Next Generation JDBC Database Drivers for Performance, Transparent Caching, Load Balancing, and Scale-out

Ramon Lawrence
University of British Columbia
Kelowna, BC, Canada
ramon.lawrence@ubc.ca

Erik Brandsberg
Heimdall Data Inc.
Palo Alto, CA, USA
erik@heimdalldata.com

Roland Lee
Heimdall Data Inc.
Palo Alto, CA, USA
roland@heimdalldata.com

ABSTRACT

Despite having a significant impact on overall data system performance, database drivers connecting the application to the database system have not innovated at the same pace as the database systems themselves. This work describes a database driver designed for the requirements of cloud-based systems requiring flexibility, high availability, scaling, and performance. The unique contribution is a rule-based query routing system that supports real-time configurations and optimizations without requiring any changes to the application code or database system. With the increasing migration of applications and databases to the cloud as well as different database technologies such as NoSQL systems, this flexibility allows application owners to optimize and migrate legacy applications to exploit the advantages of new database technologies. Experimental results demonstrate how queries cached by the driver can improve query response times by an order of magnitude and reduce the overall load on the database system by up to 50%.

CCS Concepts

•Information systems → Database query processing; Structured Query Language;

Keywords

SQL, JDBC, cache, driver, load balancing, high availability

1. INTRODUCTION

There has been tremendous innovation in database technology with the introduction of new database systems such as NoSQL systems as well as the migration of database systems to cloud-based hosting. The combination of open source database systems and cloud-based hosting allows for improved scale-out and performance at a lower cost. New applications are designed to leverage these technologies, and even existing, in-house applications are migrating to the cloud. One of the major challenges with this migration is that converting legacy systems to use new database technology is expensive and risky and often involves significant code changes or complete rewrites. The cost of these changes often far exceeds the cost reduction benefits of migration.

Many applications use standard interfaces such as ODBC [2] and JDBC [1] to connect the application to the database. These standards are well-proven and have evolved over the years to add new features. Despite being designed for SQL-based, relational databases, ODBC and JDBC drivers are used for all sorts of systems including NoSQL databases. The standardization is a major advantage with interoperability between applications, databases, and query software tools. The driver's main purpose is to translate application requests to the network protocol for the database and return results in a form usable by the application.

This work describes a JDBC database driver implementation that expands the functionality of the driver and improves overall system performance. Although there has been some prior work on implementing advanced functionality including caching and load balancing (C-JDBC [3]) and data virtualization (UnityJDBC [6]) into JDBC drivers, our JDBC driver, called Heimdall, has a unique rule-based query routing system that allows it to support many more features without requiring any application or database modifications. Features supported include dynamic configuration of servers and databases including load balancing, query performance monitoring, analytics, caching recommendations, transparent caching, and in-memory scale-out. Performance testing demonstrates that the Heimdall JDBC driver can improve query response times by an order of magnitude compared to retrieving results from the database and can reduce the load on the database system by up to 50%.

2. BACKGROUND

To improve interoperability, standard database APIs such as ODBC [2] and JDBC [1] were developed. Database vendors provide ODBC or JDBC drivers that implement the API and perform the required protocols for their database. Since the driver is in the critical path between the application and database, the implementation of the driver and its efficiency can have a major impact on the overall performance that the application sees when accessing the database.

In the research community, there have been only a few ef-

SAC 2017,April 03-07, 2017, Marrakech, Morocco

http://dx.doi.org/10.1145/3019612.3019870

forts to add additional functionality to drivers. C-JDBC [3] was a JDBC-driver that performed auto-partitioning of data across databases for improved performance. It also had some features for load balancing and caching. UnityJDBC [6] implemented a relational database virtualization engine in a JDBC driver. This allowed for queries across multiple databases which appeared to the application as a single database. The UnityJDBC system was extended to support NoSQL databases such as MongoDB [4]. The commonality in these prior works is the goal of adding features without changing the database or the application. This is especially important for legacy systems whose cost for migration and modification is high. Improved drivers reduce hardware and software costs for the application and database servers and decrease costs related to administration and maintenance.

A different architectural approach to provide database load balancing and scale-out is by using a proxy server in between the application and the database. The proxy server is an intermediary that communicates the database network protocol. The application connects to the proxy server which forwards requests on its behalf to the database servers. The advantage of this architecture is that there are no changes to the application or the database, and the proxy server can perform load balancing, failure recovery, and result caching. There are several disadvantages including that the proxy server itself is a single point of failure which often requires the proxy server to be replicated. Since all queries and data flow through the proxy server, the proxy server must be a powerful machine capable of handling the load. From the implementation perspective, the proxy server must communicate the network protocol for the specific database (e.g. MySQL). To support multiple databases, the proxy server must implement each database's network protocol. This is not even possible due to licensing restrictions for some database products such as Oracle. Although the proxy server can cache results to reduce database load, there is less of a benefit for query response time as the data still must be sent over the network from the proxy server to the client application. In comparison (see Figure 1), the driver-based approach does not have a single point of failure as each application server runs independently, does not require a separate server or network hop for communication (which improves query response time), and is database network protocol independent as it operates using standard JDBC. There are commercial products implementing the proxy server approach including ScaleArc[1] and MaxScale for MySQL[2]. Supporting high-availability was also a goal of RemusDB [7].

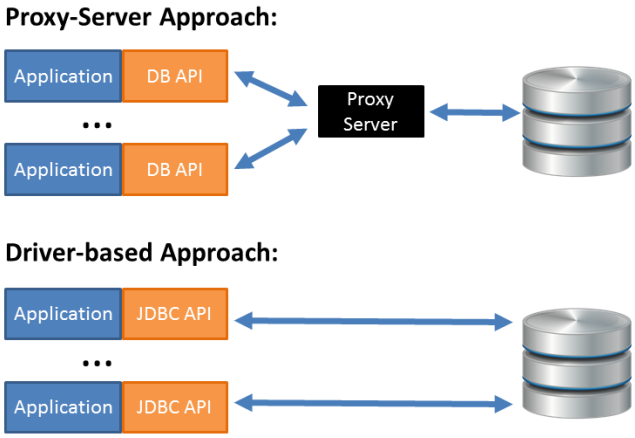

Figure 1: Proxy vs Driver Architecture

3. ARCHITECTURE

The Heimdall system shown in Figure 2 consists of two components: a Heimdall JDBC driver deployed with the application and a configuration server running on another machine. The configuration server is responsible for providing an interface for administrators to dynamically change the configuration of the system including the data sources accessible and rules for load balancing, caching, and failure handling. The configuration server provides real-time configuration updates to the Heimdall JDBC driver and collects statistics and performance metrics. On a periodic basis, the Heimdall JDBC driver sends a message to the configuration server providing its statistics and requesting configuration updates. The configuration server aggregates this information for analysis and provides a real-time dashboard displaying query performance. The Heimdall JDBC driver appears to the application like any other JDBC driver as it implements the JDBC API. Depending on its configuration, the Heimdall JDBC driver may re-route application requests to an in-memory cache, translate them to different queries, or load balance them across database servers all transparently to the application.

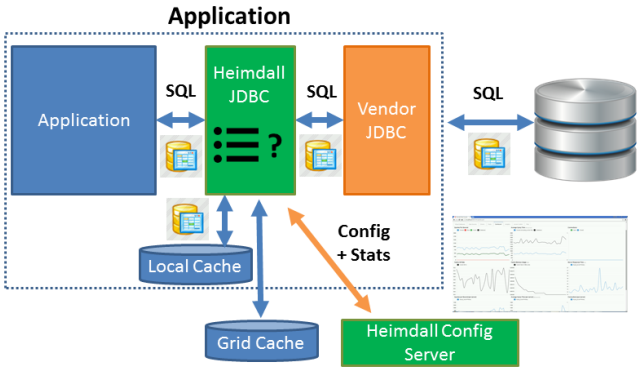

Figure 2: Heimdall Architecture

The benefit of this architecture is that it supports high scale-out as each application server has its own Heimdall JDBC driver that contains the intelligence. The driver can function without the configuration server once it has retrieved its initial configuration. Unlike a proxy implementation where all queries and data flow through the proxy server, the configuration server only provides configuration (control) information to the JDBC drivers which connect to the underlying databases directly. The Heimdall JDBC driver uses the underlying JDBC driver for the database. This allows it to support any database that has a JDBC driver. By not using a proxy, it removes the additional network latency and overhead when communicating with a proxy server. Since the

[1]http://scalearc.com
[2]https://mariadb.com/products/mariadb-maxscale

Table 1: Heimdall Rules

Rule	Description
Log	Collect SQL query statistics
Trace	Track data changes
Cache	Cache query result
Async	Execute update asynchronously
Forward	Send query to another server
Transform	Convert SQL into another form
Learn	Learn typical queries
Allow	Allow query pattern
Drop	Ignore query pattern

system is implemented as a JDBC driver, it can be deployed in an existing application with no code level changes in less than 15 minutes. The only change required is the JDBC URL connection string and driver. For example, an application using a MySQL database would change its JDBC URL from `jdbc:mysql://server/database` to `jdbc:heimdall://server/database` and the JDBC driver class would change from `com.mysql.jdbc.Driver` to `com.heimdalldata.HeimdallDriver`.

4. RULE-BASED EVALUATION

The novel contribution of the Heimdall system is its user-configurable rule-based engine that allows for configuration changes in real-time without application restart. Similar to network devices like load balancers and routers that inspect network packets and may re-route them based on a rule system, Heimdall inspects database traffic at the request level and may re-route the traffic based on user-specified rules. This is a powerful and flexible approach allowing the addition and removal of rules at any time. The rules supported by the Heimdall system are in Table 1.

Every rule has an associated regular expression that is used to match the SQL statement. When any SQL statement is sent to the driver for execution, it is matched against all rules in the system. If one or more rules match, then that behavior is performed. If no rules match, then the SQL statement is sent to the underlying database driver to execute as usual.

4.1 Collecting Statistics and Caching

The rules for logging and tracing allow a system administrator to generate application-level performance statistics. These statistics can be used to identify poorly performing queries. A logging rule records the query response time and time to retrieve the data by the application. This complements database-level statistics which provide the database performance information but do not see application and network overheads. The tracing rule performs more detailed logging, and the query result is analyzed and hashed using MD5. By examining multiple trace results, it is possible to see how frequently the results of a particular query are changing. The Heimdall cache prediction system uses this information to provide recommendations on what queries and data can be cached based on historical query and update patterns. Using these two rules and running an application for a short-time (hours or days), the system will provide a report of query performance and cache recommendations.

A user can select a query pattern to cache which generates a cache rule. A cache rule specifies the query pattern regular expression, the location of where to cache the query result (in-memory cache or grid cache), and a time-to-live (TTL) value for how long to cache before the result expires. When a query is submitted by the application, the Heimdall JDBC driver determines if it matches any of the cache rules. If it does and a query result is in the cache, it returns the result directly from the cache. If there is a cache rule for the query but no result in the cache, the result is retrieved from the database and inserted into the cache for the next request. Another optimization is the ability to execute insert, update, or deletes asynchronously using a background thread that does not block the main application.

4.2 Cache Implementations

Heimdall supports many different types of caching. The default is a local, in-memory cache using Hazelcast[3]. This cache is not synchronized across application servers. If cache consistency across application servers is required, then users may use grid caches such as distributed Hazelcast, Redis, memcached, or any JPA-compliant cache. There is an order of magnitude performance improvement when retrieving queries from cache rather than the database. Using Heimdall allows an application to take advantage of the scale-out and in-memory performance of grid caches without making any changes and still using the JDBC API. Serving requests from cache also reduces the load on the database server.

Cache entries are invalidated based on time-to-live (TTL). Heimdall will recommend TTL values based on its trace statistics collected. The Heimdall JDBC driver is also intelligent to automatically invalidate cache entries when SQL commands are executed that would trigger an invalidation. For example, assume a query result is cached that used table `R`. If the driver intercepts an INSERT, UPDATE, or DELETE on table `R` any cached ResultSet that used that table is automatically invalidated. This auto-invalidation allows a user to write cache rules that cache at the table level and have the system handle invalidations automatically.

4.3 Transformations and Migrations

A transform rule converts a query pattern into another query pattern. This can be used to fix SQL queries that are inefficient without changing code. It is also useful for database migration scenarios to translate one dialect of SQL to another (i.e. Oracle to MySQL).

4.4 Load Balancing and High Availability

For open source databases like MySQL and PostgreSQL that have limited support for automated fail-over, the Heimdall configuration server is capable of monitoring server performance, and automatically triggering a fail-over of the database server for the application. Load balancing and high availability is achieved by defining multiple database servers as part of the configuration. The Heimdall driver is replication aware allowing it to route queries to particular servers. For instance, a typical configuration is a primary-secondary configuration where the primary server is for writes and the

[3]https://hazelcast.com/

secondary server is a backup that may be accessible in read-only mode. Heimdall can be configured to perform a read-write split for this database configuration to allow queries to go to both servers for performance and writes only to the primary. Although database servers support replication and fail-over, one of the challenges to achieving high availability is that the application is not aware of database server changes. For example, a failure of a primary server and promotion of a secondary server to a primary may not be seamless from the application perspective. The Heimdall driver has visibility on database server changes and can isolate the application from those changes. This includes automatically reconnecting to an alternate server on server failure (without the application seeing the connection change) and holding queries waiting for a server fail-over to complete.

4.5 Security Firewall

With the increasing number of database and system hackers, the ability to determine attacks at the SQL and database level is important. Since Heimdall is intercepting all query traffic to the database, it is possible to use Heimdall as a security firewall to prevent SQL injection attacks and malicious SQL. The learn rule will learn the valid SQL traffic to the database and generate rules that white list (allow) SQL statements. Then, the user can define drop rules that will reject SQL statements that do not conform to the established patterns. The drop rule may throw an exception at the application level, log the intrusion, notify an administrator, or some combination of these actions. Overall, this provides a way for an application owner to have complete visibility of database traffic, a SQL injection protection mechanism, and a detailed access log for historical compliance audits.

5. PERFORMANCE RESULTS

The Heimdall system has been evaluated in different scenarios by several companies. The performance improvement for caching depends on the query traffic. Applications with heavy read traffic that is often repetitive such as e-commerce and media sites are especially suitable for performance optimization via caching. Highly transactional systems like reservation systems and banking data benefit less from caching. Applications that do not benefit from caching can still use load balancing and high availability.

As an experimental benchmark, Heimdall was tested with JIRA using PostgreSQL 9.4 in a primary-secondary configuration. JIRA is a well-known Java application that is representative of Java web-based applications built using technologies like JSP/Servlets and frameworks such as Spring Boot. JIRA has an application for testing performance that was used to generate a sample JIRA database and application traffic. The test setup followed the configuration recommended by JIRA[4] and used a database with 450,000 issues. A primary-secondary database server configuration was used to test load balancing and fail-over. Heimdall is able to detect a database failure, trigger a promotion of secondary to primary, and re-route application traffic seamlessly to the new primary database in less than 30 seconds.

[4]https://confluence.atlassian.com/enterprise/jira-data-center-performance-608960753.html

For the JIRA benchmark tests, cache rules were enabled for queries identified by the system for caching. Before caching, the database load in queries per second was about 375, and when caching rules are enabled, the database query load drops to about 225 (a reduction of 40%) with many queries now answered by the cache. This also enables the overall queries per second to increase about 10%. Before caching the average query time was about 0.68 ms, and with caching enabled, the average query time dropped by almost 50% to about 0.35 ms. On queries that were identified as cacheable, the query response time decreased by eight times. Even though only a percentage of queries were cacheable, this still had a major impact on the overall database performance of the application. For INSERTs executed asynchronously, the response time improvement is 20 times.

6. CONCLUSIONS AND FUTURE WORK

As new database technologies are developed and database applications migrate to cloud infrastructures, it is very important for the driver technology connecting the application to the database to evolve and improve. The Heimdall system is a next generation database driver that allows applications using JDBC to leverage in-memory caching, flexible deployments, and high availability without requiring changes to the application or database. Performance results on caching show an order of magnitude improvement for query results from cache rather than the database and a significant reduction on database load. The ability to automatically suggest queries to cache is a unique feature.

Future work will improve the cache invalidation to perform row-level rather than table-level invalidation, develop a completely in-memory option where the driver can act as a database without a backing store, and implement features of data virtualization and cross-database querying as done by UnityJDBC [6]. Users have also requested features to help with multi-tenancy and database partitioning.

7. REFERENCES

[1] JDBC. In Liu and Özsu [5], page 1580.

[2] ODBC. In Liu and Özsu [5], page 1947.

[3] E. Cecchet. C-JDBC: a Middleware Framework for Database Clustering. *IEEE Data Engineering Bulletin*, 27(2):19–26, 2004.

[4] R. Lawrence. Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB. In *2014 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 285–290, 2014.

[5] L. Liu and M. T. Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009.

[6] T. Mason and R. Lawrence. Dynamic Database Integration in a JDBC Driver. In *ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems, Miami, USA, May 25-28, 2005*, pages 326–333, 2005.

[7] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. RemusDB: Transparent High Availability for Database Systems. *PVLDB*, 4(11):738–748, 2011.