Use Logical Decoding to build your own application cache

By Blagoj Atanasovski

Powered by SORSIX

Who am I

- Software Engineer at Sorsix
  - https://www.sorsix.com/
- I work on:
  - Backends for Web Applications
  - Solutions for Fast Data Processing
  - And other stuff

Caching

- A cache is a hardware or software component that
  - stores data so that future requests for that data can be served faster
  - might be the result of an earlier computation
  - or a copy of data stored elsewhere
- Hits are served by reading data from the cache
  - faster than recomputing a result or reading from a slower data store
  - the more requests served from the cache, the faster the system performs

Different caches

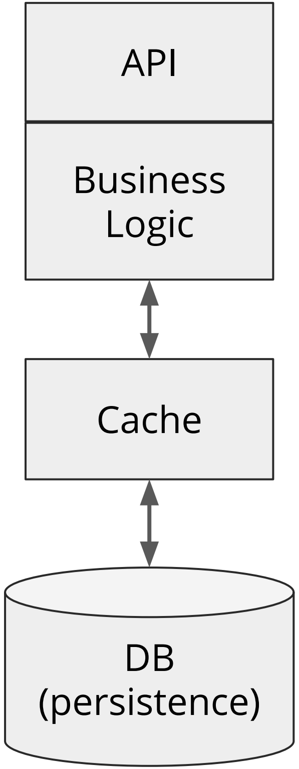
- Local browser cache
  - On clients computer
  - HTML, CSS, JavaScript, graphics or other multimedia files
  - Only good for static files - content is not static
- Web cache (HTTP cache)
  - Web server, CDN or ISP stores copies of documents passing through it
  - Cross-requests cache
  - Only good for static files
  - Client may request fresh copy explicitly, max-age, last-modified header, PUT/POST/DELETE invalidation

Different caches - Application cache

- Cache in our application (business logic)
  - You can cache everything very easy and fast
  - You can read from the cache also easy and fast
  - Invalidating it in a correct moment is nightmare
- Types of application cache
  - In Process
    - Same heap - super fast, any object, no serialization, perfect for single node applications
    - No sharing between servers, gone on restart
  - Out-of-process
    - Shared cache between servers, can handle application restart
    - Serialization (same network, different network),

Application cache - Invalidation

- Cache is between database and business logic
  - Module is responsible for everything cache related
  - All read/write operations go through the module
  - Good luck introducing this to a large codebase
  - What about foreign keys to your cached data?
  - Can you distribute it?
  - You can use an existing solution
    - How many out there with persistence in Postgres?
    - Are you going to use NoSQL?
    - What if you need to rollback?
  - Build your own
    - What we did, but a bit differently



What is logical decoding?

Write-Ahead Log (WAL)

- Ensuring data integrity.
- Changes to data files must be written only after those changes have been logged
- After log records describing the changes have been flushed to permanent storage.
- No need to flush data pages to disk on every transaction commit

Logical Decoding

- Introduced in 9.4
- Plugin infrastructure (Extensible, Adaptable)
- The process of extracting all persistent changes to a databases tables into
    - Coherent
    - easy to understand format
    - interpreted without detailed knowledge of the database's internal state.
- Implemented by decoding the contents of the write-ahead log
    - into an application-specific form such as a stream of tuples or SQL statements
- Relies on Replication Slots

Replication Slots

- In the context of logical replication
  - Stream of changes
  - Can be replayed to a client in the order they were made on the origin server
  - Each slot streams a sequence of changes from a single database.
- Each has an identifier that is unique across all databases in a cluster
- Persisted independently of the connection
- Crash-safe

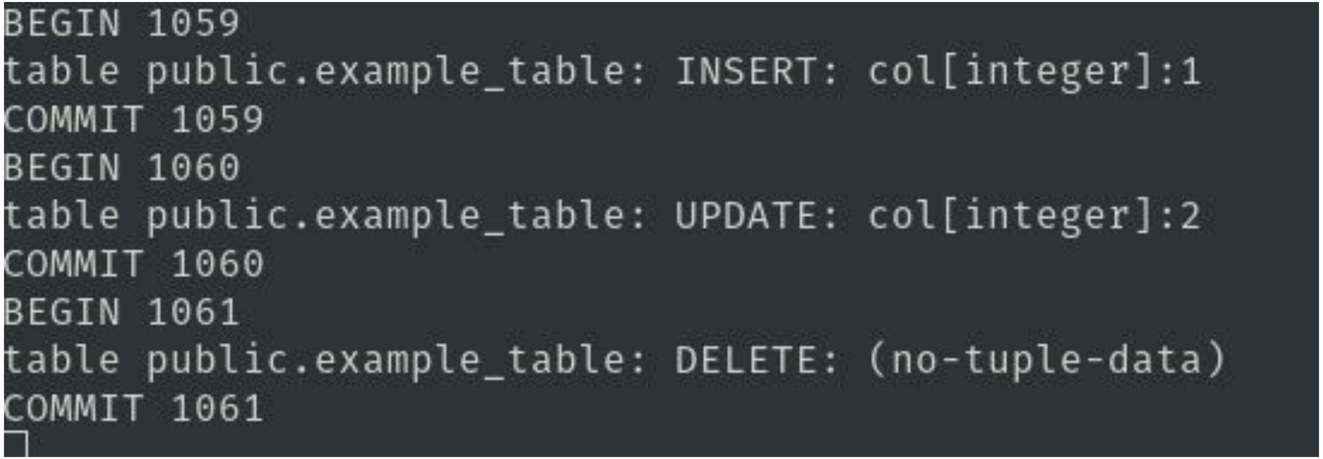
Replication Slots

- Each change is emitted only once
  - Current position of each slot is persisted only at checkpoint
  - In case of a crash, the slot returns to an earlier LSN
  - Changes will be resent on server restart
- Up to logical decoding clients to handle same message more than once
  - May record the last LSN they saw

Logical Decoding Plugins

- The format in which those changes are streamed is determined by the output plugin used
- An example plugin is provided in the PostgreSQL distribution
- Additional plugins can be written to extend the choice of available formats without modifying any core code
- Every output plugin has access to each individual
  - new row produced by INSERT
  - old new row version created by UPDATE
  - The id and old version of a row removed with DELETE

Example Logical Decoding Output

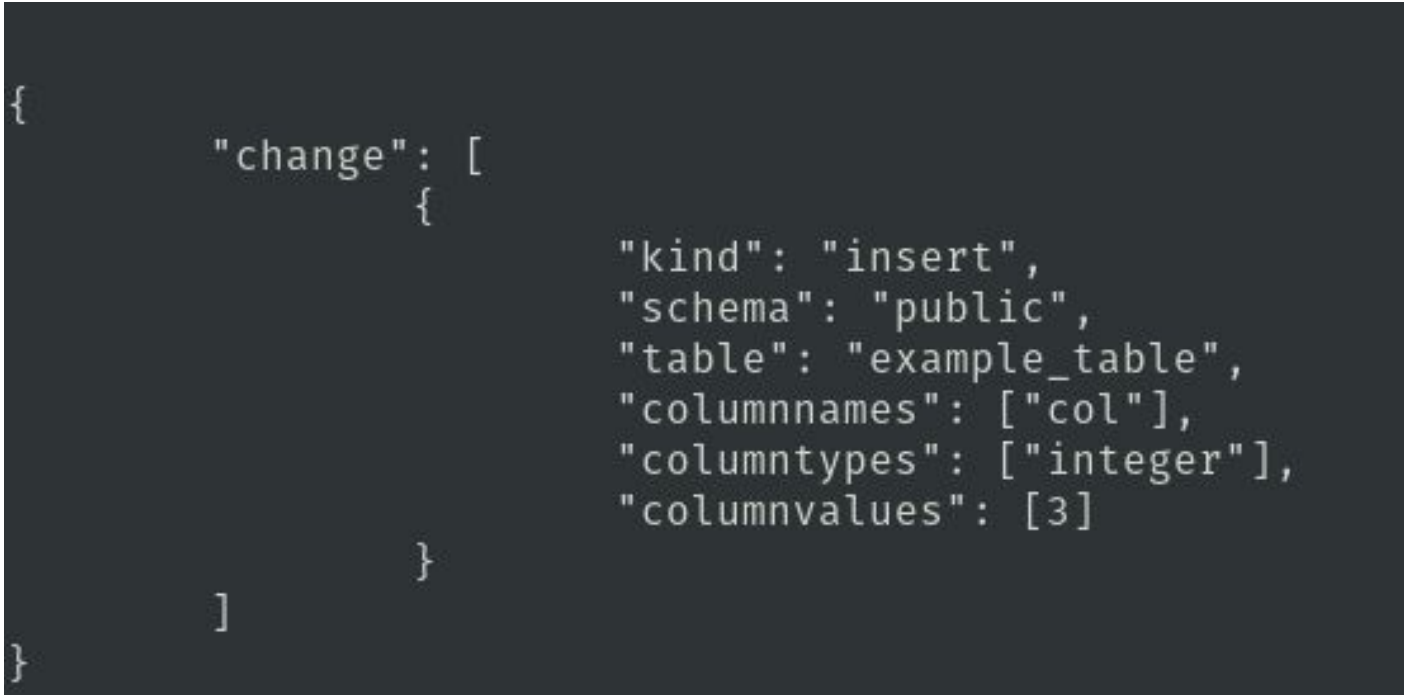


```
BEGIN 1059
table public.example_table: INSERT: col[integer]:1
COMMIT 1059
BEGIN 1060
table public.example_table: UPDATE: col[integer]:2
COMMIT 1060
BEGIN 1061
table public.example_table: DELETE: (no-tuple-data)
COMMIT 1061
```

Logical Decoding Plugins

- Changes can be consumed
  - using the streaming replication protocol
  - Or by calling functions via SQL
- It is the responsibility of the plugin to produce the desired output the consumer expects and to filter out unnecessary changes

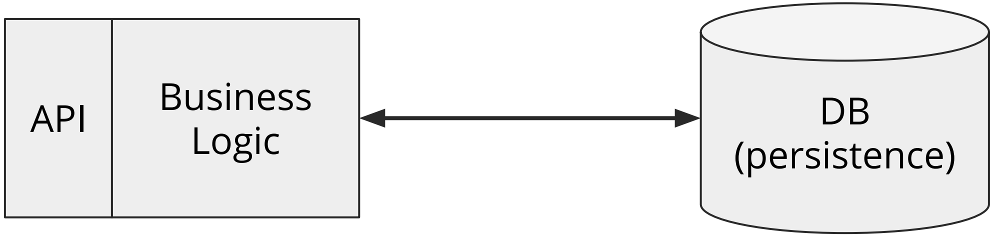
Example Output of Wal2Json



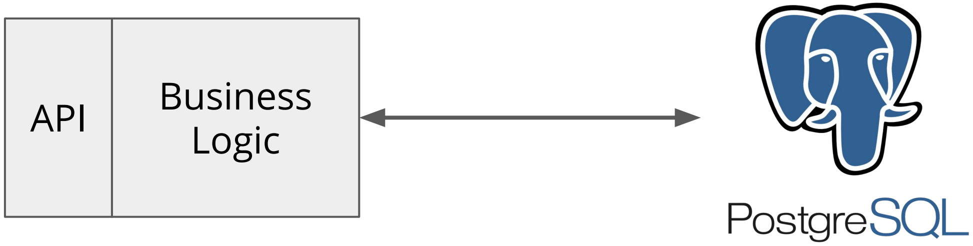
```
{
        "change": [
                {
                        "kind": "insert",
                        "schema": "public",
                        "table": "example_table",
                        "columnnames": ["col"],
                        "columntypes": ["integer"],
                        "columnvalues": [3]
                }
        ]
}
```

Building our cache

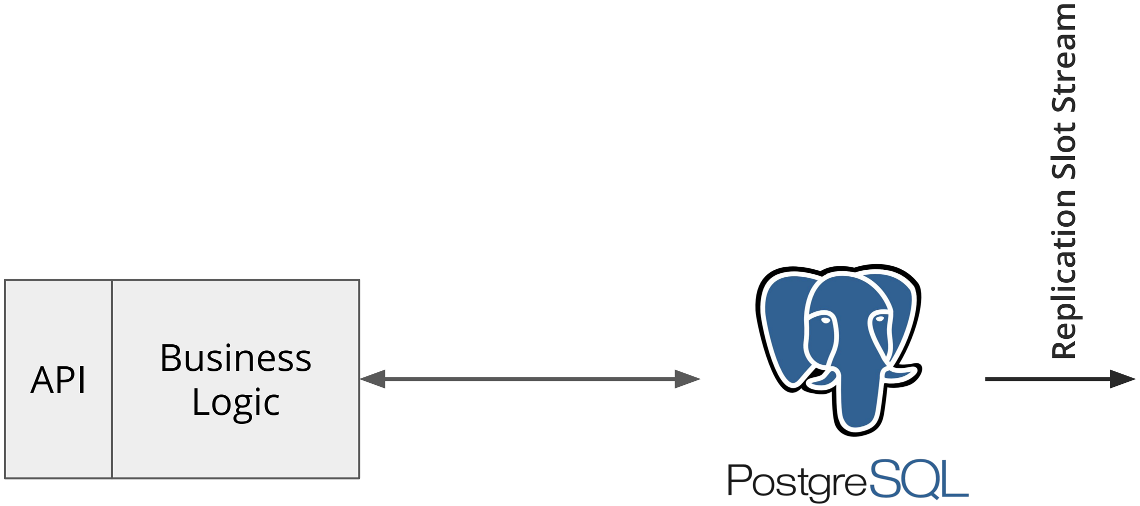
An app and a database



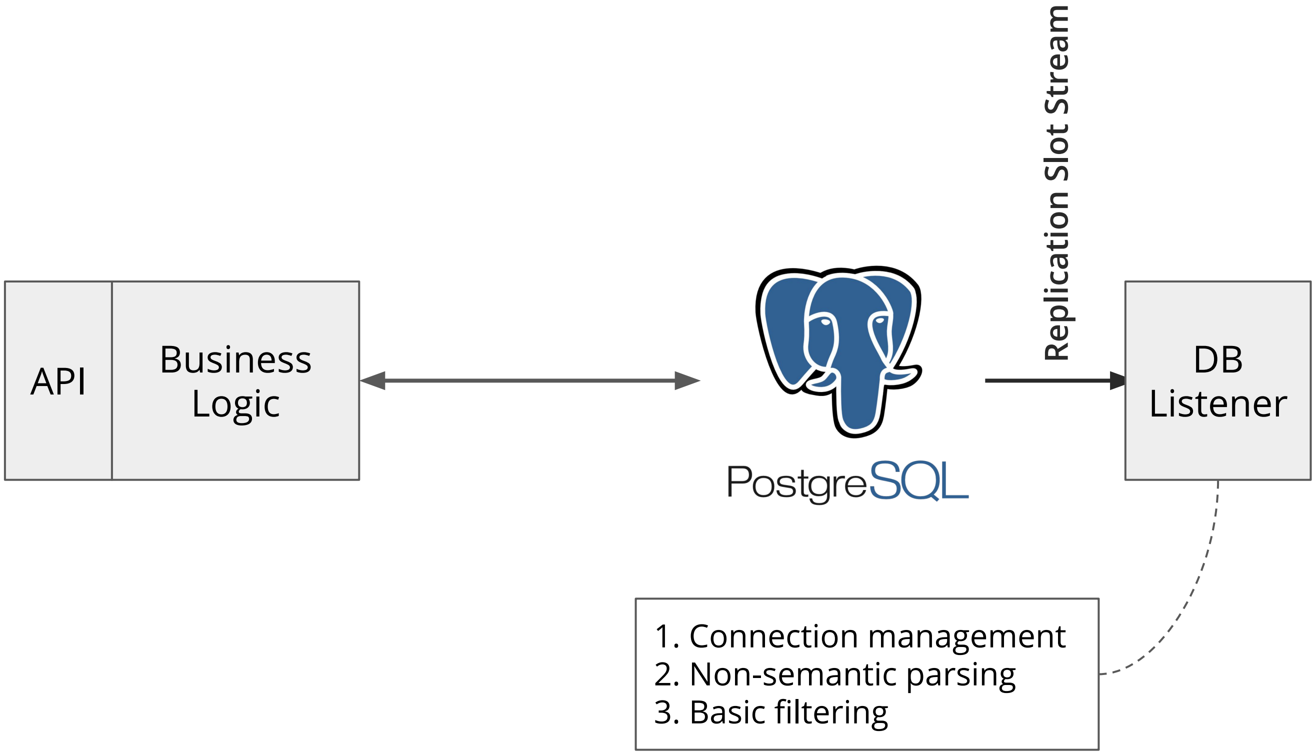
An app and a database



An app and a database

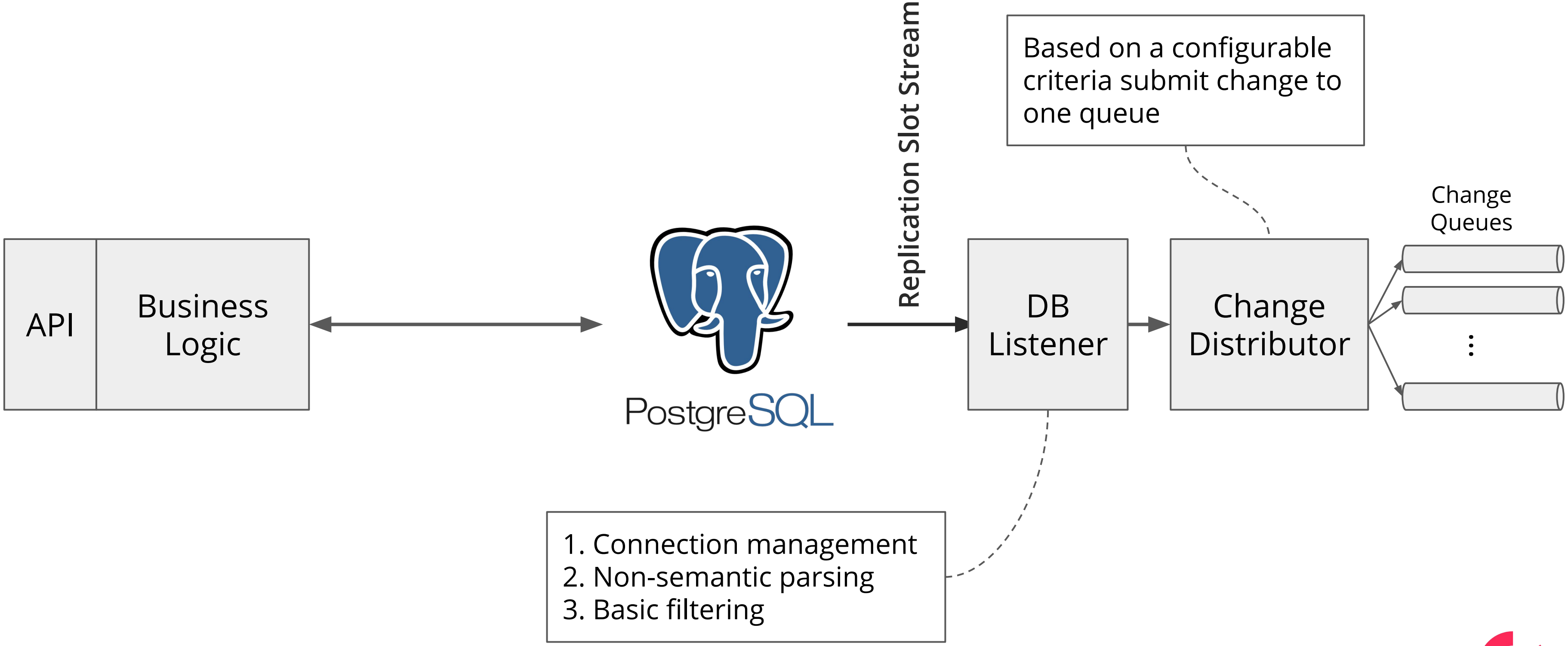


An app and a database



1. Connection management
2. Non-semantic parsing
3. Basic filtering

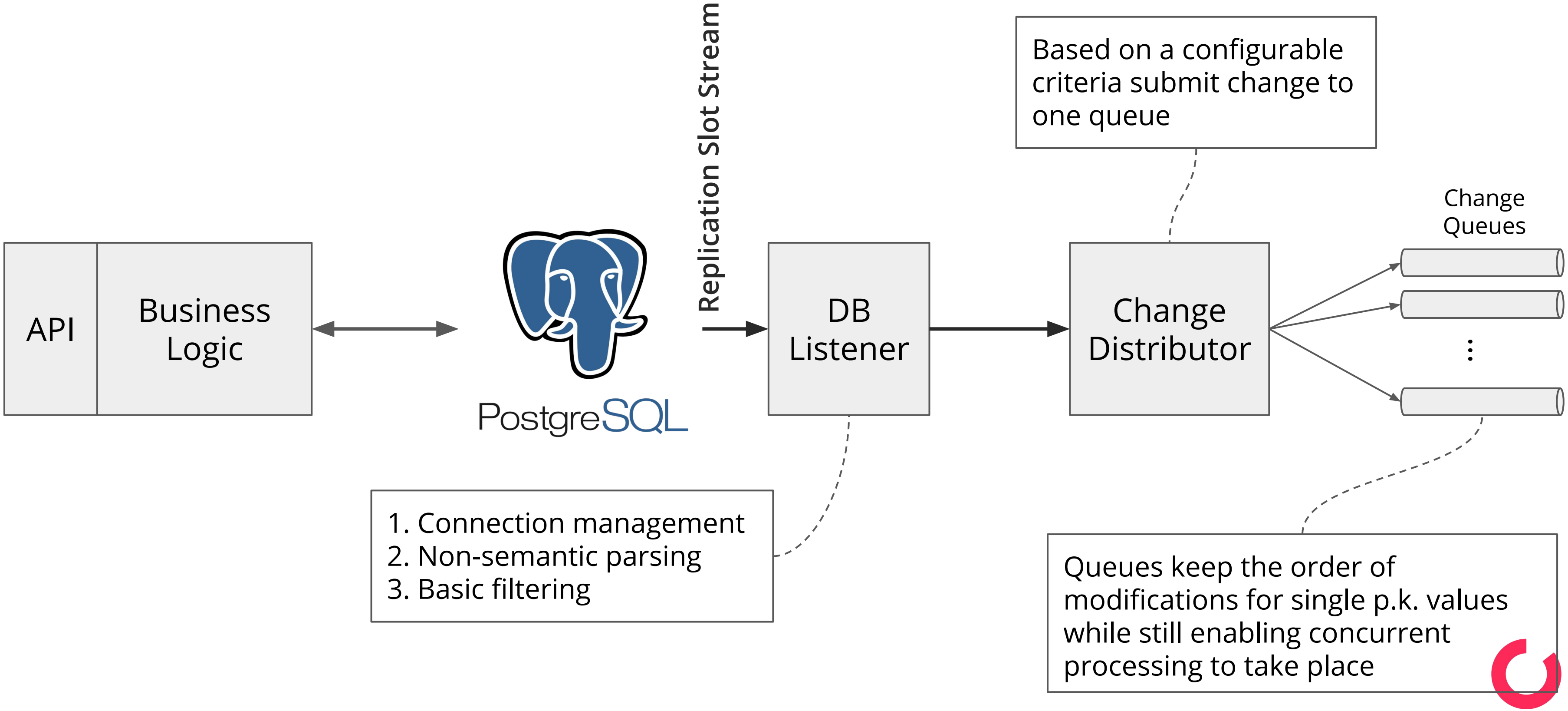
An app and a database



Based on a configurable criteria submit change to one queue

1. Connection management
2. Non-semantic parsing
3. Basic filtering

An app and a database



API

Business Logic

PostgreSQL

Replication Slot Stream

DB Listener

Change Distributor

Change Queues

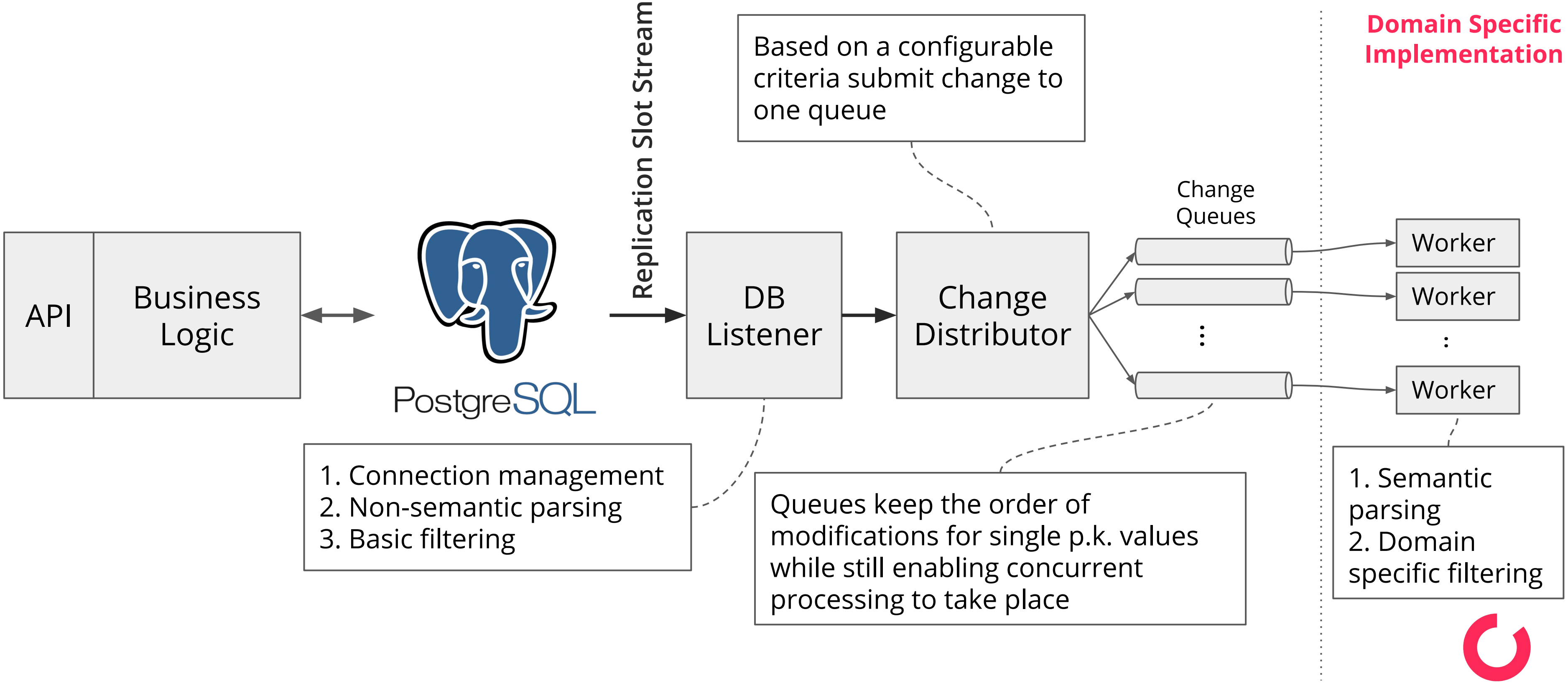
⋮

Based on a configurable criteria submit change to one queue

1. Connection management
2. Non-semantic parsing
3. Basic filtering

Queues keep the order of modifications for single p.k. values while still enabling concurrent processing to take place

An app and a database



API

Business Logic

PostgreSQL

Replication Slot Stream

DB Listener

Change Distributor

Change Queues

Worker

Worker

Worker

Domain Specific Implementation

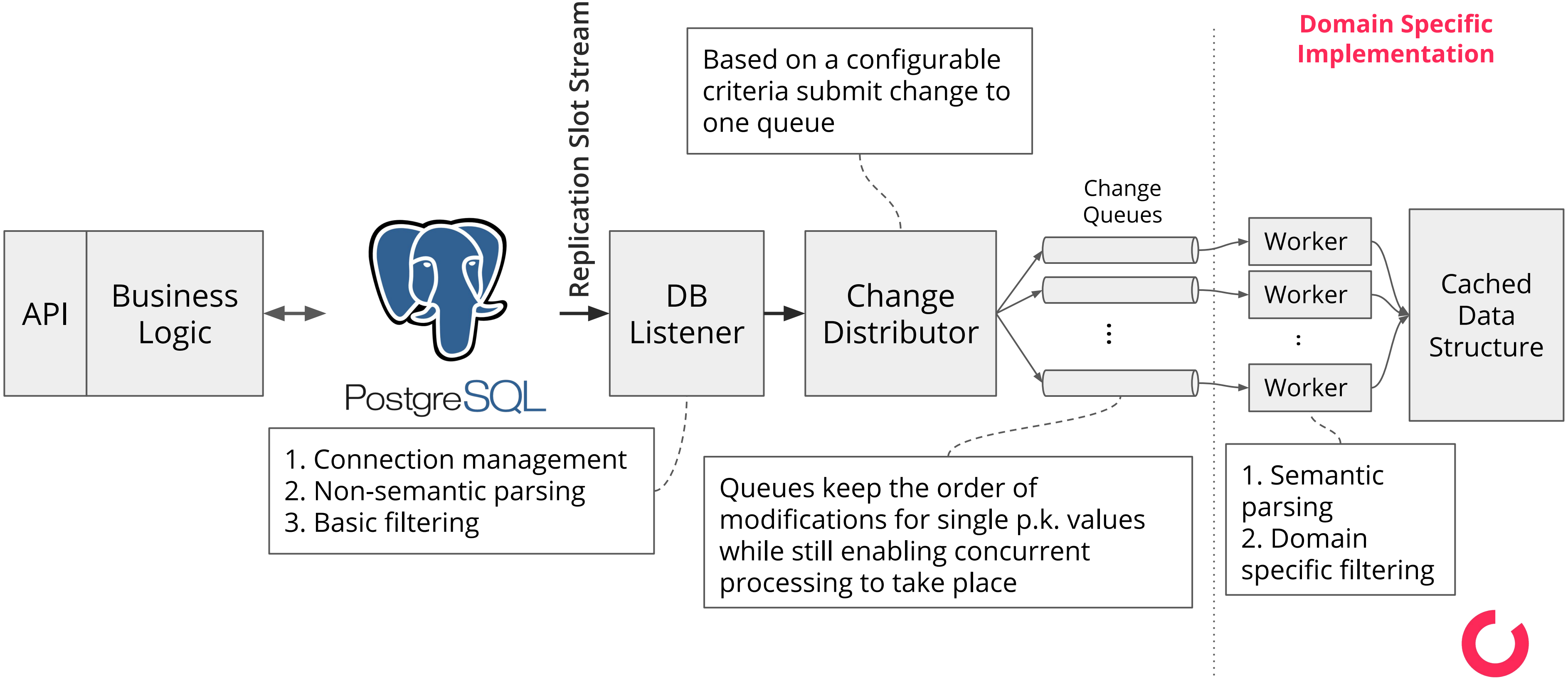
Based on a configurable criteria submit change to one queue

1. Connection management
2. Non-semantic parsing
3. Basic filtering

Queues keep the order of modifications for single p.k. values while still enabling concurrent processing to take place

1. Semantic parsing
2. Domain specific filtering

An app and a database



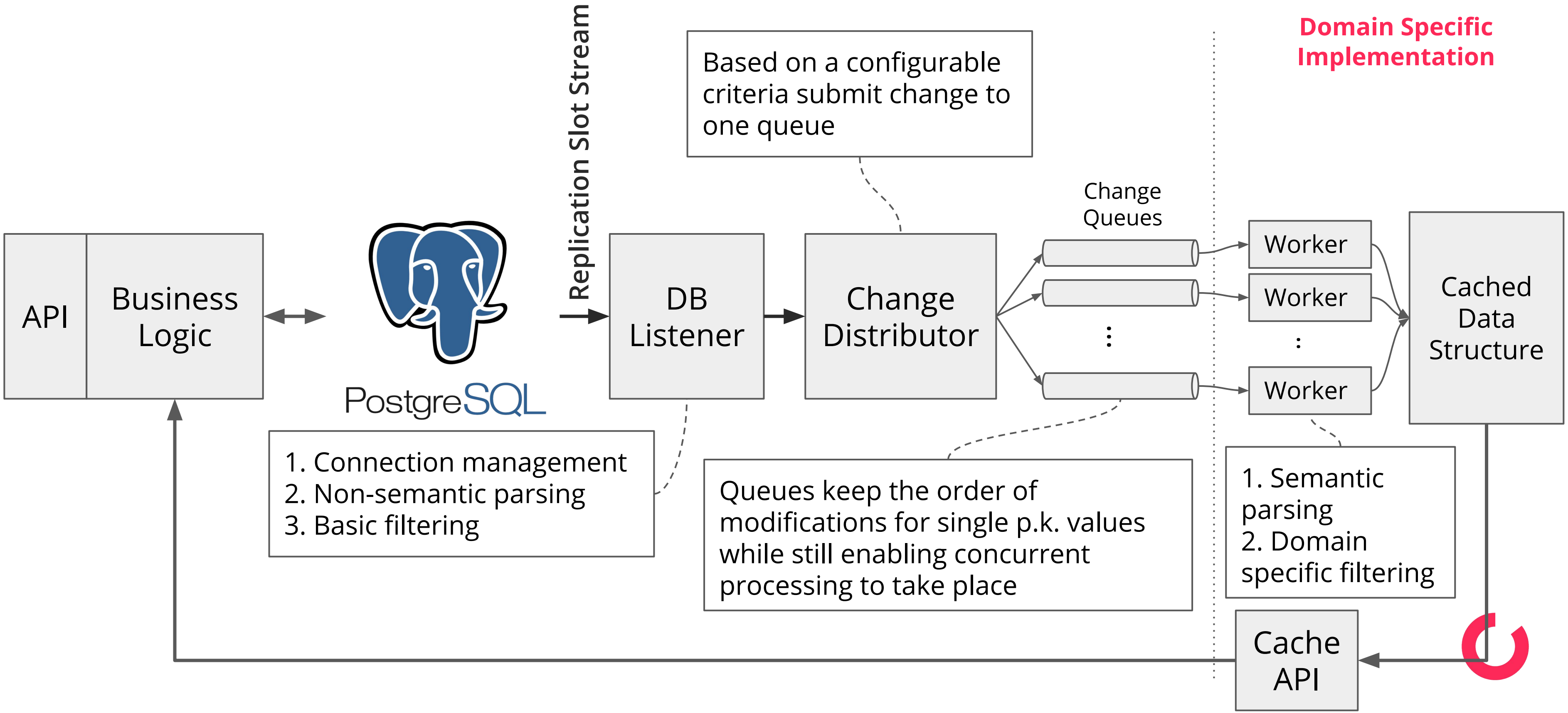
Based on a configurable criteria submit change to one queue

1. Connection management
2. Non-semantic parsing
3. Basic filtering

Queues keep the order of modifications for single p.k. values while still enabling concurrent processing to take place

1. Semantic parsing
2. Domain specific filtering

An app and a database



Based on a configurable criteria submit change to one queue

1. Connection management
2. Non-semantic parsing
3. Basic filtering

Queues keep the order of modifications for single p.k. values while still enabling concurrent processing to take place

1. Semantic parsing
2. Domain specific filtering

Advantages of logical decoding for caches

- Consistency and invalidation become trivial
  - No need to change your application code to update the cache every time you write something to the database that should be cached
  - No need for complex caches that handle the write-back for you (can you trust them?)
  - No need to worry about constraints failing after you’ve updated the cache
  - No expensive queries needed to keep cache up to date

Advantages of logical decoding for caches

- Separate development
  - You can work on your cache independently
  - Only need to know what data needs to be cached, and define an access method
  - Don’t need to know where the cache is going to be used
  - Focus on the logical decoding stream

Advantages of logical decoding for caches

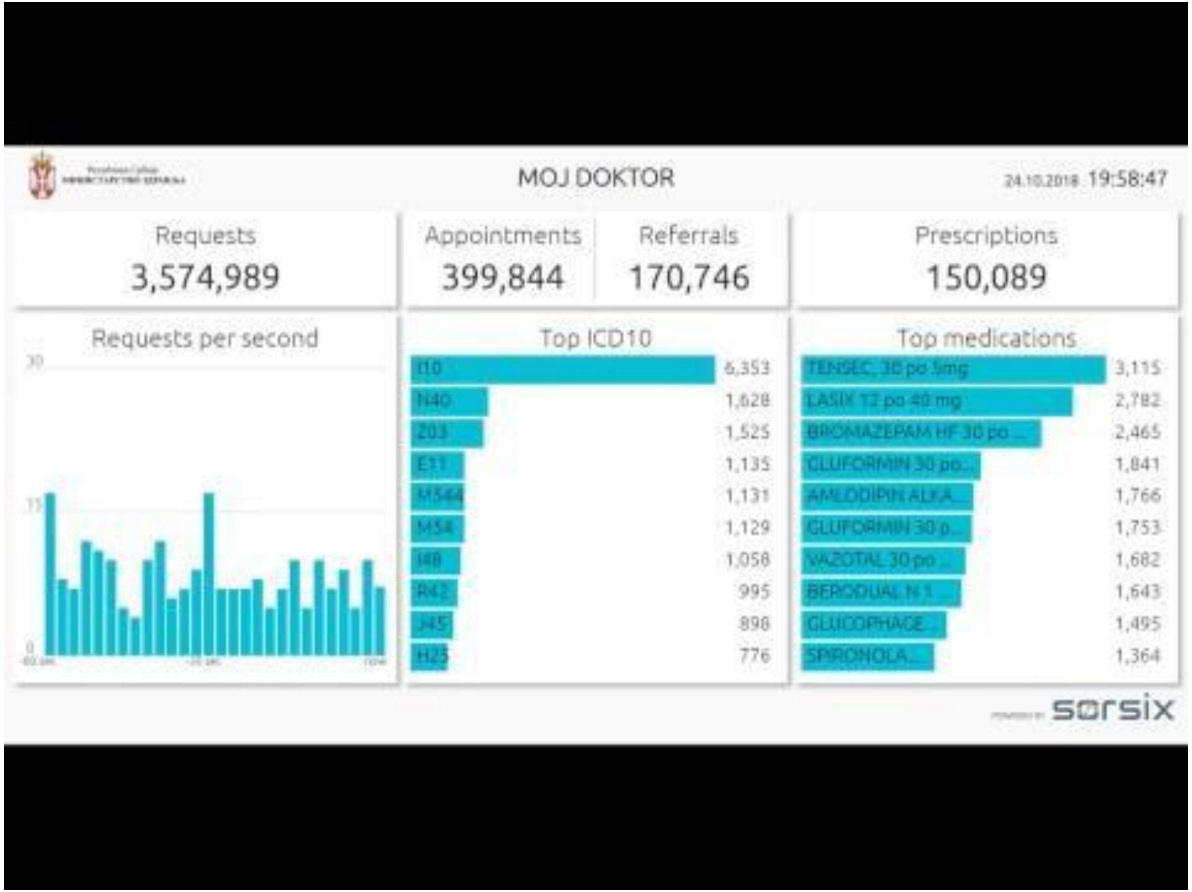
- Adoption can be one step at a time
    - The cache is independent, start using it one query at a time
    - Gradual adoption
    - Safe, can always fall back to the database
    - Impact can be measured with each step
    - No changes needed for saving/updating values
    - Can choose if a stale value is ok or latest one is required for each requirement

Sorsix Pinga
Example and Results

Sorsix Pinga

- National end to end EHR platform
- Serving a combined 10 million population
- Running live in Macedonia & Serbia

Sorsix Pinga Rollout in Serbia - serbia-rollout.sorsix.com/



MOJ DOKTOR

24.10.2018 19:58:47

Requests	Appointments	Referrals	Prescriptions
3,574,989	399,844	170,746	150,089

Requests per second

Top ICD10

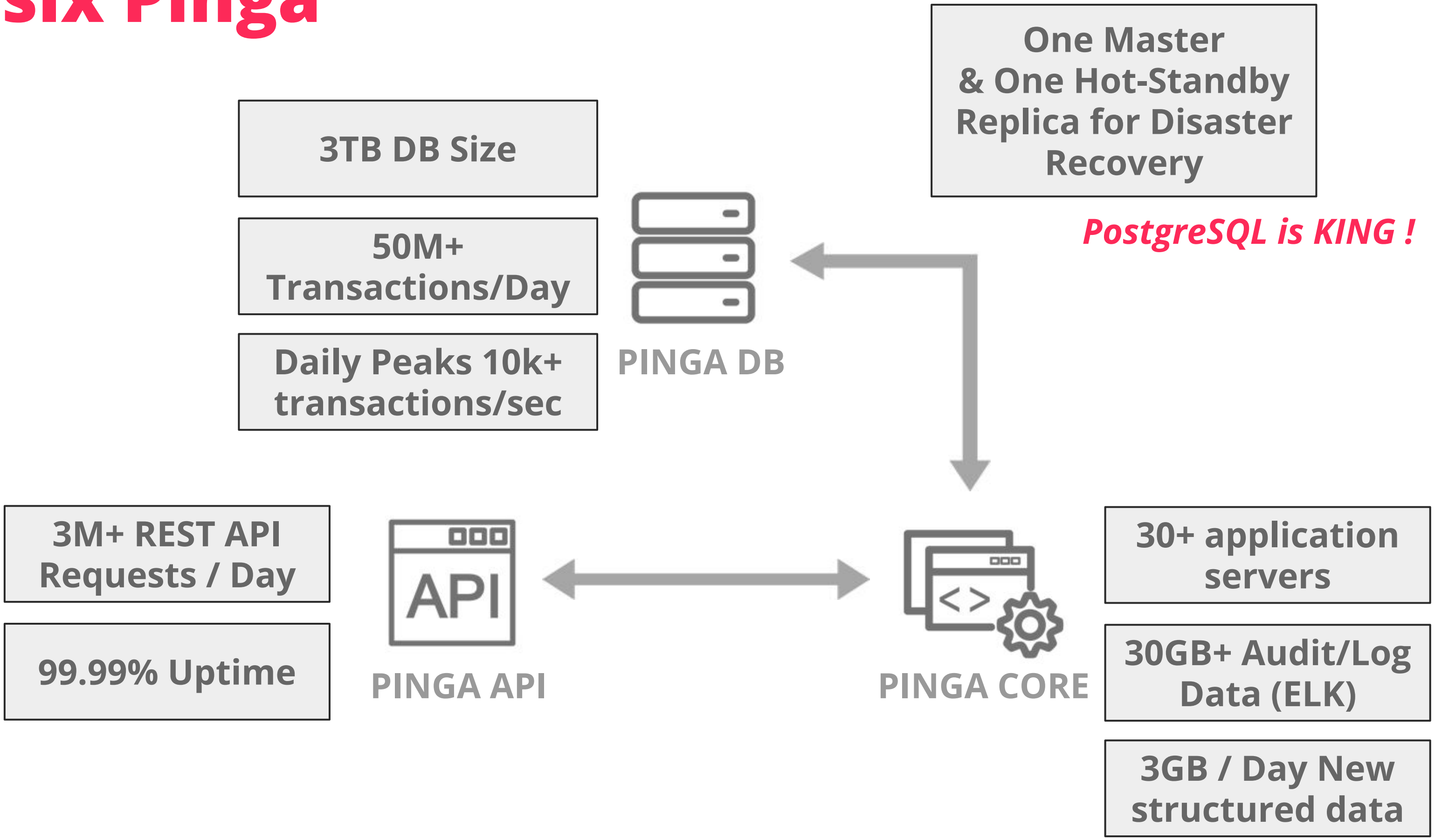
I10	6,353
N40	1,628
Z03	1,525
E11	1,135
M544	1,131
M54	1,129
I48	1,058
R42	995
J45	898
H25	776

Top medications

TENSEC, 30 po 5mg	3,115
LASIX 12 po 40 mg	2,782
BROMAZEPAM HF 30 po ...	2,465
GLUFORMIN 30 po...	1,841
AMLODIPIN ALKA...	1,766
GLUFORMIN 30 p...	1,753
VAZOTAL 30 po ...	1,682
BERODUAL N 1 ...	1,643
GLUCOPHAGE...	1,495
SPIRONOLA...	1,364

Live Referral and Prescription Dashboard

Sorsix Pinga



3TB DB Size

50M+ Transactions/Day

Daily Peaks 10k+ transactions/sec

PINGA DB

One Master & One Hot-Standby Replica for Disaster Recovery

PostgreSQL is KING !

3M+ REST API Requests / Day

99.99% Uptime

PINGA API

PINGA CORE

30+ application servers

30GB+ Audit/Log Data (ELK)

3GB / Day New structured data

Sorsix Pinga - Issues

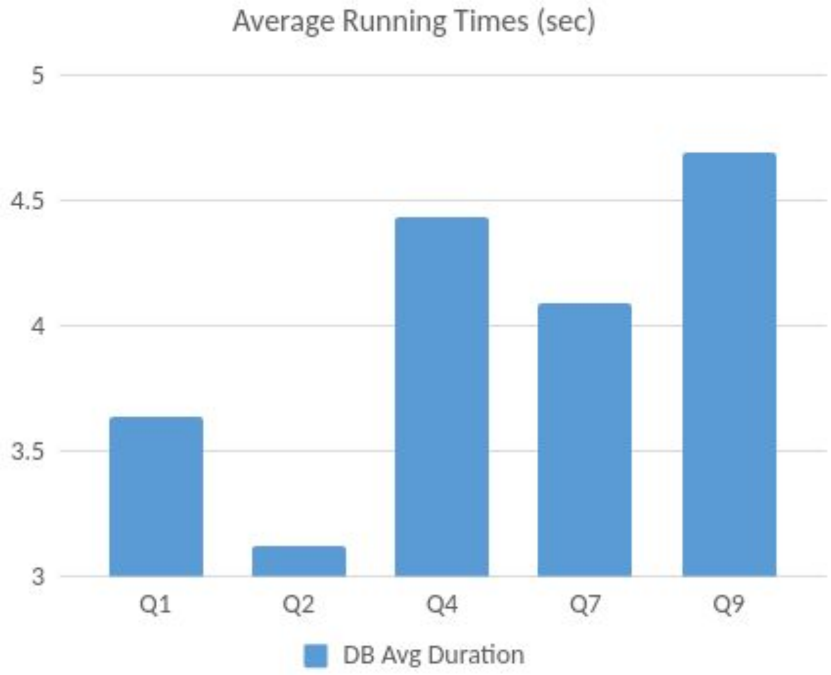
- Setup
  - Database optimized for fast insert and update
  - Indexes on important columns
- Requirement
  - Aggregate and window queries that look ahead in the future
  - Selection based on user input (can be a combination from 1 to 10 different predicates)
- Problem
  - Requirement is executed by almost every user every time they use the system
  - Queries in requirement run in > 4 seconds time

Sorsix Pinga - How to fix it

- Constraints
  - System is live
  - System handles the most crucial of personal data
  - The more limited a change is - the more safe it is
  - The faster a change is implemented - the more benefit there is
- Solution
  - Build an independent cache
  - Integrate cache one query at a time
  - Fail-safe in case cache fails - just execute old code (go to db)

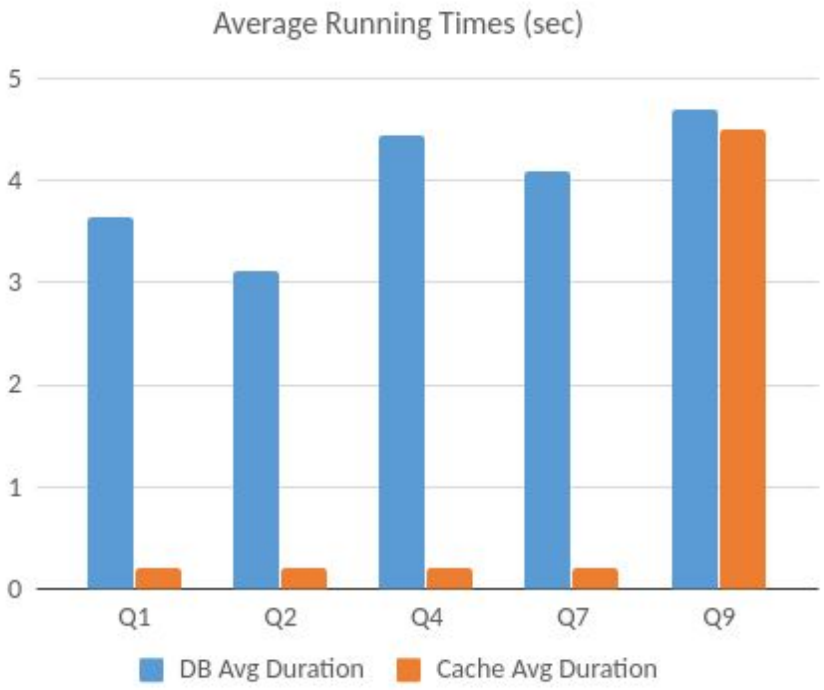
Results

- From PgBadger
- The top 5 out of the top 10 slowest queries were replaced with requests to the cache
    - Average between 3.1s and 4.6s



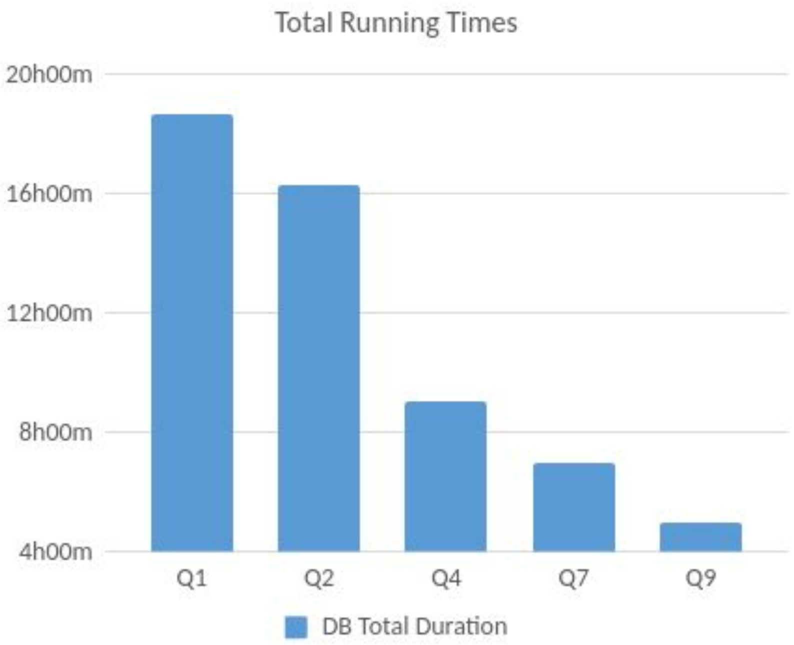
Results

- From PgBadger
- The top 5 out of the top 10 slowest queries were replaced with requests to the cache
  - Average between 3.1s and 4.6s
- Cache returns a result on average
  - in 0.2s (50ms lookup +150ms transfer and serialization)
  - For queries 1, 2, 4 and 7
  - 4.5s for query 9



Results - Total Time

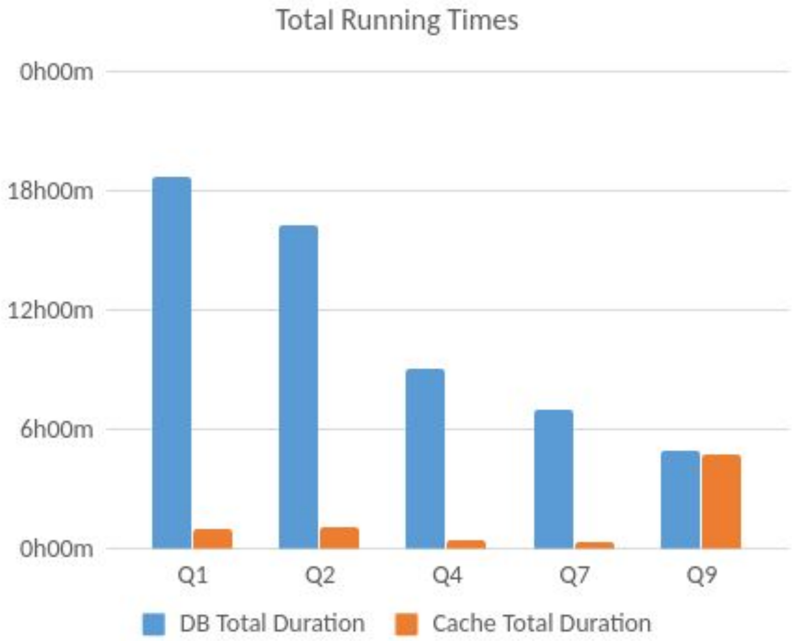
- Number of times query is executed:
  - Q1 - 18,474
  - Q2 - 18,785
  - Q4 - 7,333
  - Q7 - 6,146
  - Q9 - 3,812
- DB Total Time: 168h20m (on all queries)



Results - Total Time

- Speedup per query:
  - Q1 - 18.19
  - Q2 - 15.62
  - Q4 - 22.18
  - Q7 - 20.43
  - Q9 - 1.04
- DB Total Time: **168h20m** (on all queries)
- Saved Time: **48h21m**

28.72%


Total Running Times
0h00m
18h00m
12h00m
6h00m
0h00m
Q1
Q2
Q4
Q7
Q9
🟦 DB Total Duration
🟧 Cache Total Duration