A Modular Approach to Distributed Caching

Martin Kostov and Kalinka Kaloyanova

Faculty of Mathematics and Informatics, University of Sofia St. Kliment Ohridski
5 James Bourchier Blvd., 1164, Sofia, Bulgaria
kkaloyanova@fmi.uni-sofia.bg

Abstract. It is essential today organizations to have fast and stable access to information stored in different sources. Last generation of in-memory database demonstrates much better productivity in data processing compared to classical relational database management system. In this paper, an approach for modular distributed caching is proposed. The approach is based on a modular layered architecture, which extends the primary relational based systems and will make it possible to increase the speed of queries processing. Some tests based on a prototype are performed and discussed.

Keywords: database, cashing, distributed cache, modular architecture.

1 Introduction

Despite the great achievements in modern database development, several issues related to data processing have not yet been adequately addressed. For example, the optimization of the processing data from large distributed data sets is still a main challenge for many applications.

In this paper, we propose an approach for processing large data sets located in different database management systems (DBMS) based on the distributed modular cache [1]. The proposed modular model could operate in the memory of different types of configurations. It could be implemented into existing applications or deployed as a standalone cluster of applications, which can take the data queries. Theoretically, the data can be pre-fetched from different sources, but our focus will be on the relational database management systems (RDBMS) due to their widespread use. Our main goal is to provide a way for improving the performance of existing systems in order to keep them usable when they have to transfer large amounts of data coming from different sources [17].

2 Related Works

At the time of the introduction of RDBMS, the number of processed requests was much smaller than now. Nowadays RDBMS are attempting to overcome this challenge [2]. All modern solutions have introduced asynchronous replication,

which reduces the level of consistency [3]. They also support memory tables that are completely stored in memory and queries do not need to access data on the hard disks.

Furthermore, many query optimization problems were addressed with the introduction of NoSQL solutions [8]. They try to respond to the queries faster with greater throughput. Such an example is Streaming SQL for Apache Kafka (KSQL) [18]. Another viable decision is RethinkDB [4]. It uses streaming approach [12], [13] - once a request is processed, the database is responsible to generate new responses for each change, which affects records matching the request conditions. In the following sections of this paper with our modular approach, we propose an extension of this decision.

3 Modular Model

In this section, we define a model that practically works as a lightweight DBMS. We will present the structure of the model (Fig.1). It is based on layers and we will discuss several use cases of the application of this caching model.

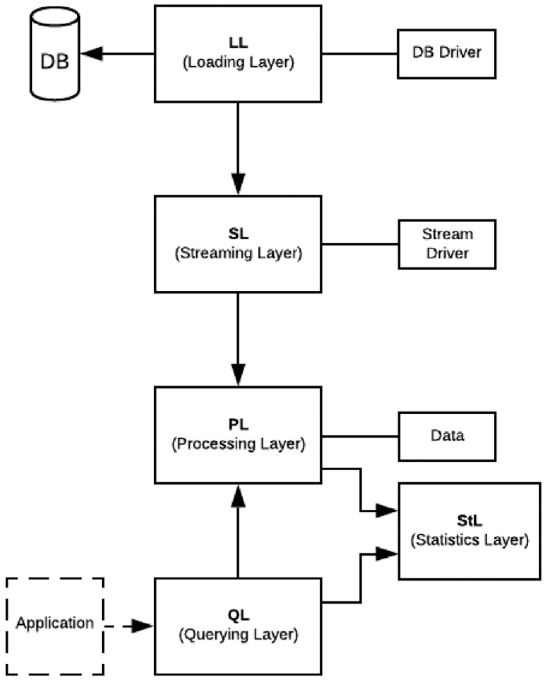

Fig 1. Fundamental layers used to build modular distributed cache.

The proposed model consists of five layers all of which are described in more details below.

Loading layer (LL) is the layer that loads the data from a data source. The load strategies are RDBMS independent [13], so they can work across different systems with different DB drivers responsible for creating DB specific queries. The strategies supported by the LL include:

- *Updatable Immutable* – in the initial load includes everything that will be processed. Afterwards we only load newly added rows. The modification of existing rows is prohibited.
- *Updatable* – it is the same as Updatable Immutable, with the exception that the modification of existing rows generates updates.
- *Full Updates* – we do not load updates. In a scheduled manner all data is loaded.

In terms of performance, the immutable way is the fastest, while full update shows the only case with worst performance.

Streaming layer (SL) is responsible for generating standard stream from the data retrieved by LL. We have the ability to use different streaming providers, from in-memory to third party like Kafka [5] or RabbitMQ [6]. This is the scalability layer of our distributed cache. With the introduction of this layer, we may support the work of two identical applications A and B. Application A can load its data from a RDBMS, and application B can load the very same data from a Steam. The behavior of these two applications is identical. With the stream, we can have easily geo-located applications, which load the data from a geo-distributed stream [8]. In such case, we do not need to distribute our centralized database to each location in order to have instant response times [11].

Processing layer (PL) - once we have all the changes as a stream, the PL is a direct binding to the defined SL. We are subscribing with some sort of condition (a query). In this layer, the data is stored in appropriate data structures. We have clustered indexes, which sort and store the data rows in a dictionary like structure based on their primary key values. We also provide standard non-clustered indexes where key value entry has a pointer to a clustered entry.

Statistics Layer (StL) - this layer is responsible for storing all information for the queries – the execution plan, how long a query was executing, etc. This layer will support asynchronous indexation, which means it will be able to create indexes in the background. In addition, using statistics indexes can be created on "hot" columns, which are responsible for slow queries. In this layer, we analyze the queries from the Querying layer and store statistics for them. These statistics are used in the processing layer to update the indexation.

Querying Layer (QL) is responsible for queries answering. This layer exposes SQL like language as a tool to the users, but it is not full implementation with some differences. The following shows an example query:

```
/query?select=*&from=MyData&where=Id=1&take=100          (1)
```

Select and ***from*** clauses are required, while ***where*** and ***take*** clauses are not mandatory. Join operations are also supported as they are standard. The layer is restful oriented and as such, order of parameters is irrelevant. In addition, it is possible to query statistics gathered by the SL automatic indexation on "hot" columns responsible for slow queries. Our prototype can work inside existing application, in this scenario Language Integrated Query (LINQ) [16] can be used in the application.

4 Typical Use Cases

Web application, web service, worker service, etc. usually read data from multiple different RDBMS. We exclude the trivial case where these RDBMS can be linked together if they are from the same provider. If the databases cannot be linked, the application must retrieve the data from all the different sources, after which it must produce the desired result. In this case, the developer must manually provide standard SQL operations like join, filter or group.

In some scenarios, the application needs to react to filtered data. Legacy applications tend to work with direct reads. There are already load balancers like *Nginx* [10], which help with the distribution of web applications, with their help one monolithic site can have hundreds of instances behind the same domain name. Unfortunately, RDBMS do not scale well in this situation.

In the case where the large amount of data cannot be loaded from all applications simultaneously, we can have dedicated application, which is exposing its Query Layer restfully. In this way, we can combine and use different combination of layers in different places.

In the next figures, we can see two different configurations. In Fig. 2a one module, which contains all the layers inside, is presented. In Fig 2b we have three modules. The difference between the two configurations is that DB1 and DB2 in the second one may be from different vendor, so different DB Drivers must be used. One of them is responsible for storing the data. The other two modules are responsible for loading and streaming the data.

These most trivial use cases could be resolved using the cache model.

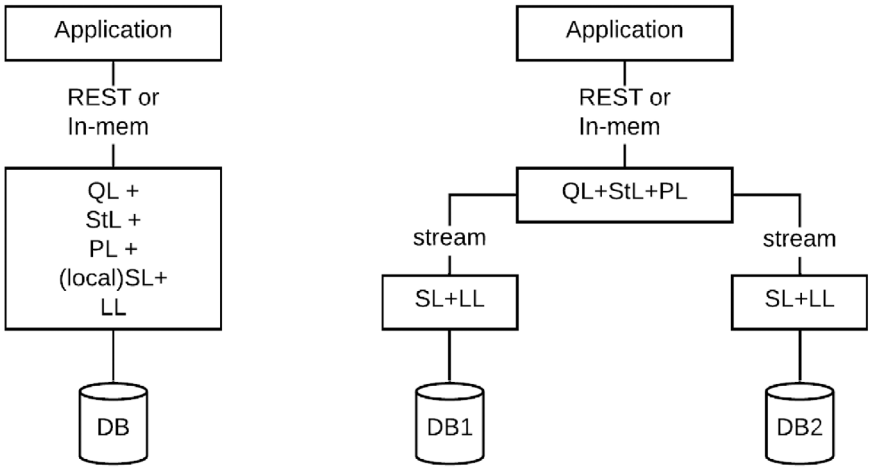

Fig. 2a. Single Module Configuration Fig. 2b. Multi Module Configuration

5 Results and Discussions

A prototype of the caching model has been developed. Several tests have been performed. The tests implement the case, are shown in Fig. 2a.

Because of the limited functionality implemented in the prototype, the focus of the tests is to provide information on how well the prototype of the caching layer is working in high load scenarios, and how fast it is responding in comparison with similar products. The obtained results are compared to several NoSQL and RDBMS decisions.

5.1. Testing platform

The prototype is based on .NET Core version 3.1 framework [14]. In the prototype we have implemented SL with Apache Kafka [5] driver for streaming.

- Testing tool: BenchmarkDotNet=v0.11.5
- OS: Windows 10.0.17134.885 (1803/April2018Update/Redstone4)
- CPU 3.60GHz 8 logical and 4 physical cores
- Memory: 2x8GB dual-channel 2400Mhz
- SDK: .NET Core SDK=3.1.201
- Our Prototype: 0.2.1
- Redis: 5.0.5
- SqlServer: 2017 Express Edition
- PostgreSql: 11.4

5.2. Dataset

The dataset, used for the tests is obtained from the IMDB movies database. The name of the set is ***title.basics.tsv.gz*** [9] and it contains information about films and their details. The information in the dataset includes:

- tconst (string) - alphanumeric unique identifier of the title
- titleType (string) – the format of the title
- primaryTitle (string) – the most popular title
- originalTitle (string) - original title, in the original language
- startYear (YYYY) – represents the release year of the title
- endYear (YYYY) – TV Series end year. '\N' for all other title types
- runtimeMinutes – primary runtime of the title, in minutes
- genres (string array) – includes up to three genres associated with the title

Since the dataset is updated daily, we must mention that our tests were provided over the version from 09 July 2019 when the total number of movies inside the dataset was 9,439,923 [9].

5.3. Results Summary

In Table 1, the results summary for getting random row by primary key is presented. The numbers show the amount of time required to obtain an item by its primary key. The keys are randomly generated numbers from 1 to 1 000 000, and this is repeated 1 000 000 times. The databases and the tested applications were deployed in the same virtual machine, so the network delay is minimal.

Table 1. Results summary

Cache	Mean	Error	StdDev	Median	Ratio	RatioSD
Prototype	680.2 us	16.61 us	48.98 us	680.6 us	1.00	0.00
MSIM	1,093.9 us	110.24 us	325.06 us	1,003.1 us	1.61	0.48
Redis	2,635.7 us	54.26 us	117.95 us	2,634.1 us	3.87	0.30
SqlServer	2,909.4 us	57.75 us	150.09 us	2,871.8 us	4.28	0.37
PostgreSql	4,979.1 us	152.02 us	428.78 us	4,901.0 ms	7.32	0.81

Mean: Arithmetic mean of all measurements
Error: Half of 99.9% confidence interval
StdDev: Standard deviation of all measurements
Median: Value separating the higher half of all measurements (50th percentile)
Ratio: Mean of the ratio distribution ([Current]/[Baseline])
RatioSD: Standard deviation of the ratio distribution ([Current]/[Baseline])

The prototype of our modular distributed cache is set to be the baseline for the ratio calculations. Several solutions are involved in the comparison: Redis – a general-purpose in-memory database, the Microsoft in-memory (MSIM) implementation [15], Redix and two relational DBMS – SQL Server and PostgreSQL.

The results show that the prototype model is very stable and fast. It is 61% faster than the closest solution, provided by MSIM and more than 3 times faster than Redix. The execution times of relational databases are highest, as expected.

5.4. Detailed Results

In Fig. 3, we can see the response histogram from the work of our prototype. The prototype was able to respond faster to the other contenders, even faster than application in-memory cache, because of better primary key usage. In general, the response time is stable, and the deviation is very small. In 1% of the responses, the time required by our prototype was less than 533.699 us. In addition, we have the stabilization in the interval between 628.364 us and 730.890 us. In this interval, we answered up to 75% of the requests. Moreover, the slowest 1% of the queries took more than 782.933 us.

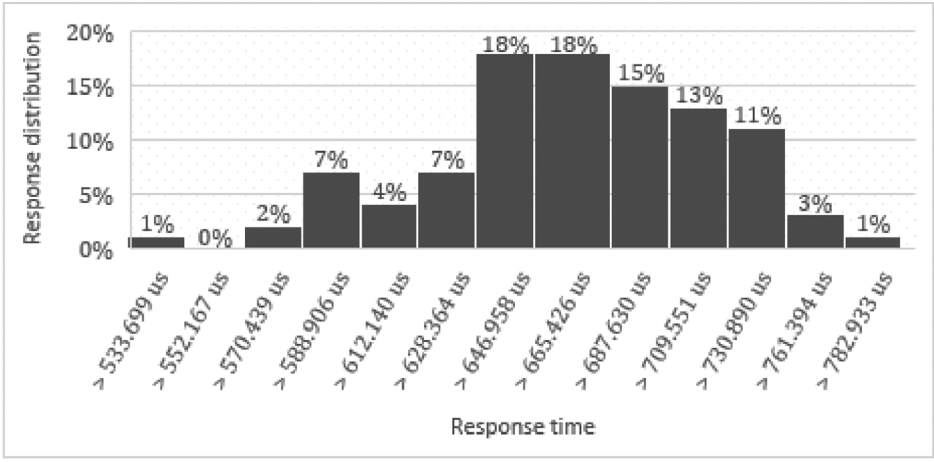

Fig. 3. Prototype Response Histogram

The primary difference between our solution and the one, implemented via in-memory cache, is that instead of general-purpose implementation which use GUID as a key, our solution uses a generic key, which in our movie dataset is of type int64 (long). In addition, this is the main reason for the difference in the response time. Redis is proven stable in its performance and we can see it in the histogram below.

As shown in Fig. 4 the response histogram from Redis cache implementation is similar. Redis had stable interval between 2.372ms and 2.836ms where 96% requests were. Only 4% were outside of the interval.

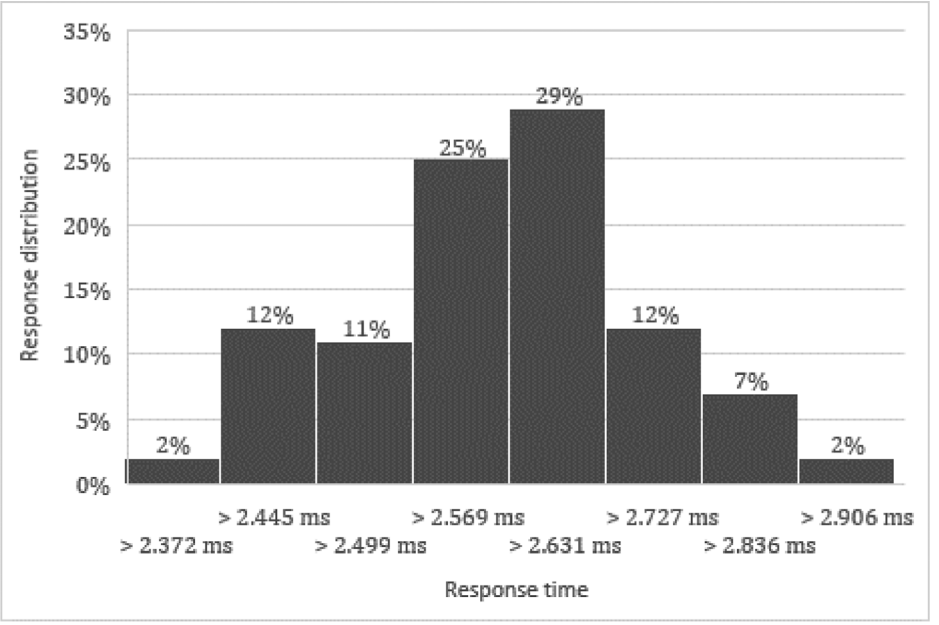

Fig. 4. Redis Response Histogram

6 Conclusions

In this paper, we propose a model for distributed processing of data retrieved from multiple different RDBMS. We are combining the data into standardized stream. To work effectively with this stream, we propose a specific toolset, which is RDBMS independent and this tool could be shared between different data providers.

Indexing data in memory reduces the response time for complex queries. Processing data streams from a constant flow in the background, enables us to have a stable solution with predictable and low latencies over time. Using predefined modular deployments in a cluster helps us to scale the number of streams reading applications very well, without a direct performance penalty on the primary RDBMS.

Having the stream of data provides unique opportunity to the users, because now they can use it with all available tools for the specific streaming provider.

Further testing may include comparison of our prototype with different streaming providers.

Acknowledgements

This research was supported by the project BG05M2OP001-1.001-0004 Universities for Science, Informatics and Technologies in the e-Society (UNITe) and the project "GloBIG: A Model of Integration of Cloud Framework for Hybrid Massive Parallelism and its Application for Analysis and Automated Semantic Enhancement of Big Heterogeneous Data Collections", contract DN02/9/2016.

References

1. Modular Distributed Caching, https://github.com/KostovMartin/Cache.Collections/tree/0.1.0
2. Dimitrov V, Data Structures in Initial Version of Relational Model of Data, In Proc. of Seventh International Conference Information Systems & Grid Technologies, Sofia, Bulgaria, 2013, pp:365-369
3. Werner Vogels. Eventually consistent. Communications of the ACM, January 2009 vol. 52, no. 1, Doi:10.1145/1435417.143543
4. RethinkDB. https://rethinkdb.com/docs/architecture
5. Kafka. https://kafka.apache.org/uses
6. RabbitMQ. https://www.rabbitmq.com/#features
7. Redis. https://redis.io/modules
8. Anil Pacaci, M. Tamer Özsu. Distribution-Aware Stream Partitioning for Distributed Stream Processing Systems. Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. June 2018. Article No 6. Pages 1–10. https://doi.org/10.1145/3206333.3206338
9. IMDb Dataset with information for titles. https://datasets.imdbws.com/title.basics.tsv.gz
10. NGINX. https://docs.nginx.com
11. Michael Armbrust, Tathagata, Joseph Torres. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. ACM. SIGMOD '18: Proceedings of the 2018 International Conference on Management of Data. May 2018. Pages 601–613. https://doi.org/10.1145/3183713.3190664
12. Javed M. Haseeb, Xiaoyi Lu, Dhabaleswar K. (DK) Panda. Characterization of Big Data Stream Processing Pipeline: A Case Study using Flink and Kafka, In the Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, 2017. pp. 1–10. https://doi.org/10.1145/3148055.3148068
13. Nori Anil K. Distributed caching platforms. Proceedings of the VLDB Endowment, Vol.3, Issue 1-2, 2010. https://doi.org/10.14778/1920841.1921062
14. BenchmarkDotNet. https://github.com/dotnet/BenchmarkDotNet
15. Distributed Memory Cache. https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed?view=aspnetcore-3.1#distributed-memory-cache
16. Language Integrated Query (LINQ) https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/
17. Kuppili Venkata, S., Musialy K., Sub-query Fragmentation for Query Analysis and Data Caching in the Distributed Environment, Preprints 2019, 2019100054
18. Introducing KSQL: Streaming SQL for Apache Kafka, https://www.confluent.io/blog/ksql-streaming-sql-for-apache-kafka/