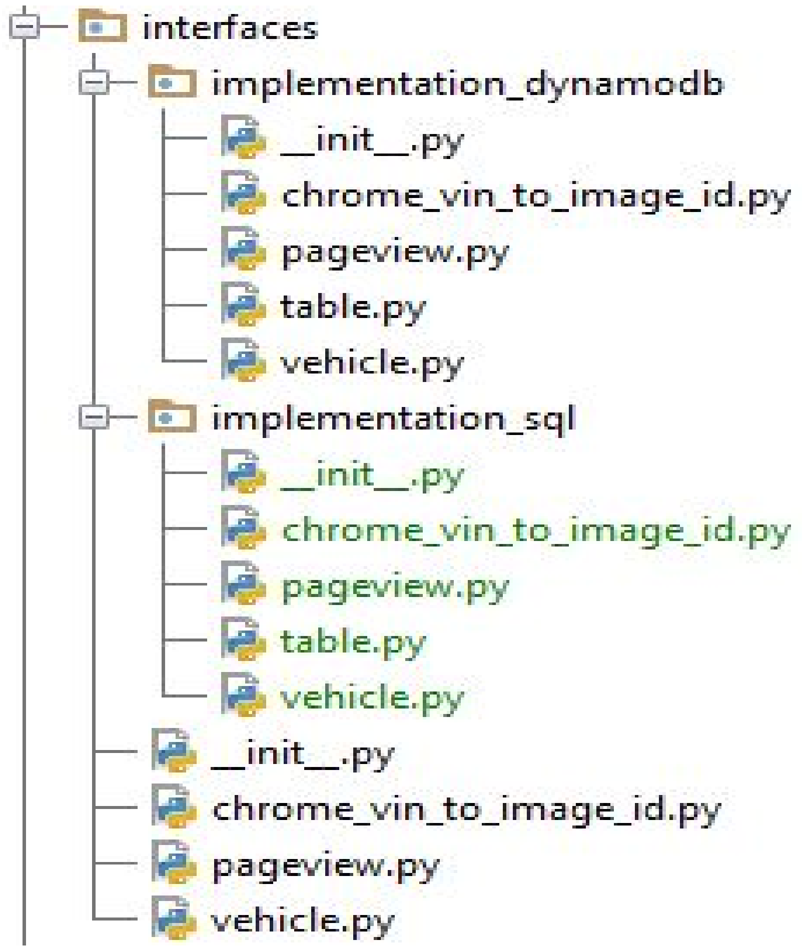
```
interfaces
    implementation_sql
        __init__.py
        chrome_vin_to_image_id.py
        pageview.py
        table.py
        vehicle.py
    __init__.py
    chrome_vin_to_image_id.py
    pageview.py
    vehicle.py
```

```python
def get_vehicles (user_id, vins=None, stock_nums=None, fields=None, min_last_seen=None, max_last_seen=None):
    '''
    Looks up a set of vehicle records for one user given a set of vins, a set of stock numbers, or a last_seen range.
    Fetches only the fields specified.
    '''


def upsert_vehicle (user_id, vin, attributes):
    '''
    Upserts a vehicle record, updating only the fields that appear in `attributes`.
    Certain fields are treated in a special way:
        - first_seen is only set if not already set, and is otherwise ignored
    '''
```

Migration: Step 2

- Create a **second** implementation of the interface that reads and writes DynamoDB tables
- Deploy **both** implementations
- Now all **new** data is written to SQL **and** DynamoDB



```
interfaces
    implementation_dynamodb
        __init__.py
        chrome_vin_to_image_id.py
        pageview.py
        table.py
        vehicle.py
    implementation_sql
        __init__.py
        chrome_vin_to_image_id.py
        pageview.py
        table.py
        vehicle.py
    __init__.py
    chrome_vin_to_image_id.py
    pageview.py
    vehicle.py
```

Migration: Step 3

- Migrate existing data from SQL to DynamoDB
- We wrote scripts to do this a little at a time
- They took days to run
- Now **all** data is in both SQL and DynamoDB

Migration: Step 4

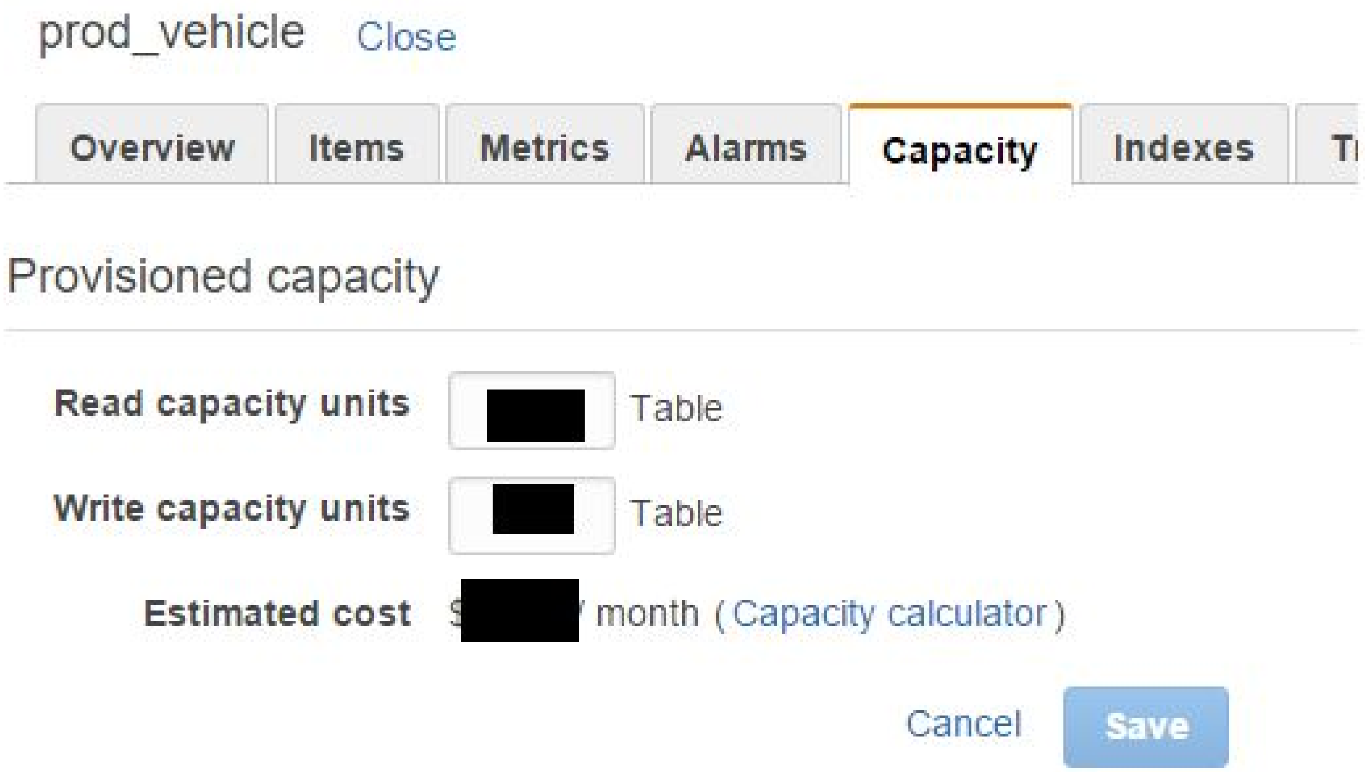
- Test the heck out of it!
- We ran regression tests to verify software features continued to function as they used to
- We spot-checked randomly selected records from the SQL and NoSQL database to make sure they matched

Migration: Step 5

- Stop invoking the SQL interface
- Delete the SQL interface, which is now dead code
- All reads and writes now use DynamoDB!

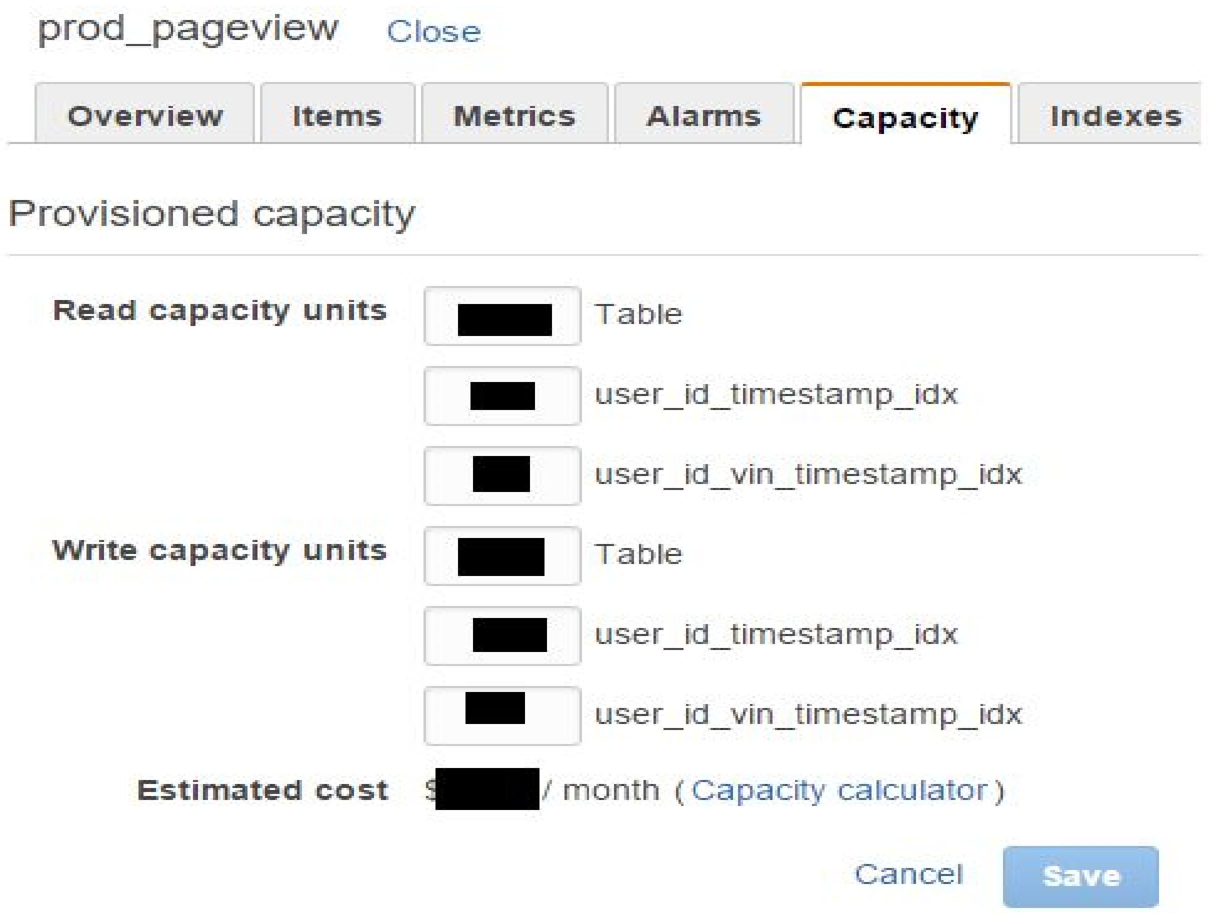
DynamoDB capacity planning

- DynamoDB offers unlimited read and write capacity
  - But only if you ask for it
  - And pay for it!
- Amazon calls this “provisioned throughput”
- Separate provisioning for each table
- Separate provisioning for reads and writes
- Separate provisioning for indexes (discussed later)


prod_vehicle Close

Provisioned capacity

Read capacity units	[redacted]	Table
Write capacity units	[redacted]	Table
Estimated cost	$[redacted] / month (Capacity calculator)	


prod_pageview Close

Provisioned capacity

Read capacity units	[redacted]	Table
	[redacted]	user_id_timestamp_idx
	[redacted]	user_id_vin_timestamp_idx
Write capacity units	[redacted]	Table
	[redacted]	user_id_timestamp_idx
	[redacted]	user_id_vin_timestamp_idx

Estimated cost $[redacted]/ month (Capacity calculator)

RCUs and WCUs explained

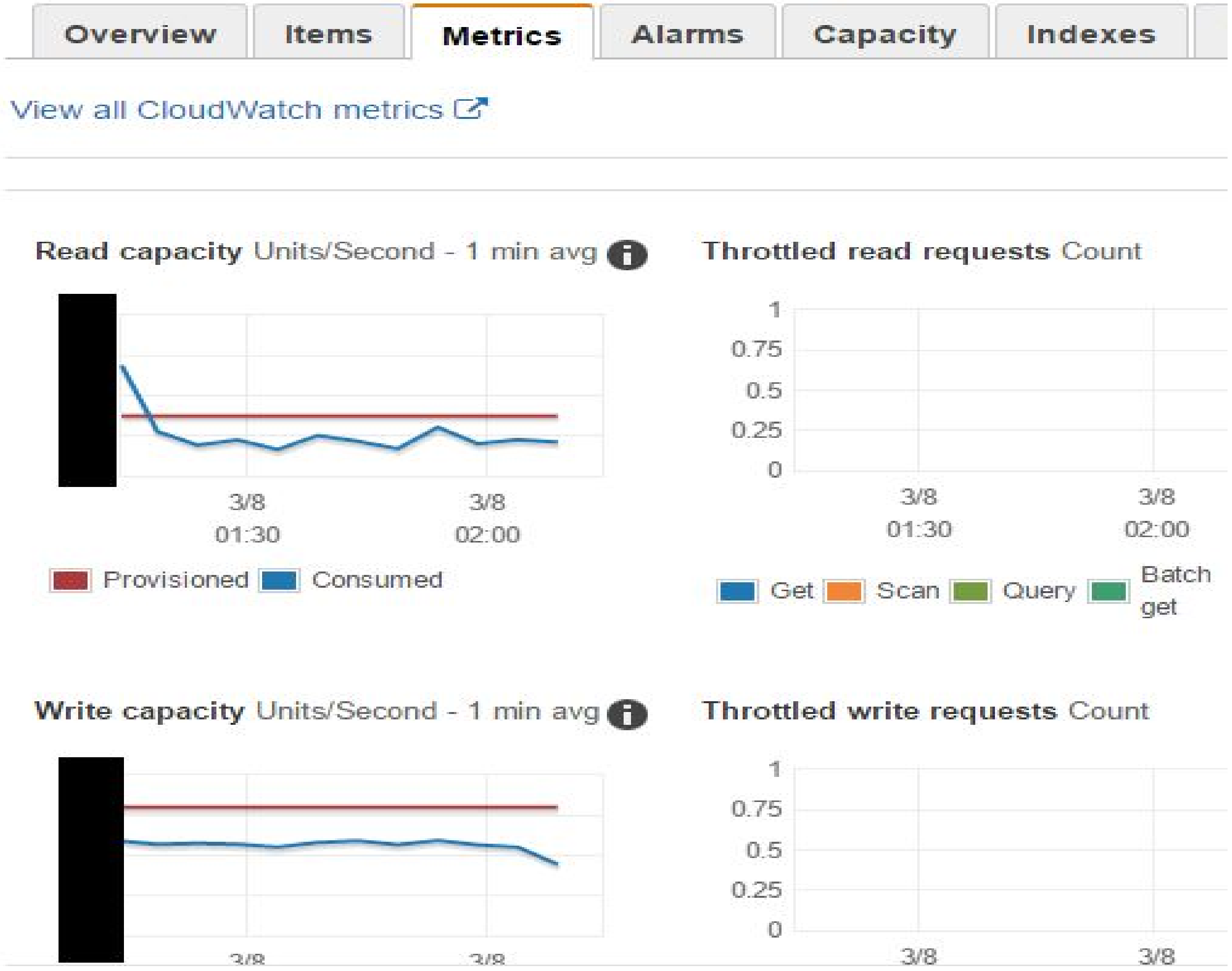
- RCU = Read Capacity Unit
  - One read capacity unit represents one strongly consistent read per second
  - Or two eventually consistent reads per second
  - Up to 4 KB per item
- WCU = Write Capacity Unit
  - One write capacity unit represents one write per second
  - Up to 1 KB per item
- http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ProvisionedThroughput.html

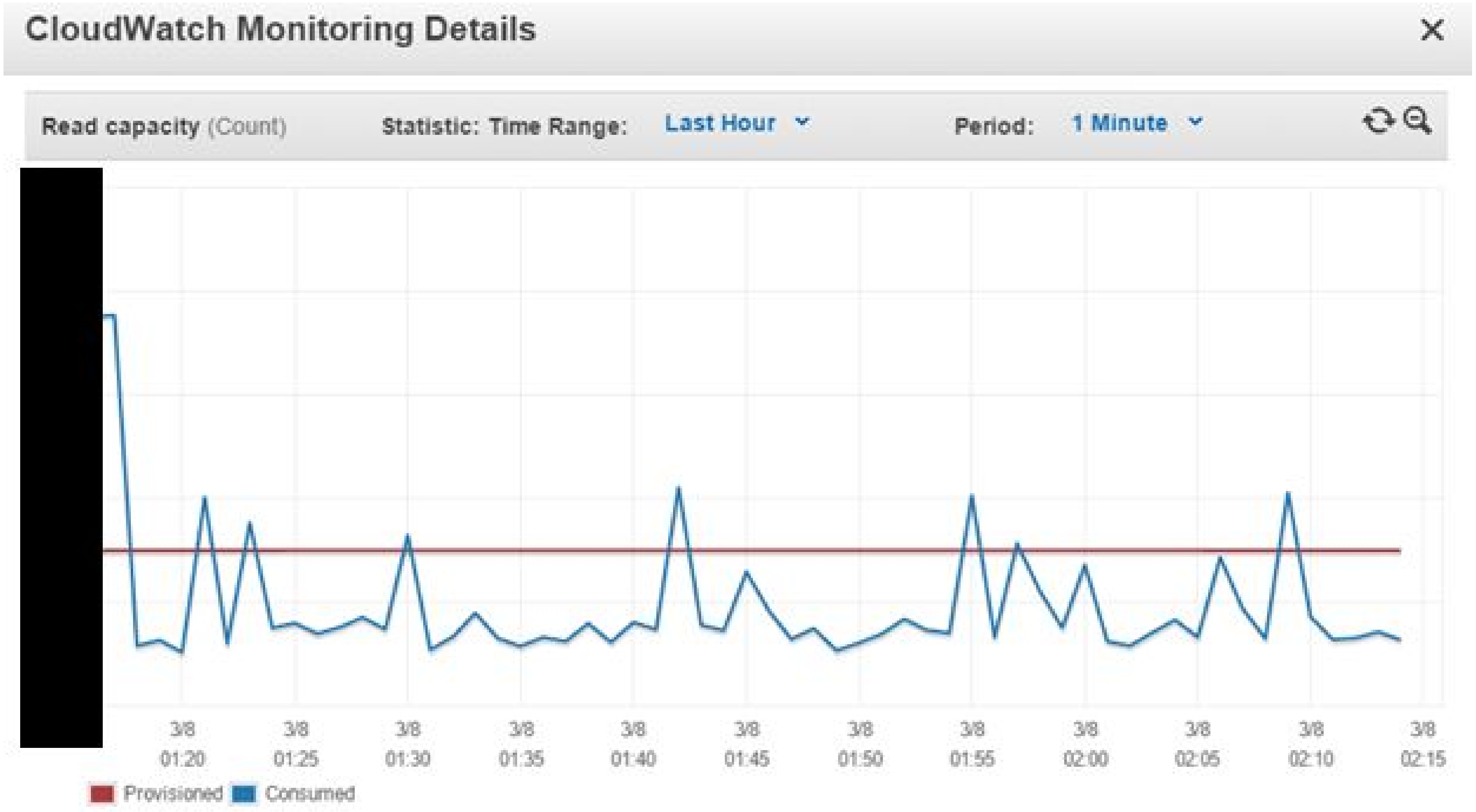
How much throughput do I need?

- It depends!
- Some ways to estimate:
  - If you’re migrating an existing application to DynamoDB, measure the rate of reads and writes before migration
  - Extrapolate from known quantities like number of page visits or number of orders processed, multiplying by your best guess as to number of reads and writes per visit, order, etc
  - Take a wild guess, err on the side of overprovisioning, and adjust as needed

How do I know if I got it right?

- Compare provisioned and actual throughput in the AWS console
  - It's okay to **occasionally** exceed provisioned capacity, thanks to credits that accumulate
  - If you see frequent throttled requests, you need more capacity
- Set a CloudWatch alarm to alert you (email, text message, etc) when capacity comes close to being exceeded


Overview
Items
Metrics
Alarms
Capacity
Indexes
View all CloudWatch metrics
Read capacity Units/Second - 1 min avg
[redacted]
3/8 01:30
3/8 02:00
🟥 Provisioned 🟦 Consumed
Throttled read requests Count
1
0.75
0.5
0.25
0
3/8 01:30
3/8 02:00
🟦 Get 🟧 Scan 🟩 Query 🟩 Batch get
Write capacity Units/Second - 1 min avg
[redacted]
3/8
3/8
Throttled write requests Count
1
0.75
0.5
0.25
0
3/8
3/8


CloudWatch Monitoring Details
Read capacity (Count)
Statistic: Time Range: Last Hour
Period: 1 Minute
3/8 01:20
3/8 01:25
3/8 01:30
3/8 01:35
3/8 01:40
3/8 01:45
3/8 01:50
3/8 01:55
3/8 02:00
3/8 02:05
3/8 02:10
3/8 02:15
🟥 Provisioned
🟦 Consumed
[redacted]

DynamoDB pricing

- Three components to DynamoDB cost
  - Provisioned throughput
  - Storage
  - Data transfer
- https://aws.amazon.com/dynamodb/pricing/

DynamoDB pricing: provisioned throughput

- $0.0065 per hour for every 50 units of Read Capacity
  - That is, $0.00013 / RCU
- $0.0065 per hour for every 10 units of Write Capacity
  - That is, $0.00065 / WCU
- Writes cost 5x as much as reads
  - 10x as much as eventually consistent reads

DynamoDB pricing: storage

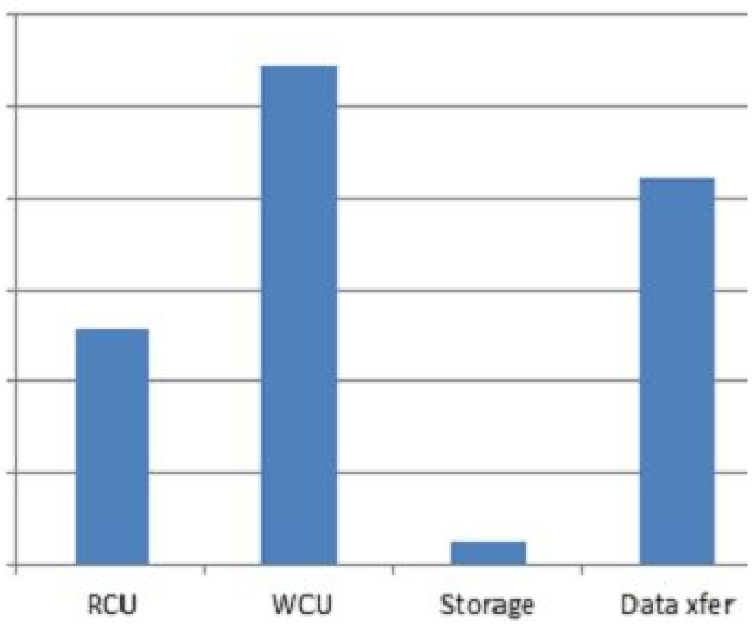
- 25 GB free
- Thereafter, $0.25 / GB / month
- Actual storage used will somewhat exceed the size of your data due to indexing

DynamoDB pricing: data transfer

- Data transfer IN is free
- First 1 GB / month of data transfer OUT is free
- Thereafter, data transfer OUT is $0.09 / GB

DynamoDB pricing: a real-world example

- From SpinCar's Feb 2016 bill:
  - RCU: xxxx RCU-hrs @ $0.00013 = $xxxx
  - WCU: xxxx WCU-hrs @ $0.00065 = $xxxx
  - Storage: xxxx GB-months @ $0.25 = $xxxx
  - Data transfer: xxxx GB @ $0.09 = $xxxx
- Total ~ $xxx / month
- About 10% of our total AWS bill
- Serving xxx customers
- Cost is expected to scale linearly with number of customers

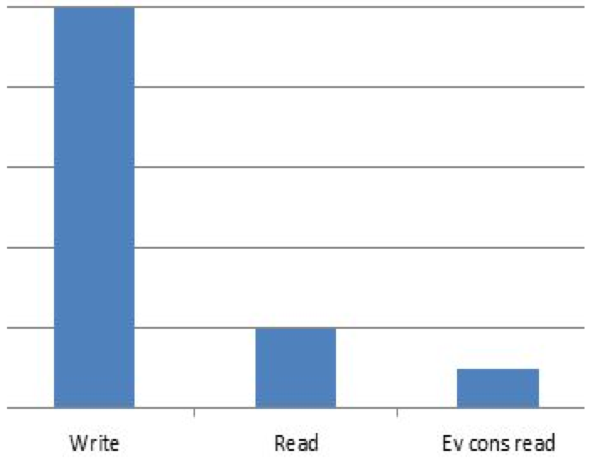


Comparison to RDS pricing

- SpinCar’s January RDS bill was $xxxx
- We were recently able to downgrade to a smaller RDS instance, on track to cut RDS cost by almost half
- We avoided upgrading to a larger RDS instance that would have doubled cost

How to save $ on DynamoDB

- Reserve pricing
  - 1- or 3-year commitment and a modest up-front payment
  - Almost 90% reduction in hourly rate for RCUs and WCUs
- Don’t write unless you have to
  - It’s worth doing a read to see if anything needs to be updated before you write
- Especially if the write will update an index
  - Each access to an index costs as much as a write to the underlying table
  - Doing a read to possibly avoid a write is extra-worthwhile if the attributes you’re writing include an indexed attribute



DynamoDB indexes

- Unlike with a relational database, you can’t query on just any attribute
- Unless you create an index
- A “local secondary index” is limited to the same “partition key” as the underlying table
- A “global secondary index” can have any attribute(s) as the key
    - Separate provisioned throughput for each global secondary index
- Indexes can be complicated; Amazon’s documentation provides details

Backing up DynamoDB databases

- SQL backups are pretty easy
- RDS backups are really easy: just check a box, and you can restore to any point in time
- DynamoDB backups, not so easy
  - No “set it and forget” backup available
  - May not be easy to “freeze” data in a consistent state for backup
  - Beware using up your provisioned read throughput on backups
- SpinCar’s backup strategy
  - Custom Python script
  - Runs many times per day
  - Selects a small subset of the data, reads it from DynamoDB, writes it to a file on S3
  - Another script can read and restore the backup files

DynamoDB + SQS -- perfect together

- DynamoDB provides unlimited scalability
- But that doesn’t make it bulletproof
  - If throughput is exceeded, reads and writes can be throttled; you could lose data
  - Bugs in, or maintenance of, your app could prevent DynamoDB access
  - Unlikely, but DynamoDB itself could experience downtime or latency
- A queue is a great way to provide protection
- Application writes to queue
- Separate process consumes the queue and writes to DynamoDB
- Like DynamoDB, Amazon SQS queues are infinitely scalable
- Unlike DynamoDB, you don’t have to reserve capacity in advance

Questions?