# Transparently Scale-out SQL Databases with Data Grids

Erik Brandsberg, CTO
Heimdall Data

# Agenda

1. Rolling Your Own Solutions

   Considerations when implementing a cache & read/write split in your application

2. How Database Proxies Solves the Problems For You

   Automated cache and read/write split driven by rules, not code

3. Live Demo & Q&A

# What are the Target Features

## Caching

A cache is a high-speed data storage layer that stores a subset of data, typically transient in nature, so that future requests are served up faster.

## Read/Write Split

The act of dividing database statements into reads vs. writes. Read/write split allows applications to leverage scaled-out databases and auto-scale load more easily.

***Both together provide database scale-out functionality***

# Rolling Your Own Cache

It can be done… Anything can be done, but should you?

At the surface, caching is simple:

- Create a key based on the request
- Serialize the result returned
- Store in the cache with the key
- Check if the key exists in the cache before requesting again from the source

BUT… it becomes complex very fast:

- When is the data invalidated?
- Is your cache adding latency on cache misses?
- Are you over-caching, and flushing objects that could provide a benefit?
- Have you accounted for the security context of the object (user permissions?)
- Debugging can be difficult

# Invalidation

## The hardest part of caching is knowing what to throw out

Definition: *Removing cached objects that may have been changed at the source of truth*

**Hints to make invalidation less painful:**

- Classify content by broad categories, and invalidate on writes to the category, i.e. DB tables instead of rows
- Use timestamps on the category to determine if something is stale
- Avoid walking the cache to evict objects--use TTL settings to do the bulk of the work and only explicitly evict if a stale object was retrieved
- Broadcast category invalidations via a pub/sub interface to other nodes

*IMDGs supports TTL based invalidation and pub/sub messages to facilitate invalidation*

# Reduce Latency

A network hop is a hop, it doesn’t matter what it is to, the impact is the same

Network latency often overlooked and a critical component of data performance

Accessing the same data from a cache and DB often are the same speed

**Hints to reduce the impact of latency:**

- Avoid cache misses by tracking keys in the remote cache
- Avoid caching when a cache hit performs the same on the DB as the cache (and adapt)
- Use local memory for frequently used objects
- Optimize serialization overhead--not all serialization schemes are created equal

*Most IMDGs supports keyspace notifications to update nodes what keys are available, use it!*

# More can be Less

## Selective caching can improve performance overall

When caching, over-caching can reduce cache benefit by adding unnecessary overhead

Caching isn’t free-there is serialization overhead and memory tradeoffs

**Hints to avoid over-caching:**

- Limit the size of objects that can be cached (when appropriate) to avoid purging memory of small objects that are more likely to be used
- Track the actual cache benefit based on invalidation category--if a category provides no benefit, stop caching it
- Track invalidation frequency, if a category is invalidated too often, simply stop caching that category (for a time)

# Always Think Security

## Are you bypassing your database security by caching?

Cached data can expose critical data to intruders, or malicious employees

Leverage available identifiers as part of the cache key to prevent exposure

**Hints on securing your cache:**

- Lock down the cache to authorized users
- Include the user that accessed the data as part of the cache, at least by default
- Store objects only in local memory if unencrypted, and use TLS if stored over the wire
- Encrypt the data on serialization if necessary
- Beware of serialization vulnerabilities that can trigger remote code issues

*IMDGs generally include password login, TLS support, as well as on-disk encryption*

# Read/Write Split

## Simple to do, difficult to master

Deceptively simple, just:

- Open two connections, one to the write node, one to a read node
- Send queries to the connection desired

But… is it that simple?

- Replication lag means a read following a write may not see the written data, SQL Server may take a full second before exposing the written data
- Read after write is a common pattern, and broken by read/write split often
- Transaction logic complicates things more
- How to scale beyond a single read-only node?
- Increases the risk of failures by adding more DB nodes

# Read/Write split, Continued

- Replication lag can be estimated, or a simple “safe” value assumed, say 10s
- Cache invalidations on writes can be used to track the last write time to a table, but requires cache invalidation logic to be in place
- Leverage the last write time to determine if it is safe to read from the read-only server

There is no easy way to do this “right” without much of the cache logic already being in place except for limited reference table lookups.

*There is no easy way to do caching and read/write split*

# Transparent Database Proxies

# Database Proxy Vendors

| Feature | HEIMDALL DATA | Pg-Bouncer | MariaDB | ProxySQL |
|---|---|---|---|---|
| Automated Failover | ✓ | ✓ | ✓ | ✓ |
| Read/Write split | ✓ | ✓ | ✓ | ✓ |
| Database Vendor Neutral | ✓ | | | |
| Automated Cache invalidation | ✓ | | | |
| Reduces network latency | ✓ | | | |

# IMDG vs. Database Proxies

Pg-Bouncer ProxySQL

- Best scale & performance
- Greenfield applications
- Requires code changes

- “Good enough”
- Existing applications, small dev
- **No code changes**

# Heimdall Proxy Abstraction Layer

Abstracting the hard parts away from the application

Batch Processing
OLTP/OLAP Routing
Query Triggers

Application

Heimdall Driver/Proxy

Automated Failover
Load Balancing
Read/Write Split

PostgreSQL

Microsoft SQL Server

MySQL

redis

SQL Auto-Caching
Auto-invalidation
Auto-Cache Refresh

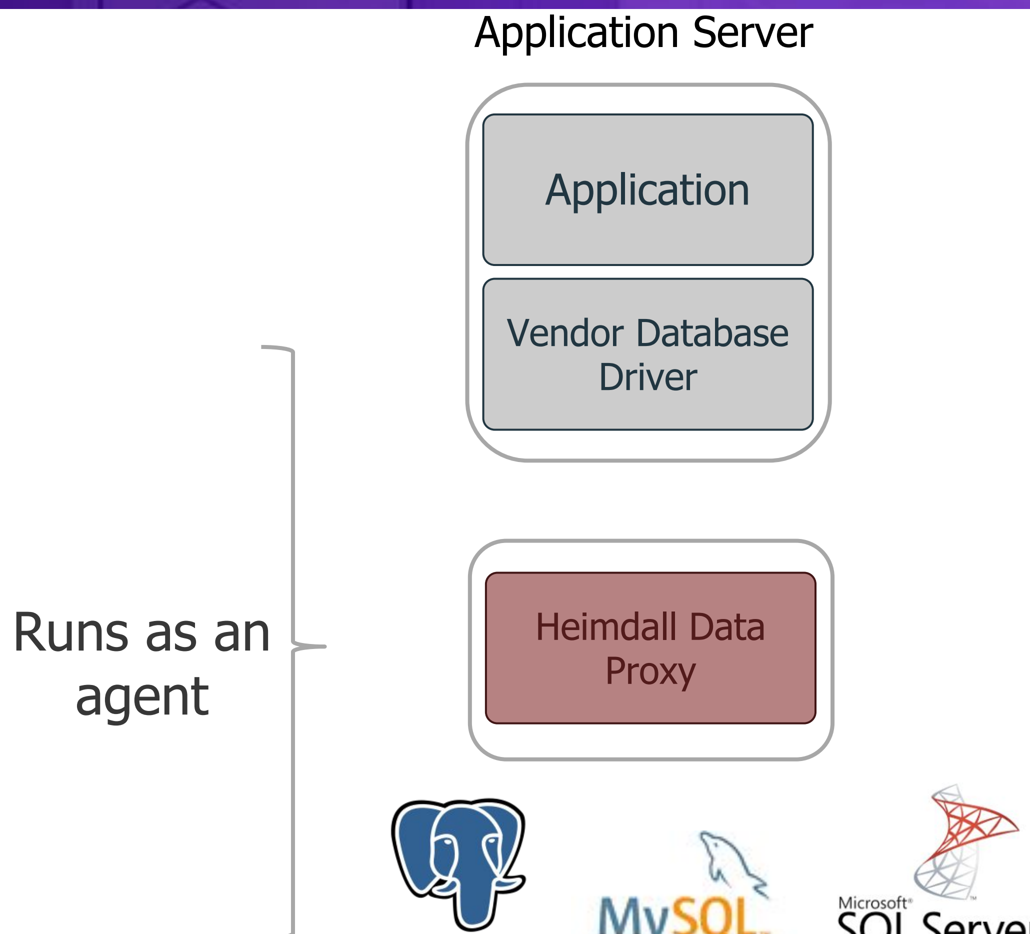
Query Analytics & Transformation

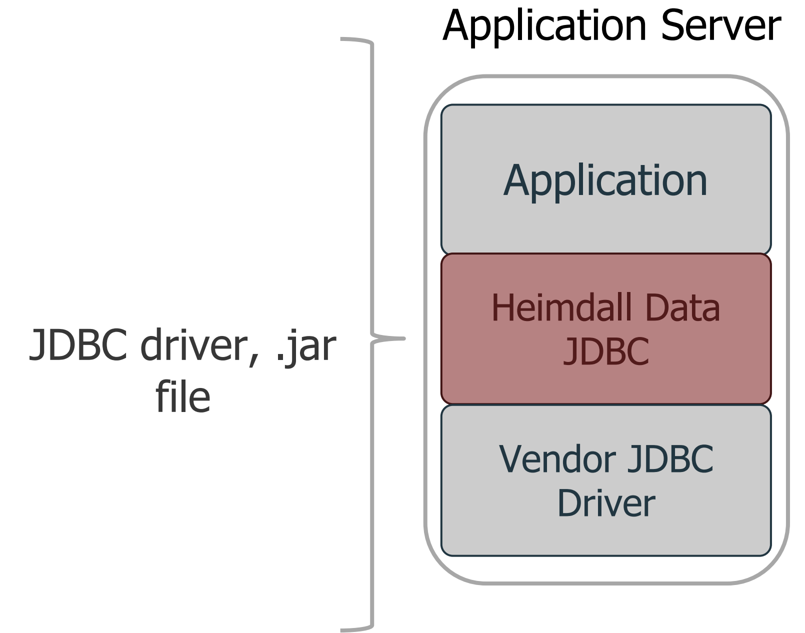
Connection Pooling and Multiplexing

GridGain

hazelcast

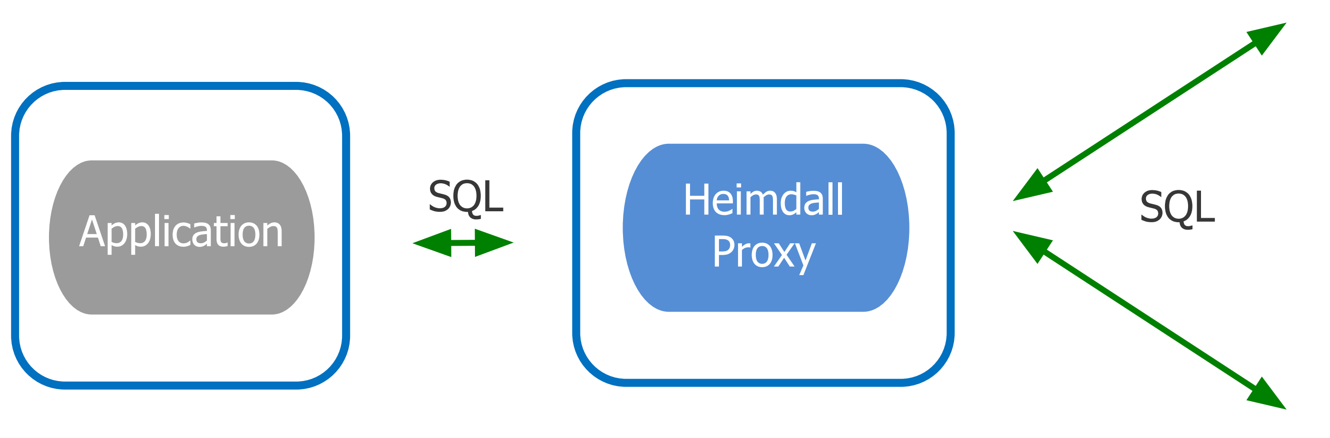
# Software Package Options

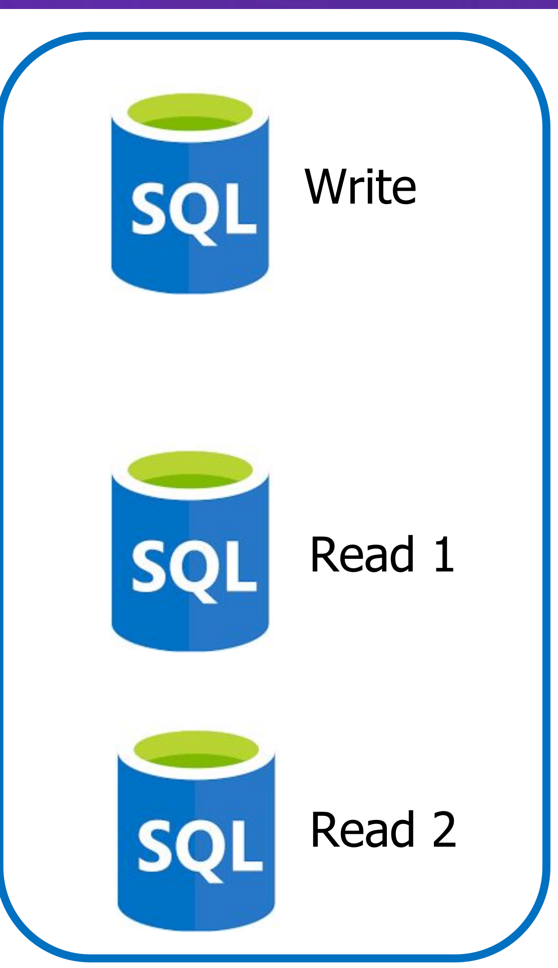



Any JDBC data source

# Read/Write Split with Replication Lag Detection



# Database Proxy Platform

SQL CACHING

AUTOMATED FAILOVER

BATCHING INSERTS

CONNECTION POOLING

READ/WRITE SPLIT

ACTIVE DIRECTORY

HEIMDALL DATABASE PROXY PLATFORM

RDS, Redshift

# Automated Caching and Read/Write Split by Rule

| # | Enabled | Regex | In-Trans | Action | Parameter | Value | Edit | Copy | Delete |
|---|---|---|---|---|---|---|---|---|---|
| 1. | ☑ | (?i)^\s*select | ☑ | Reader Eligible | | | | | ☐ |
| 2. | ☑ | Match all if empty | ☐ | Cache | ttl<br>maxLocalSi<br>maxSize | 5 m<br>1000000<br>10000000 | | | ☐ |
| 3. | ☑ | Match all if empty | | | | | | | |
| 4. | ☐ | Match all if empty | | | | | | | |

Queries Per Second

Overall, Slow, Cache, Database, Async

Average Query Time (µs)

Overall, Local Cache, Grid Cache, Database

Cache Hit Rate

All Cache, Local Cache, Grid Cache

Queries per Second (per server)

Master, none, Reader

Average Query Time (per server) (µs)

Master, none, Reader

Monitor Response Time (per server) (µs)

Master, none, Reader

# Live Demo

## How Heimdall Proxy Works Live

Three Demo applications are available on our website:

- Magento - Backed by MySQL
- Odoo - Backed by Postgres
- Wordpress – Backed by MySQL

Questions and Answers!

# For More Information

Please visit online at:

www.HeimdallData.com

Available on the AWS, Azure and CGP (soon) Marketplaces as well!

# Thank you