University of Twente

Master Thesis

Low latency asynchronous database synchronization and data transformation using the replication log.

Author:
Vincent van Donselaar
vincent@van-donselaar.nl

Supervisors:
Dr. Ir. Maurice van Keulen
Dr. Ir. Djoerd Hiemstra
Ruben Heerdink MSc

August 14, 2015
Non-Confidential

Abstract

Analytics firm Distimo offers a web based product that allows mobile app developers to track the performance of their apps across all major app stores. The Distimo backend system uses web scraping techniques to retrieve the market data which is stored in the backend master database: the data warehouse (DWH). A batch-oriented program periodically synchronizes relevant data to the frontend database that feeds the customer-facing web interface.

The synchronization program poses limitations due to its batch-oriented design. The relevant metadata that must be calculated before and after each batch results in overhead and increased latency.

The goal of this research is to streamline the synchronization process by moving to a continuous, replication-like solution, combined with principles seen in the field of data warehousing. The binary transaction log of the master database is used to feed the synchronization program that is also responsible for implicit data transformations like aggregation and metadata generation. In contrast to traditional homogeneous database replication, this design allows synchronization across heterogeneous database schemas.

The prototype demonstrates that a composition of replication and data warehousing techniques can offer an adequate solution for robust and low latency data synchronization software.

Preface

This thesis is the result of my final project for the Computer Science master programme at the University of Twente. It took me quite some time to finish this project, perhaps partially because I accepted the offer of full-time employment at Distimo immediately after an internship of six months. Still being very satisfied with that decision, it turned out to be quite a challenge to finish an academic study while working for more than 40 hours a week with great contentment. My daily work was at times highly correlated with the topic of my thesis. For me it was never a question whether I was going to finish my research or not. The planning on the other hand certainly was a question, up until now.

My time at Distimo was a great experience and for me it was the definitive confirmation of my interest in computer science and software engineering. I do not regret my choice starting the natural sciences oriented Advanced Technology bachelor. But in hindsight I would have picked a CS bachelor instead. I never took the chance of following bachelor courses like Compiler Construction and Functional Programming. Fortunately I am able to look back at great highlights like Djoerd's BigData course including a trip to SARA, and every single database related course Maurice taught me. Not least I'm grateful for having both gentlemen as supervisors for this final project. Their dedication and patience was encouraging and pleasant.

At the time of writing the final words of this thesis I have moved on to a next step in my professional career outside of Distimo. Times change and they often do that in unexpected ways. Hopefully the memories of a great team of colleagues will not. My gratitude goes out to Ruben and Tom for supporting me in my research and for offering me a position as a professional software engineer. It has been a true pleasure working together.

Vincent van Donselaar

Acronyms

ACID Atomicity, Consistency, Isolation and Durability. 6, 24, 26, 29, 33, 34

API Application Programming Interface. 3, 5, 22, 27, 39

binlog Binary (transaction) log. 38

BLOB Binary Large Object. 39, 41

CDC change data capturing. 17, 19–22, 29, 30, 33

DDL Data Definition Language. 41, 43, 44

DML Data Manipulation Language. 37

DRY Don't repeat yourself. 6

DWH data warehouse. i, 1, 3–6, 9, 11, 14–17, 25, 33, 38, 44

ETL Extract, Transform and Load. 6, 8, 11, 16, 22, 33

GTID Global Transaction Identifier. 41

JDBC Java Database Connectivity. 23, 38–40

MITM Man-in-the-middle. 3

ORM Object-Relational Mapping. 13, 24–26

RDBMS Relational Database Management System. 18, 20, 26

UUID Universally Unique Identifier. 41, 42

WAL Write-ahead log. 36

ZLE Zero Latency Enterprise. 17

Contents

Chapter 1

Introduction

1.1 Problem statement

Mobile app analytics company Distimo is an organization that is heavily driven by large sets of relational data across multiple database systems. Keeping these systems in sync is a challenging task which demands continuous improvement. The next step is to advance the synchronization between structurally different databases in a low latency manner.

1.2 Distimo: the app store analytics company

Distimo was founded in 2009 with the ambition to provide transparency in the mobile app market. In its first years of existence the company offered custom reports with market insights on metrics like the total number of apps per platform and the estimated number of downloads and revenue for popular applications. A web application was created over the years to allow customers to do their own analysis based on the data gathered in the Distimo backend data warehouse (DWH).

With support for the Apple App Store, Google Play, Windows Phone, BlackBerry and Nokia, the product is unique in a sense that it allows a cross market overview of mobile app performance. The premium product AppIQ allows app developers to keep an eye on the competition by estimating downloads and revenue of nearly every popular app in the market. Apart from app developers ranging from individuals to large software companies, a vast number of investment firms likes to stay informed on the fortunes of the mobile app ecosystem. This made Distimo a major player in the fast growing app market. The definitive success was finally proven when Distimo got successfully acquired by competitor App Annie in May 2014 [Perez, 2014].

1.3 Data processing challenges

The ever increasing data volume continuously raises the bar for efficient data processing solutions. A well known and notorious problem commonly encountered with growing data sets are the limits of vertical scaling: buying faster hardware is a medium to short term solution because there are limits to what one single server can handle. Once this barrier has passed by introducing the inevitable 'horizontal' measures like

sharding and replication, a new domain of challenges unfolds. Data has to be kept consistent. Querying should still be fairly quick and there must be an adequate disaster recovery plan. For a data driven company like Distimo, this scenario happened in an early phase. It doesn't mean all problems have been settled already. The development of a data processing pipeline is an evolutionary process and the target of this research is to streamline the data flow by moving from a batch oriented design to a continuous replication-like approach. Appendix A describes the nature of Distimo's data and the setup of the database and data flow within the company. The new approach will eventually turn out to be an improvement in matter of latency across the database cluster, while preserving the great goods like consistency, easy maintenance, and existing business logic. Chapter 2 will identify the general problems related to the typical setup and scenarios for an organization like Distimo.

1.4 Heterogeneous database synchronization

The backend and frontend databases have to be kept in sync. Throughout the day new information gets added to the DWH by the scrapers, which in most cases must find its way to the frontend as well. As described in appendix A, both databases can be quite different. This means that regular replication is not a viable option to keep the systems in sync. Because of that, a dedicated synchronization process takes care of propagating the relevant changes to the frontend database on regular intervals. This process contains the domain logic that is required to guarantee this consistency across the databases. It can be seen as a mapping that projects the data from the backend database to the frontend database. This mapping involves transformation, aggregation and metadata generation and is therefore certainly not a trivial view on the DWH. The synchronizatoin program runs various Extract, Transform and Load (ETL) steps to fully update the frontend with the latest state dictated by the state of the DWH.

The primary focus of this research is to improve this very specific part of the infrastructure while keeping in place the complicated business logic that defines the mapping from the DWH to the frontend database.

1.5 Research goals

The synchronization of the internal database systems is a vital process for a data-driven company. Data latency plays an important role because customers make their decisions based on this data. The aim of this research is to streamline and improve the recurring ETL steps involved with typical database synchronization by shifting from a batch oriented approach[Inmon, 2005] to a low-latency, continuous data integration solution. Key factors of design are:

- Low latency in order to deliver up to date information to the customer. A Streaming setup is preferred over batch processing to cancel the costly overhead introduced by the setup of the batch process.
- Maintaining Atomicity, Consistency, Isolation and Durability (ACID) properties. Replication delay is allowed however. This means that replicas of the database are allowed to 'run behind' in time in the order of seconds while they are processing the already committed transactions on the master.
- Re-use of existing business rules and logic following the Don't repeat yourself (DRY) principle.

- Easy maintenance for operations team: a self healing system that can easily be restarted to continue where it had stopped, just like normal replication.

Apart from these architectural design goals, there are also functional requirements that should be met. These requirements are not a fundamental part of the design although they are likely to be applicable to most situations. Section 2.6 elaborates on the Distimo-specific considerations of the following generic functional requirements:

- Filtering: Selective data replication based on custom business logic.
- The possibility to do data transformations: One single source record could result in multiple target records and vice versa.
- Metadata processing, i.e. index and cache generation: the process produces metadata on the fly which is cheaper than using a separate process to maintain caches and indices.

1.6 Design validation

The design and prototype of the system will be assessed for production-grade quality and fitness. This assessment will be done by running the prototype in parallel with the existing synchronization code while writing to a separate database table. The contents of this table are expected to be the same as the production table. Each of the goals from the previous section will be evaluated to see if the criteria were met.

1.7 Contents

The next chapter gives a detailed insight in the current situation at Distimo's infrastructure and database setup. Current and future bottlenecks will be highlighted and requirements for an improved synchronization system will take shape. Driven by this agenda, chapter 3 offers an overview of relevant literature on useful approaches and applicable architectures. Chapter 4 introduces a prototype synchronization framework addressing the intended goals. Verification of the proclaimed features and a performance improvement analysis is addressed in chapter 5, on which chapter 6 draws both conclusions and suggestions for further investigations and improvements.

Chapter 2

Data synchronization in depth

2.1 Characteristics of a multi-purpose database cluster

Having two databases with different schemas offers the advantage of being able to optimize the database for a specific purpose. This ideally results in a backend database that stores data efficiently and allows data to be appended easily. Other databases are more likely to be optimized for long running queries or real time user interfaces. The middleware that is responsible for the synchronization between the databases is a very important part of such a typical setup: a failure of the middleware results in the target database(s) being out-of-sync with the source database. Apart from the availability requirements, the middleware is also the place where data transformations usually take place. Instead of just copying data from A to B, these middleware programs become complex ETL tools.

This chapter walks through the principles of data synchronization between heterogeneous databases. The next subsection considers the implications of fitting object entities to a relational database while being able to address issues like merging with existing data and the prevention of deadlocks. Section 2.3 focuses on the heterogeneity and how (and when) to transform data while synchronizing. Following that, a list of limitations of the 'sync state' sync will be enumerated on which a list of requirements for improvement will follow in section 2.6.

2.2 The structure of the data types in depth

Confidential

2.3 Synchronization across heterogeneous structures

Confidential

2.4 The architecture of the 'sync state' synchronization

Confidential

2.5 Bottlenecks of sync state based synchronization

Confidential

2.6 Requirements

The analysis of the baseline 'sync state' situation and the effort to describe the individual bottlenecks of the design have resulted in the functional requirements of the prototype as already described in section 1.5. This section adds some background to these requirements that will play a role in the design of the prototype.

2.6.1 Low latency by streaming synchronization

An evident, and presumably the most important requirement, is the desire to stream the changes to the target as soon they are recorded in the source DWH rather than polling for changes. Polling will always introduce overhead and delay, while a continuous stream will decrease the latency when implemented correctly.

2.6.2 Full recoverability from the source database

The new sync framework must be able to reconstruct a consistent target state regardless of the (possibly faulty) changes that were applied earlier. Example situations are cases of a software bug, an outage, or a manual intervention where the synchronization has to be rerun. In other words, the sync should still have one modus operandi in which it can idempotently reconstruct arbitrary parts of the frontend database. This immediately contradicts the previous requirement that is solely based on changes from a stream. Therefore it will be likely to keep the sync state table in place in case the streaming solution has failed.

2.6.3 Minimized I/O operations

A heavily used source DWH may easily processes several thousand queries per second. It is important to keep the DWH available as much as possible, without being counteracted by other transactions keeping locks on tables or parts of tables. This must be an implicit design requirement.

2.6.4 Event detection

Event detection is a special case of a transformation. An event can be described as a transformation that needs the context of a record that is not part of the transaction. A typical example is when the price of a product changes. In order to detect the price change, yesterday's price has to be known in order to compare that price with the product's price of today. Transformations that need such a sliding window with information are denoted as 'Events' and need to be supported.

2.6.5 Consolidation of asynchronous post-processing

All asynchronous post-processing must happen within the synchronization process. This is merely a result of the requirement to deliver low latency data.

2.6.6 Relaxed ACID transactions

All data is originally inserted in the source database by atomic transactions at a sufficiently strict isolation level to guarantee consistency. Such a transaction is, by definition, a consistent delta within scope of the source database. This same logic delta must be applied to the target database eventually, but it must conform to a different schema. It is not possible to implement a two phase commit across both databases because of performance reasons. Nonetheless the transactions must be replicated to the target database correctly. This should be done by adopting a relaxed form of ACID transactions that is similar to normal (MySQL) replication: the transactions are guaranteed to be committed to the target, albeit by adopting an 'eventual consistent' guarantee.

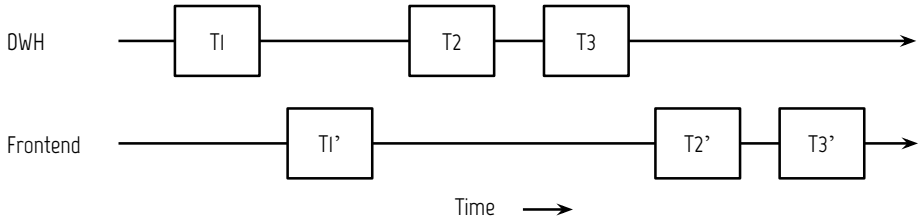

Figure 2.1: Transactions applied on the source DWH must eventually be committed to the frontend target as well.

2.6.7 Summary

The synchronization of sets of object graphs across structurally different databases often results in deadlocks when not properly optimized. Deadlocks can be avoided by reordering entities in-memory. The in-memory pre-processing of the data allows easy event detection and post-processing of data that is being synchronized. These extra data manipulations are likely to come with the cost of sacrificing at least one of the ACID properties.

Chapter 3

Literature overview

3.1 Introduction

The aim of this chapter is to find answers and adequate techniques required to realize the requirements stated in section 2.6. Literature considered relevant is expected to be found in the field of data warehousing which aims at synchronization and refreshing of data warehouse cubes. The traditional data warehouse paradigm differs a bit from the situation at Distimo because the Distimo DWH is the source of the ETL process rather than the target. This is primarily a naming issue; one should consider the frontend database to be a data warehouse as well that uses an ETL process (i.e. the synchronization program) to refresh its data.

3.2 Data warehousing

Data warehousing techniques have become crucial in enterprise decision making. The architecture of Distimo's infrastructure shows similarities with that of a traditional enterprise data warehouse, although aspects differ on certain fields. At Distimo, the 'data warehouse' is seen as the database containing the original, raw data as it was harvested by the data jobs. The aggregation step (in DWH terminology known as 'building the cube') stores its data in the frontend database. The web application known as Distimo App Analytics acts upon this aggregated data, directly serving customers in their data demands. Such a live queryable data warehouse exposed to end-users is often denominated as a 'data mart' [Inmon, 2005]. A data mart is a subset of a full-blown data warehouse, optimized for a specific company department or in the case of Distimo, for a specific customer.

3.3 Zero-latency Data Warehousing

In the beginning of the data warehousing era the primary concern was coping with - according to today's standards - limited resources. Data warehouse cubes in the order of terabytes were rebuilt in batches once per week or even once per month [Chaudhuri and Dayal, 1997]. As a consequence, queries on such DWH were not real-time. Modern business is however more likely to be interested in up-to-the-minute information and the technique of today makes it possible. The architecture presented by Bruckner et al.

[2002] is a proposition to accomplish continuous data integration using Java based middleware. Key in their work is real-time acquirement of data from various sources, the possibility to directly act and make decisions while processing, and maintaining high availability. More elaborate research by Nguyen and Tjoa [2006] proposes the concepts of a Zero-latency data warehouse (ZLDWH), presumably inspired by the Zero Latency Enterprise (ZLE) [Schulte, 1998]. Different stages of DWH evolutions are highlighted: traditional warehouses used for reporting (put together by pre defined queries) evolved in sources for analysis. An increase in analytical model construction resulted in the DWH becoming a source for prediction. Continuous updates of the DWH made it possible to react on operational events, which could be used to adapt organizations. The final state is depicted as 'Automate and Control' which covers a continuous feedback loop of performance metrics.

3.4 Change data capture

Incremental data synchronization includes a fundamental step known as change data capturing (CDC). This step entails determining the delta between the state of the source and the target data set. This delta is used to update the target, which results in both sites being identical. In the case of two-way synchronization this can be a very hard task, which could result in conflicts when records on different nodes become inconsistent. In situations with one-way synchronization this problem diminishes since the state of the master is always leading.

Change data capture can be achieved using several techniques. Globally, the following options are worth considering.

Metadata storage.

Metadata storage involves storing additional information describing the current states of the data to be synchronized. Several solutions are possible. In case of one-way synchronization, which is often the case in DWH techniques, annotating data with a timestamp is often sufficient. Complex synchronization involving multiple sites quickly requires dedicated storage solutions for these metadata. Besides a timestamp indicating last synchronization, version numbering and so called 'tomb stones' indicate which data was changed or deleted. Regardless of the metadata format chosen there will always be some overhead in both maintaining and querying this metadata. A number of aspects is highlighted and pointed out by Chen et al. [2010].

Triggers.

Database triggers offer a variety of possibilities to track changes of a database. A common practice in data warehousing is to automatically update aggregation tables upon an update or insert on the raw data. The penalty for doing this is a delay in write operations. Using triggering and scheduling algorithms, this effect can partially be circumvented [Shi, Bao, Leng, and Yu, 2009; Song, Bao, and Shi, 2010].

Event programming.

Instead of capturing changes after the events of interest happened, the CDC category known as event driven data capture is based on a different approach. Often a database proxy or a framework

hook is used to generate events. Eccles et al. [2010] claim this is the only approach capable of truly capturing real-time data changes. Event programming became popular after the principle was formally described by Fowler [2005] on his weblog under the title of Event Sourcing. Fowler describes the principle of identifying and tracking of changes of an application's state. The application developer essentially creates a transactional log, which correlates with the log scanning technique that assumes an already existing log.

Log scanning.

Log scanning involves analysis of database transaction logs. These logs are practically the first derivative of the state of the database over time. The framework proposed by Shi et al. [2008] exploits this property for doing change data capturing. The advantages of log scanning are very promising according to this research. Transaction logs are a very reliable source of information, since they unambiguously represent what happened to the state of the database. Non-deterministic queries are annotated in such a way that they become deterministic [Cecchet et al., 2008]; usually by storing the final result rather than the original query.

3.5 Change data capture

Database replication is a technique which is often applied to improve availability and performance of a Relational Database Management System (RDBMS), but is primarily meant to act as a fail-over solution. The term 'replication' involves homogeneous one-way synchronization most of the time, which is relatively easy to set up and maintain. Most popular RDBMSs support this type of replication. More complex cases of replication, like multi-master replication are less common and are relatively complex because the consistency between replica's is harder to guarantee [Wong et al., 2009]. Another special case of replication is heterogeneous replication, which involves data stores of different types or brands. The idea of heterogeneous replication is not new [Wang and Chiao, 1994]. Connecting different data stores often involves data transformation and conversion. A combination of both worlds is worth considering: a one-way replication between two MySQL databases having a heterogeneous structure.

3.6 Hybrid sharding and replication

A common practice in the Big Data domain is sharding across multiple commodity type servers [Ghemawat et al., 2003]. Shards are often made redundant to prevent data loss and to balance the load. This design does not appear to be very well suited for ad-hoc querying because of the lack of data locality and the need to sort the data while employing MapReduce for data processing [Dean and Ghemawat, 2008]. Although projects like Apache Spark [Zaharia et al., 2010] offer impressive results in the field of interactive querying of large data sets, there is still a long way to go before real-time applications can be built on top of it. Meanwhile solutions offering a hybrid approach of both sharding and replication start to look promising [Dhamane et al., 2014], although being often heavily inspired and built upon existing distributed database technology like MySQL Cluster or C-JDBC [Cecchet et al., 2008].

3.7 Literature reflection

The general consensus of the literature is that a refresh interval between thirty minutes and one day using a batched synchronization cannot be considered a low-latency approach [Bruckner et al., 2002; Nguyen and Tjoa, 2006]. Even a drastic increase in the synchronization frequency will not result in a system that can be considered real-time. It only increases the amount of overhead involved with polling for change sets. The CDC method on which the current solution is based, is metadata storage (i.e. the so called 'sync states'). Of the types of CDC addressed in previous section, this one is least suited for low-latency applications. Based on findings in this chapter, table 3.1 summarizes the pros and cons of each CDC type. Using table 3.1 as a guidance to determine effective CDC mechanisms, the following conclusions can be drawn:

	Pro	Con
Metadata storage	Robust. Allows full state recovery.	Inappropriate for zero-latency application. Introduces overhead (database tables).
Triggers	Abstraction at database level. Real-time.	Inflexible programming environment. Inefficient due to excessive locking. Vendor specificity undermines robustness. Recursive triggers are hard to predict.
Event programming	Very flexible.	Increases programmatic coupling.
Log scanning	No impact on database I/O. Both real-time and historical data processing.	Vendor specific. Format is prone to structural changes over time.
Replication	Mature and vendor supported.	Only supports homogeneous data structures.

Table 3.1: Pros and cons of different CDC approaches

- The CDC mechanism that is based on metadata storage ('sync states') is an imperative way of tracking changes similar to the situation described in section 2.4. Being thoroughly tested, a synchronization job that acts upon sync states operates very reliable, but usually quite slow. The metadata storage is the most reliable among the CDC methods mentioned: it offers the possibility to rebuild a consistent state between the databases at any time. Other CDC methods (except log scanning) only act upon live changes. Reliable recovery from inconsistencies is important, which makes metadata storage indispensable.

- Database triggers are far from ideal because of the limitations of the environment. Besides the confined programming abilities, the use of database triggers will more likely introduce more locking issues than it will ultimately be able to solve. The MySQL documentation states that "If you lock a

table explicitly with **LOCK TABLES**, any tables used in triggers are also locked implicitly[1]". Triggers acting over multiple database servers are also not worth considering.

- Event programming looks promising, although it increases programmatic coupling, contradicting separation of concerns of the individual web scrapers. The event processing must not introduce any additional delay. It is more likely to encapsulate other CDC methods, extending them with additional logic. This spares the scraping jobs from additional responsibilities while keeping flexibility.
- Log scanning offers both online (real-time) and offline (historical) analysis of changed data. This technique is often incorporated for replication (see next section). The CDC on the log file can be executed using a separated and isolated thread, running at its own pace. Log scanning is however a bit more complex than other solutions. The possibilities are limited by the vendor specific format of the log file, which usually contain the bare minimum of data required for crash recovery and/or replication. Appendix B.5 offers an overview of important aspects of the MySQL transaction log. Like the metadata storage method, log scanning allows historical change capturing, but only as long the log files are maintained. This makes log scanning a very versatile approach as long the log files aren't deleted after a certain period.
- Database replication embraces the principle of a transaction log that tracks changes on the master database replica over time. The log will eventually be shipped to one or more replication slaves. Shipment of the log might happen in chunks or by a continuous stream of data and is eventually replayed on the replicas. Appendix B.1 describes the internal operation a replication log specific for MySQL. Other replication-capable RDBMSs have implemented log shipping in different ways, but the principle of replication is always more or less the same. A major restriction that can be identified across all popular databases is the lack of transformation capabilities and expressiveness. Either all data will be replicated or nothing at all. The flexibility of custom business logic that would otherwise be available in an application layer is absent.

Putting together these conclusions, there is no silver bullet among the evaluated CDC methods. The metadata storage method stands out in particular because it is the only mechanism that is able to fully recover an inconsistent state, because the metadata is always available regardless of the age of the data. Real-time approaches do not allow historical state recovery, except for the log scanning method which is limited by the retention of the log. The metadata storage is however exceptionally unsuitable for near real-time applications. In order to benefit from both robustness and low latency, a hybrid solution would be appropriate. This solution aims to combine techniques to exploit advantages of various techniques, while minimizing the limitations.

[1] `http://dev.mysql.com/doc/refman/5.5/en/lock-tables-and-triggers.html`

Chapter 4

Introducing the log sync framework

4.1 Introduction of the log sync framework design

The log based synchronization framework presented here is a design that leverages multiple CDC techniques that were mentioned in the previous chapter. The design is a hybrid of metadata storage and a replication technique based on log scanning and event programming. The reason for choosing a hybrid approach is because it is not possible to adopt one single method that fulfills all requirements. The metadata oriented approach is the only approach that is able to recover a consistent state reliably. This capability comes with the price of increased latency and overhead. Continuous log scanning overcomes this limitation and allows the framework to operate like an asynchronous database replicator. Unlike traditional replication, the principles of event programming allows selective synchronization and immediate transformations of the data. This is usually not possible with normal replication which is limited to homogeneous database structures.

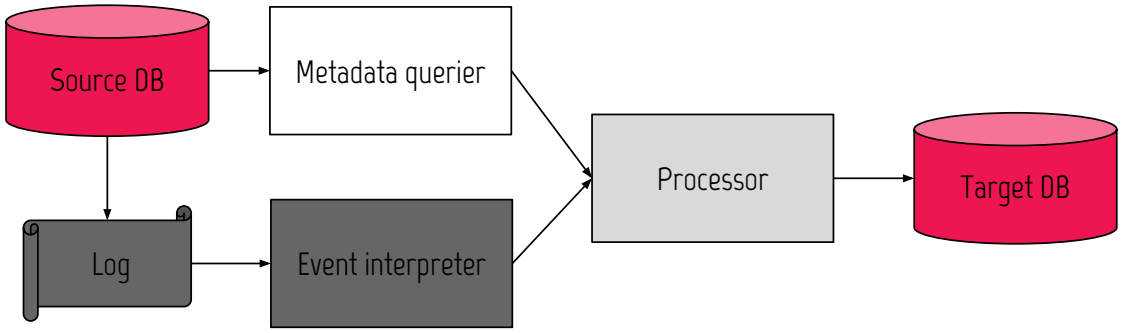

Figure 4.1: A hybrid design both supporting metadata synchronization and log scanning + event programming.

Figure 4.1 displays the two data streams that are involved with a hybrid solution. This synchronization framework has two modes of operation: one continuous synchronization stream originating from the log, and one ad-hoc state oriented synchronization. These operations will never run within the same runtime environment but it is possible to run multiple concurrent instances of a different kind. During normal daily operation only the log scanner part is likely to operate continuously. The metadata synchronization acts

solely as a backup for situations where the log is not able to offer conclusive information to the sync framework. This is likely to happen in situations of manual intervention: schema changes, data correction, and disaster recovery are typical examples.

The processor component contains all generic business logic that is involved with the synchronization. It must be capable of processing data from both CDC inputs and it should therefore expose a well designed interface. The actual benefit of this component is to have one single point containing all business logic rather than having multiple independent data streams.

4.2 Process description

This section describes the log scanning data flow from fig. 4.1 in more detail. The whole process from source to target can be seen as a sequence of ETL steps. All of these steps can be explained as a serial process: there is no intermediate storage involved except for buffering. Note however that change sets acquired from the log are processed on a per-transaction basis, which could contain multiple records at once. The end of this section will summarize the high level principle.

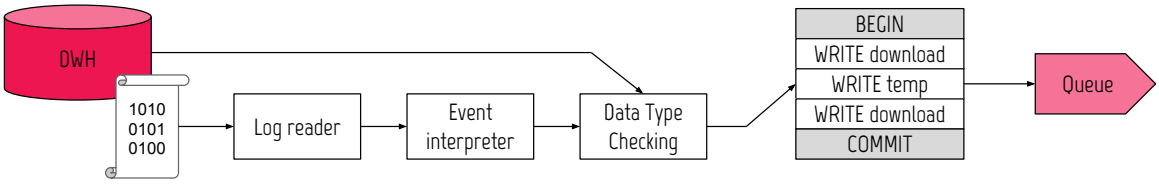

Figure 4.2: Steps 1 - 4: Log scanning, event checking and queuing of a transaction containing two write operations of 'download' entities and one irrelevant write operation to a temporary table.

4.2.1 Part I: The log scanner ('Extract' steps)

Step 1: Log reading

The begin of the process is shown on the left side of fig. 4.2. The synchronization job starts by specifying a log identifier and a position within that log. The server will start to transfer the log to the client where it will be processed by the next step.

There are various ways to ship a database log. MySQL allows a slave process to connect with the master database server. Such a connection can be established over MySQL's regular network interface that is also used for regular clients. Reading the log file directly from disk is also an option, although this is probably more error prone because the log files can be stored at different locations. There are various Application Programming Interface (API)s available for reading a MySQL binary log. An overview is given in appendix B.2. Other database systems that were not considered offer similar solutions.

Step 2: Log event interpretation.

The log contains a chain of events. Each event describes what happened on the master at a certain time in history. An overview of relevant types of events is given in appendix B.3 and is specific to

MySQL. The most important events are those describing changes of records. Each change data event contains the state of a particular record before and after the change. The log sync interprets these events and constructs a serialized object in runtime memory containing all relevant information regarding the change. This includes the time of the event, the name of the database schema and table, and the before/after states of the record. The change data events within one transaction are preceded by a transaction **BEGIN** event and followed by a transaction **COMMIT** event. All events of one transaction are grouped together as such.

Step 3: Data type checking.

The validation step checks whether the data in the serialized object from the previous step complies with the expected data structure of the database schema. This is done to make sure that the mapping is correct in order to prevent unexpected behavior to happen. After verifying mutual consistency between event and database structure, this step converts the database's data types to the application's native data types, e.g. **VARCHAR**s to strings, **INT**s to int or long, etc. Appendix B.4 describes this process in detail for the case of MySQL in combination with Java Database Connectivity (JDBC).

Step 4: Transaction identification & queuing.

The type checked events from the previous step are grouped per transaction. This is done to identify logical units of work to process. The scope of a transaction is an obvious choice because it represents one atomic and consistent delta. For each **BEGIN** event, a new list is created and filled with data change events that follow. As soon the **COMMIT** message occurs, the list is published to an internal queue. This queue acts as the primary buffer of the system. After publishing the list of events to the queue, the log scanner continues to interpret the next transaction from the log. The scanner only blocks if the queue is saturated or when the log's tail is reached. Scanning resumes as soon the queue accepts new input, or when a new event is written to the log. The use of an internal queue decouples the log scanner from the rest of the process to prevent it from blocking on log I/O.

4.2.2 Part II: Event programming

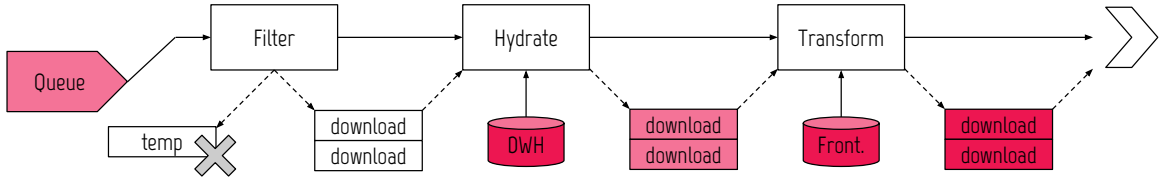

Figure 4.3: Steps 5 - 7: Filtering, Hydrating and Resolving

Step 5: Dequeuing & Filtering.

The processing tier continuously polls the internal queue for transactions to process, serialized as lists of database changes. Each transaction that was recorded in the log is analyzed for changes, for every single record that was inserted, deleted or updated. Not all of them are relevant for synchronization. A simple filter matches events solely on database and table name. This reduces the load on the

process in the next two steps, which are certainly the most resource intensive. This step shifts the process from typical replication to the principles of event programming.

Step 6: Resolving to original source entities & Hydration.

This step is responsible for reconstruction of the original entities as Object-Relational Mapping (ORM) objects in runtime like they were originally inserted on the master. Based on the table name included in each event, the corresponding schema can be determined. By using the ORM the other way around (i.e. bottom-up), a runtime object can be filled with data from the database event. Filling an empty ORM object with data is known as 'hydrating' the object. In some cases the event does not contain enough information to hydrate all the object fields that are mandatory. In such cases an extra **SELECT** query will be necessary. The primary key is always available from the event, so the query will likely be efficient. Nonetheless this violates the ACID principles which makes this an edge-case that developers want to avoid. Section 4.3.3 will describe this limitation in more detail.

4.2.3 Part III: The Processor ('Transform' and 'Load' steps)

Step 7: Transforming to target entities.

The target database uses its own ORM definitions specific to its domain model. The ORM entities from the source database will be transformed to their respective target entities. The source and target entity structures may differ. Some target entities contain extra fields while others lack fields that do exist in the source definition. It is the responsibility of the source entity to do its own transformation correctly. The transformation is done as follows. **a**) The source object is detached from its database session **b**) the object is forced transform itself to a target entity **c**) the newly created entity is attached to the target database and optionally further hydrated.After the transformation the in-memory ORM objects are consistent with the target database schema and could be inserted immediately. There is still some post-processing to do in the next step however.

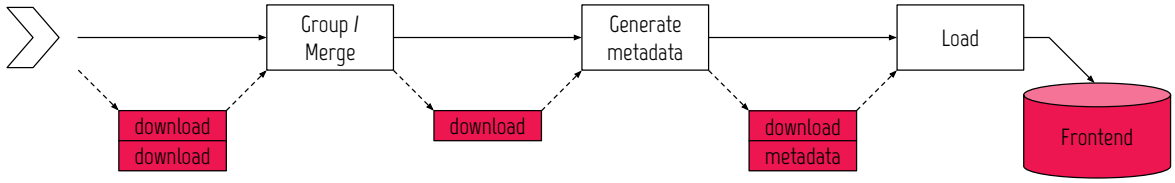

Figure 4.4: Steps 8 - 10: Merging and grouping, generation of availability metadata and loading.

Step 8: Post processing: merging & grouping.

Some entities might require post processing. Note that his is solely business logic and is not a fundamental step in the log-sync process, but nonetheless important to address. The following post-processing issues can be distinguished:

- **Merging** The transformation done in the previous step assumes that one single entity from the source maps on exactly one entity in the target database. This is not always the case however.

Some entities for example may span multiple records in the source database but not in the target. Such entities will be merged according to entity-dependent business logic.

- **Grouping** This is similar to merging but with a different discriminator. It could be the case that an entity has to be merged with an entity that was part of another transaction that happened earlier. This can only be done by querying the database for data which was already inserted.

Step 9: Post processing: Cache maintenance

The last processing step before the data gets inserted to the target database allows the framework to trigger additional, possibly out of band processes. The data itself is not manipulated anymore, that was already done in the previous step. This step is meant to do cache maintenance (warming up, invalidation, creation) and index creation. There is a good reason to incorporate this maintenance work in the sync process: All relevant data is in memory now which eases the processes that are likely to be involved here.

Step 10: Load data to the target database.

The last step in the process involves persisting all entities to the target database. This is done by triggering a 'persist' method on all ORM objects in memory. The ORM library will take care of the rest.

4.2.4 Recap: A summary of the log sync process

The multiple steps that were described in the previous section can be summarized on a high level as follows:

- A process monitors the log of the DWH and filters the relevant data that must be synchronized to the target database.
- The filtered data is used to reconstruct the original ORM objects. Possibly missing data is added by additional querying.
- These reified objects go through various business processes for transformation, aggregation and metadata generation.
- After these transformations, the objects are persisted to the database.

4.3 Limitations and system boundaries

4.3.1 Availability constraints

The log sync acts similar to a replication slave and is allowed to run behind in time on the master's operations. This latency is an important variable in the freshness of the data in the target database. The latency itself is dictated by both the processing power of the node that runs the synchronization process, as well by the number of write events per unit time written to the log. In case the synchronization process fails or needs to be stopped, operation can be resumed later. As long as the logs are retained and the position is known, the synchronization will start to catch up with the tail of the log and will keep reading along with the current transaction events happening.

4.3.2 Bootstrapping

The log sync can be started at any moment for reading from the current tail of the log. It is however important to realize that it will only monitor **changes** applied to the master database starting from that particular moment. Before the process starts, the states between both databases should already be synchronized to guarantee the data to be consistent in the future. In case the source and target are inconsistent, there are two possible options.

1. Use a the metadata storage to achieve consistency between both sides. Then start the log sync from the tail of the log.

2. Start the log sync, pointing to the last log position on which the source and target were known to be consistent.

The latter method is a common approach for crash recovery applied by most RDBMSs [Kifer et al., 2005]. Problems could occur if modifications to the database were applied afterwards. This will be discussed in the next section.

4.3.3 Log limitations

Transaction logs contain as little data as possible. This is an understandable attempt to keep the log files a bit manageable because they can grow quite fast. Appendix B gives an overview of the MySQL log format in more detail. The flip side of a minimalist log is that the log scanner needs to do a huge effort to identify and reconstruct the original objects in the application layer because the schema is implicit and not part of the log entries. After all, a normal replication slave has this schema already so there is no need to ship this information. The sync program solves this difficulty by matching the log entries with its ORM definitions which are effectively another representation of the very same schema. This imposes no significant problems so far. Sometimes however the reconstructed objects are not enough for the synchronization program to do all the complicated transformations. This is best expressed by an example: To calculate the daily revenue of an app, the sync needs to know the currency of the app's country. The currency was however not part of the original transaction because it is a static table and it is therefore not included in the log. The sync has to do a **SELECT** query on the database to retrieve this currency information. This violates the ACID properties of the system. The **SELECT** statement queries the most recent state of the database rather than the state at the time of the original transaction. This is similar to a 'non-repeatable read'. Fortunately this issue can be mitigated by carefully looking at the hand-written transformations. In case of the missing currency the ACID violation was allowed because currencies are never altered programmatically. A change of the currency table involves human interaction and the human in question should take care of the consequences for the sync.

4.3.4 Future compatibility

The log is a vendor specific format and the implementation of the log scanner heavily depends on this. Although every decent database embraces the same principle for either disaster recovery or replication, a migration to a different RDBMS will not be trivial. Even across versions of the same database there can be

incompatibilities. Appendix B.5 highlights some of these changes across various versions of MySQL that should be taken into account. The authors are not the ones to blame in this case. The format of MySQL's binary log was never meant to be used for other things than replication. As such, there is no detailed documentation on the precise binary format of the log. The conclusion is that log scanning will never be as easy as writing an interface to an API or using a standard like SQL. Fortunately the source code of the MySQL binlog contains quite some helpful comments[1], so a persistent person will find its way eventually. There is no guarantee that other vendors offer similar documentation regarding their log format.

4.4 Conclusions

The hybrid synchronization framework that was presented in this chapter combines log scanning, replication and event programming in one streaming synchronization technique. Apart from that technique, it remains compatible with ad-hoc database synchronization based on separate metadata storage. The overall design offers a generic and flexible synchronization solution although an actual implementation will become database vendor specific due to the low level of operation.

[1] https://github.com/mysql/mysql-server/tree/5.7/libbinlogevents/include

Chapter 5

Design analysis

The goals of this chapter are two-fold. The first part in section 5.1 will qualitatively analyze the compliance with the intended business requirements. The primary goal of the development of the prototype was to meet these requirements. The second part will try to express the added value of using a log scanner CDC technique over a metadata querier for day-to-day use.

This second part is by no means a performance benchmark to assess the quality and speed of the prototype's implementation. The design contains optimizations that would in some cases be applicable to any other synchronization framework. The opposite is also true: some parts of the prototype are left for future performance improvements. In other words, section 5.2 limits itself to the analysis of the performance that is a direct consequence of the streaming design rather than the implementation. For implementation-specific data appendix D can be consulted.

5.1 Compliance with business requirements

The design of the log sync was based on the requirements described in section 1.5. The goal of this section is to investigate whether or not all of these requirements have been met and to what extend that has succeeded.

5.1.1 Primary requirements

Low latency

The log sync leverages the log mechanism that is also used for the database's homogeneous replication. As such, the system exhibits a similar behavior of a slight delay between the master and the target/slave database. During the test period, the log reader was able to process the log entries immediately and no buffering on the internal queue was required. This amount of latency is low enough to be experienced as a nearly 'instant' synchronization for the end users. These results can however be influenced by a situations that would otherwise affect homogeneous replication too. Large transactions on the master and decreased availability of the target will increase the 'slave delay' up to the magnitude of seconds as normally seen on the production environment. This is a consequence of the replication principle rather than the sync framework design.

ACID properties

The sync framework uses the original transactions on the master as batches of work to synchronize. Each transaction is in itself an atomic and consistent chunk of 'changed data'. This delta will be applied to the other database as well in a similar atomic and consistent transaction. The system is strictly speaking not able to offer isolation across the source and target database. This property is inherited from the way the asynchronous replication works. Commits are acknowledged on the master and will eventually be committed on the slaves.

The isolation property that is sacrificed was coined in section 2.6.6 with the term 'relaxed ACID'. Instead of adopting a two-phase commit mechanism the system assumes that the transaction will eventually be committed on the other replicas. This means that an expensive two-phase commit is not required. The consequence will be that developers will have to keep this in mind when making assumptions related to the state of the database. Especially the case in which objects are being hydrated with data that is beyond the scope of the original transaction.

Reuse of business logic

The 'processor' part accepts input data from both the log scanner and the sync state CDC mechanisms. This is clearly displayed in fig. 4.1. As a consequence, all business logic resides in one software component and no code duplication is needed.

Ease of maintenance

The log sync framework heavily depends on replication technology. A general understanding of replication is therefore mandatory before knowing how to troubleshoot and maintain this synchronization framework. Experienced database administrators will feel comfortable while operating a sync framework like this because homogeneous replication is likely to be in place already. The log sync framework does not add much terms of maintenance effort in that case.

5.1.2 Functional requirements

Filtering

Fitlering of the transaction log is a very trivial operation which does not deserve much more attention: Just a filter based on the name of the database/table combination was enough to filter irrelevant data.

Data structure transformations

The design offers room for custom data transformations. Reusable transformers can be attached as 'listener' for specific data types that are being synchronized. Such a transformer has access to the full scope of a transaction because the log scanner scans the logs in chunks of transactions. This makes it easy to identify the scope of the data transformation: the context of the transaction is in memory already and there is no need to re-identify transactions or to query additional data for simple transformations.

Cache and index generation

This is a special case of a data transformation. Instead of transforming data from source to target, ad-

ditional target (meta)data can be generated. This so called metadata can be used to warm up caches for example. Another use case is the maintenance of indices that are based on the synchronized data.

5.2 Performance analysis

The performance analysis was carried out on one single but important part of the log sync framework: the CDC part. The framework contains two CDC types (metadata querying and log scanning) as was described in chapter 4. The log scanner is the framework's day-to-day modus operandi and the metadata querier is the one that will be used occasionally for manual interventions. The metadata querier comes with additional setup overhead in comparison to the log scanner. Both are repeatedly visualized in fig. 5.1.

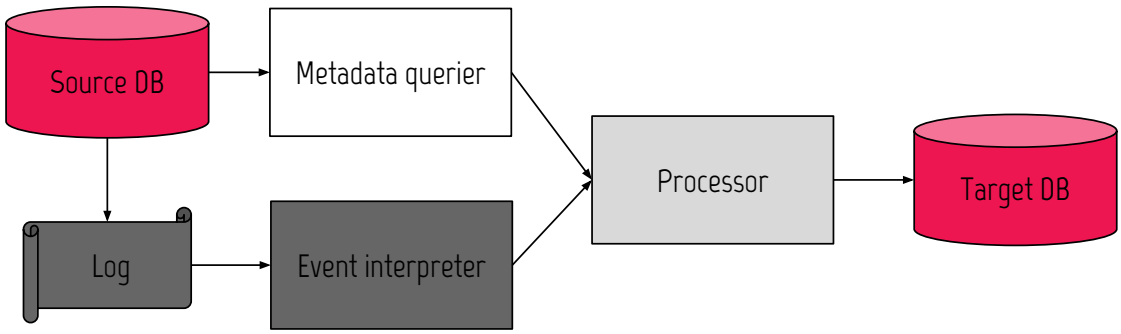

Figure 5.1: (Repeated) The sync framework supporting metadata based synchronization and log scanning CDC.

This analysis will express the amount of work saved by the log scanner with respect to the situation in which the metadata querier would have been the CDC technique. In other words, the added value of the log scanner is the amount of work that does not have to be executed in comparison with the metadata querier.

5.2.1 The cost of change data capturing based on sync states

The metadata querier scans the source database periodically for changes. This involves overhead because not all data that is referenced in the table needs to be synchronized. The more often this process runs, the less likely it will be that there is out-of-sync data. This means that the synchronization interval is a trade-off between up-to-date data and needless database load. The interval that was used for this experiment is the default production setting of the Distimo platform. That is two times per hour for a small time window and 3 times a day for a larger window to make sure that even delayed data is eventually synchronized. These intervals were chosen by trail and error and should therefore be an acceptable benchmark.

Definition of system load

The load on a database server can be expressed in several ways. Metrics like CPU time or memory consumption are among the ones that are relatively easy to measure. Others like (the chance of) deadlocks and the average response time are difficult on a live database system that handles multiple connections.

Those are the ones however that are relevant while assessing performance. In order to measure the system load objectively and reasonably independent of unrelated other processes, the load will be expressed in database time consumed. The assumption is that efficient queries will terminate quickly and inefficient queries take longer to compute.

5.2.2 Time consumption of CDC using metadata

An analysis was done over 26 days of log data from the production environment. During this period the synchronization framework was operating only on sync state metadata and the log scanner was not active. Over the course of this time span, a total of 49 hours was spent querying sync state metadata. Table 5.1 shows an itemized overview of the time per synchronization strategy. These strategies are the result of some Distimo specific fine-tuning of the system. Some data types must be updated very often, for which a small historical window is used. Others don't have to be updated very often and have therefore a bigger window to make sure that no historical data is missed.

Per strategy	Interval	Operations	Time consumption (±10m)	Percentage / total
Manual	Ad hoc	471,239	1:00 h	1.84%
Small window	2 / hour	4,950,383	9:30 h	19.31%
Large window	3 / day	20,210,059	38:30 h	78.85%
Total	-	25,631,681	49:00 h	100%

Table 5.1: Cummulative time consumed comparing sync states per 26 days is 49 hours.

Not all of the 25.6 million sync states that were analyzed did result in a synchronization action. In most cases, the sync state appeared to be in sync with the target database. Only 3 million (12%) sync states resulted in an actual synchronization. The other 88% was a waste of resources because no action was required but could not be known upfront. Regardless of the (in)efficiency of the metadata comparison, it is still important to realize that 100% of the work can be neglected once the log scanner is used.

5.2.3 Efficiency gained by log scanning

The log sync solution is able to replace the sync state comparison principle altogether because all data can be retrieved from the log. This will reduce the load on the database by approximately 49 hours per 26 days, which is almost 2 hours per day. In order to make a fair comparison with the log scanner one must take the overhead of the log scanner into consideration as well. This overhead is however negligible. The logs usually grow by tens of gigabytes per day, so the disk load of the log scanner is insignificant to the query load of the database itself.

5.3 Conclusion

The log scanning mode of the sync framework saves 2 hours of database work per day while delivering synchronization with a latency similar to that of replication. These two hours are saved by replacing an expensive and inefficient process by a process that is efficient by design because it operates only on data

in motion. The cost of this new log scanning process is almost free because it does not rely on database application resources, but on the file system.

Chapter 6

Conclusions

Data driven organizations often need to keep their data consistent across multiple data stores and database replicas. Synchronization tasks running at regular intervals introduce overhead and latency, which is undesirable for today's standards where customers demand real-time information. A synchronization framework having similarities with asynchronous replication has proven to be a suitable solution to this problem. Mobile app analytics firm Distimo has adopted this principle to improve the synchronization process between the DWH and the web application with the preservation of existing business logic and requirements in terms of ACID properties.

6.1 Achievements

The log synchronization framework is an appropriate design for low latency synchronization of structurally different ('heterogeneous') databases. The principle of log scanning is a robust setup and a proven technology in the field of database replication. Leveraging the replication process as a part of the existing ETL process results in a very versatile synchronization process that combines the best of both worlds:

1. Low synchronization latency similar to that of replication solutions.
2. Preservation of ACID properties under general circumstances.
3. Versatile data transformations using existing business logic.
4. On the fly metadata generation and cache maintenance by adopting event programming principles.
5. Efficient use of database resources by not having to poll the database for changes, saving 2 hours of database query time per day.
6. Small maintenance footprint that is similar to that of a replication setup.

The prototype that was developed for this research was tested on a production environment where it was integrated with the existing infrastructure. All business logic and data transformations are handled by the new log sync framework, as well the cache and metadata maintenance. The sync state mode is the primary CDC mode and the log scanner is used as a parallel shadow process for further testing.

6.2 Limitations

The log sync framework poses a limitation on the format of the log. The master needs to log every change to individual table rows rather than declarative statements manipulating multiple rows at once. Appendix B.1 describes the various replication formats of MySQL in more depth. As a consequence of logging every individual row changed, the log file will grow faster than it would be the case with statement-based logging.

Another limitation is primarily a design concern for data transformations. The changes of the rows that were persisted in the log lack the context of other records. It is therefore not possible to join additional data beyond the scope of the original transaction without violating ACID properties. The state of the database could have been altered since the moment of the original transaction commit.

6.3 Discussion and Future work

6.3.1 ACID compliance and limitations of the MySQL log format

When considering future work it might be an idea to look at the known issues and limitations of the design that relate to MySQL specifically. The fact that the log contains information as little as possible makes it very hard to do complex operations on the data without sending additional queries to the database and breaking the ACID principles. It has never been an option for Distimo to start patching MySQL by adding additional hints to the log: this would involve a huge effort of development and testing. Nonetheless it would be an interesting project to start investigating what would be an optimal log format to fully support the log sync design. This would simplify the overall design by not having to cache table structures and by not having to detect data types using a conversion table like table B.1.

6.3.2 Pub-sub instead of log scanning

A fundamentally different approach for the log sync principle would be to adopt a pub-sub design [Eugster et al., 2003]. The replication client would subscribe to certain parts of the data set and the database server would be responsible for pushing the adequate changes to the client. The JSON database RethinkDB[1] is modeled following this principle and it would be nice to see a similar solution for relational (SQL) databases. Microsoft SQL Server does have 'query notifications' since version 2005 but it doesn't seem to be used that much apart from being exclusively available on Windows. The idea behind a query notification is that the server signals when the result of a subscribed query changes. The underlying mechanism is also used for the detection when to rebuild table indexes, so the cost of a notification is low according to Microsoft. PostgreSQL offers inter client communication using the 'LISTEN' and 'NOTIFY' commands. This gives the possibility to address query notifications on the application layer: integration with a sync framework demands changes to the writer processes as well. Future research could focus to make this transparent like Microsoft's solution.

[1] http://rethinkdb.com/

Appendix A

The Distimo data infrastructure

Confidential

Appendix B

The MySQL binary log

B.1 MySQL Replication

MySQL replication works by using a binary logging mechanism. The MySQL master instance is the only instance on which data should be written at a particular moment in time. The master keeps track of all changes and logs these 'events' to a binary log file on hard disk. The MySQL binary log is strictly speaking not a proper write-ahead log because the log is written after the transaction is committed to the master. The MySQL InnoDB engine does use an internal Write-ahead log (WAL) however. For the sake of compatibility with other database systems, the binlog is considered to follow the same principles as real WALs just like other databases. One or more slave instances must be configured to scan the binary log and replay the events on their own database instance. Obviously, the master does not keep a log of **SELECT** statements as there is no need for the slaves to replay this queries because nothing will change in the database. Replication can be useful for load balancing and scaling. Slave instances are available as read-only databases for fast querying and the master's only concern is processing **INSERT**s, **UPDATE**s, and **DELETE**s. Important to note is that the slaves are no limiting factor for the master. While the master takes the lead in query processing, each slave can run at it's own pace. Slaves running far behind will serve outdated query results of course, so one should take caution. The master can be configured to use different types of replication, i.e. **a**) statement based, which replicates client queries literally to the slaves; **b**) row based, which replicates the changes of each row individually; and **c**) a combination of statement-based and row-based.Throughout this research, row-based replication was used and is assumed throughout the processes. The reason is that only row-based replication contains the explicit values needed for the sync to replicate to the frontend.

B.2 MySQL binary log APIs

This sections gives an overview of log scanning software that is able to interpret the MySQL binary log format.

MySQL Replication Listener

The MySQL Replication Listener is a semi-official MySQL project providing an STL/Boost based C++ library for processing a replication stream from a MySQL server [Pettersson and Kindahl, 2010]. Although being

a quite robust solution, the preference is to use a Java based solution for maintaining consistency in the code-base.
License: GNU GPL v2

Tungsten Replicator

Tungsten Replicator [Hodges, 2012] is an open-source data replication engine for MySQL, sponsored by Continuent, Inc. Tungsten Replicator is focused on achieving high performance across multiple sites within complex topologies. The replicator is the base for Tungsten Enterprise, a commercial database clustering product by Continuent. The open-source version is nevertheless quite abundant to act as a simple binlog parsing library.
License: GNU GPL v2

Open Replicator

Open Replicator is a MySQL binlog parser written in Java [Xu, 2012]. It has a small footprint and the projects aims to offer no more than just a practical library. Therefore, Open Replicator was used to build the prototype system.
License: Apache 2.0

B.3 The binary log format

The binary log consists of multiple files chained together by an internal reference at the end of each file. Every file has a header containing meta information like the version and file name. The rest of the file consists of multiple 'events' of different kinds. The following listing gives an overview of the most important events required to accomplish replication.

Format_description_event
: The first event of a log file, containing a description of the binary log version and server version.

Query_event
: This event is primarily used by statement based logging. It contains a query as executed by the master. In case of row-based logging, this event only occurs for queries which do not involve particular rows, for example data definition statements like **ALTER TABLE**-queries.

Rotate_event
: A rotate event announces a log rotation and is always the last event of a single binlog file. Its payload contains the name of the next binlog file.

Table_map_event
: Only used in row-based replication. This event precedes an event containing Data Manipulation Language (DML). That is an **UPDATE**, **INSERT**, or **DELETE** event on a particular table specified in the payload. Other payload data of interest are the column count, the column names, a definition of the data types of the columns, and a bitmap specifying which columns are nullable. The DML events

following a Table_map_event only contain raw data, without meta data like column names and data types. The binlog sync must therefore always remember the last Table_map_event since that is the only way to be able to process DML events.

Write_rows_event
This DML event corresponds to an **INSERT** query in row-based replication. Its payload contains an image of the inserted record in column order, without column names or data types specified.

Update_rows_event
This DML event corresponds to an **UPDATE** query in row-based replication. Its payload contains a before-image of the record, as well an after-image in column order, without column names or data types specified.

Delete_rows_event
This DML event corresponds to an **DELETE** query in row-based replication. Its payload contains a full after-image of the record in column order, without column names or data types specified.

Change data capture on a binary log is quite Spartan. After all, the binary log only contains almost a minimal amount of data for the replication to function. The slaves already have the right structure. Only the actual scalar data is logged since mentioning the data type is redundant in this case. Hence, the binary log stands not on his own and additional information is required in order to fully understand what the data represents. Appendix B.4 further elaborates on this.

B.4 Data types in MySQL, the binary log and Java

The binlogsync interacts with four 'interfaces', having their own data type characteristics. To guarantee correct operation, each data field is type checked to verify that the structure between them is consistent. This involves:

- MySQL data types (as the data is stored in DWH)
- Binlog / OpenReplicator data types (what is read from the Binary (transaction) log (binlog))
- JDBC data types (used for retrieving table definitions)
- Java data types (for filling Hibernate entities)

Ideally, all data types would have a one-to-one mapping. Unfortunately, this is not the case. Based on information from the binary log and JDBC, the binlogsync must be able to determine the original MySQL data type, from which the final Java type can be derived. Figure B.1 shows how the binlogsync is connected.

MySQL currently supports approximately 30 data types. The 'traditional' data types include integers ranging from 1 to 8 bytes, dates, datetimes, characters of fixed and varying lengths, and various binary types. Besides these common types, MySQL also supports enumerations and sets which are essentially bitmaps.

The **binlog** does not offer a counterpart for all 30 MySQL types. These are mapped to only 16 data types in the log. This means that it will not be possible to unambiguously determine the original data type by

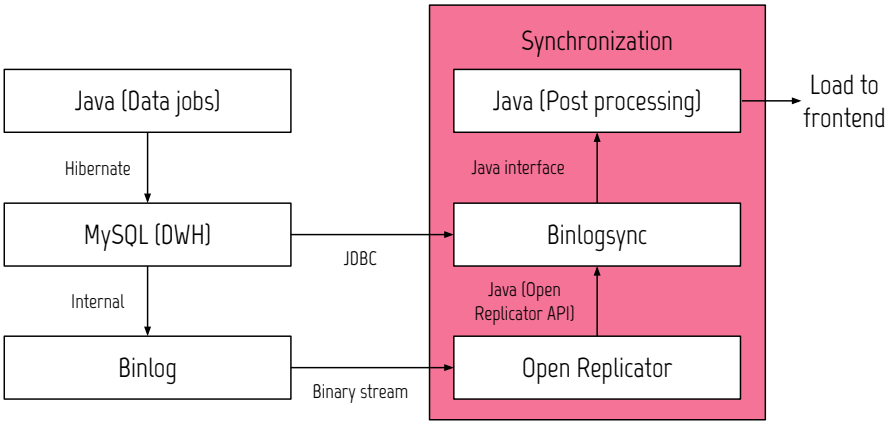

Figure B.1: The binary log determines the actual MySQL data types based on input from the binlog API and the JDBC API and outputs a Java objects of according types.

solely reading the log. This sometimes leads to confusing situations. For example, all **Binary Large Object (BLOB)** and **TEXT** data types appear as **BLOB** in the binary log. The types **BINARY**, **CHAR**, **ENUM**, and **SET** are all mapped to the **STRING** log data type.

The **Open Replicator** API does a good job in translating the 16 binlog data types to its respectful Java counterparts.

JDBC, which is the API used to retrieve the database structure, supports most MySQL data types, except for **ENUM** and **SET**. JDBC also tends to interpret **BIT(1)** and **TINYINT(1)** as boolean instead of a bytearray and an integer respectively.

The combination of data types from the different interfaces matters when in comes on type checking. Only a few combinations are valid, which are shown in table B.1. Other combinations point to an inconsistency which will result in an error.

Table B.1: Data type mapping reference for MySQL internals, binary log, Open Replicator and Java runtime.

MySQL	ID	Binlog Data Type	ID	JDBC Data Type	JDBC Java Type	Open Replicator	OR Java Type	Final conversion
bit(1)	16	MYSQL_TYPE_BIT	-7	Bit	java.lang.Boolean	BitColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
bit(2)	16	MYSQL_TYPE_BIT	-7	Bit	[B	BitColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
tinyblob	252	MYSQL_TYPE_BLOB	-2	Binary	[B	BlobColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
blob	252	MYSQL_TYPE_BLOB	-4	Longvarbinary	[B	BlobColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
longblob	252	MYSQL_TYPE_BLOB	-4	Longvarbinary	[B	BlobColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
mediumblob	252	MYSQL_TYPE_BLOB	-4	Longvarbinary	[B	BlobColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
longtext	252	MYSQL_TYPE_BLOB	-1	Longvarchar	java.lang.String	BlobColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
mediumtext	252	MYSQL_TYPE_BLOB	-1	Longvarchar	java.lang.String	BlobColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
text	252	MYSQL_TYPE_BLOB	-1	Longvarchar	java.lang.String	BlobColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
tinytext	252	MYSQL_TYPE_BLOB	12	Varchar	java.lang.String	BlobColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
date	10	MYSQL_TYPE_DATE	91	Date	java.sql.Date	DateColumn	java.sql.Date	java.util.Date
datetime	12	MYSQL_TYPE_DATETIME	93	Timestamp	java.sql.Timestamp	DatetimeColumn	java.sql.Timestamp	java.util.Date
double	5	MYSQL_TYPE_DOUBLE	8	Double	java.lang.Double	DoubleColumn	java.lang.Double	java.lang.Double
float	4	MYSQL_TYPE_FLOAT	7	Real	java.lang.Float	FloatColumn	java.lang.Float	java.lang.Float
mediumint	9	MYSQL_TYPE_INT24	4	Integer	java.lang.Integer	Int24Column	java.lang.Integer	java.lang.Integer
int	3	MYSQL_TYPE_LONG	4	Integer	java.lang.Integer	LongColumn	java.lang.Integer	java.lang.Integer
bigint	8	MYSQL_TYPE_LONGLONG	-5	Bigint	java.lang.Long	LongLongColumn	java.lang.Long	java.lang.Long
decimal	246	MYSQL_TYPE_NEWDECIMAL	3	Decimal	java.math.BigDecimal	DecimalColumn	java.math.BigDecimal	java.math.BigDecimal
smallint	2	MYSQL_TYPE_SHORT	5	Smallint	java.lang.Integer	ShortColumn	java.lang.Integer	java.lang.Short
binary	254	MYSQL_TYPE_STRING	-2	Binary	[B	StringColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
char	254	MYSQL_TYPE_STRING	1	Char	java.lang.String	StringColumn	java.nio.ByteBuffer	java.lang.String
enum	254	MYSQL_TYPE_STRING	1	Char	java.lang.String	StringColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
set	254	MYSQL_TYPE_STRING	1	Char	java.lang.String	StringColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
time	11	MYSQL_TYPE_TIME	92	Time	java.sql.Time	TimeColumn	java.sql.Time	java.util.Date
timestamp	7	MYSQL_TYPE_TIMESTAMP	93	Timestamp	java.sql.Timestamp	TimestampColumn	java.sql.Timestamp	java.util.Date
bool / tinyint(1)	1	MYSQL_TYPE_TINY	-7	Bit	java.lang.Boolean	TinyColumn	java.lang.Integer	java.nio.ByteBuffer
tinyint	1	MYSQL_TYPE_TINY	-6	Tinyint	java.lang.Integer	TinyColumn	java.lang.Integer	java.lang.Integer
varbinary	15	MYSQL_TYPE_VARCHAR	-3	Varbinary	[B	StringColumn	java.nio.ByteBuffer	java.nio.ByteBuffer
varchar	15	MYSQL_TYPE_VARCHAR	12	Varchar	java.lang.String	StringColumn	java.nio.ByteBuffer	java.lang.String
year	13	MYSQL_TYPE_YEAR	91	Date	java.sql.Date	YearColumn	java.sql.Date	java.lang.Integer

B.5 MySQL compatibility

Important changes possibly affecting binary log synchronization in future versions are listed below. All these changes are from the MySQL release notes [Oracle Corporation, 2015].

B.5.1 Changes and new features in MySQL 5.6

Global transaction identifiers This release introduces Global Transaction Identifiers (GTIDs) for MySQL Replication. A GTID is a unique identifier that is assigned to each transaction as it is committed; this identifier is unique on the MySQL Server where the transaction originated, as well as across all MySQL Servers in a given replication setup. Because GTID-based replication depends on tracking transactions, it cannot be employed with tables that employ a nontransactional storage engine such as MyISAM; thus, it is currently supported only with InnoDB tables.
Since: MySQL 5.6.5 (2012-04-10, Milestone 8)

Multi-threaded slave execution MySQL replication now supports a multi-threaded slave executing replication events from the master across different databases in parallel, which can result in significant improvements in application throughput when certain conditions are met. The optimum case is that the data be partitioned per database, and that updates within a given database occur in the same order relative to one another as they do on the master. However, transactions do not need to be coordinated between different databases.
Since MySQL 5.6.3 (2011-10-03, Milestone 6)

Transaction logging BEGIN, COMMIT, and ROLLBACK statements are now cached along with the statements instead of being written when the cache is flushed to the binary log. This change does not affect Data Definition Language (DDL) statements — which are written into the statement cache, then immediately flushed — or Incident events (which, along with Rotate events, are still written directly to the binary log). **Since MySQL 5.6.3 (2011-10-03, Milestone 6)**

No empty transactions A transaction was written to the binary log even when it did not update any nontransactional tables. (Bug #11763471, Bug #56184)
Since MySQL 5.6.3 (2011-10-03, Milestone 6)

Row imaging Added the binlog_row_image server system variable, which can be used to enable row image control for row-based replication. This means that you can potentially save disk space, network resources, and memory usage by the MySQL Server by logging only those columns that are required for uniquely identifying rows, or which are actually changed on each row, as opposed to logging all columns for each and every row change event. In addition, you can use a "noblob" mode where all columns, except for unneeded BLOB or TEXT columns, are logged.
Since MySQL 5.6.2 (2011-04-11)

Server GUIDs Globally unique IDs for MySQL servers were implemented. A Universally Unique Identifier (UUID) is now obtained automatically when the MySQL server starts. The server first checks for a UUID written in the auto.cnf file (in the server's data directory), and uses this UUID if found. Otherwise,

the server generates a new UUID and saves it to this file (and creates the file if it does not already exist). This UUID is available as the server_uuid system variable.

Since: MySQL 5.6.0 (Not released, Milestone 4)

Appendix C

Log Synchronization design overview

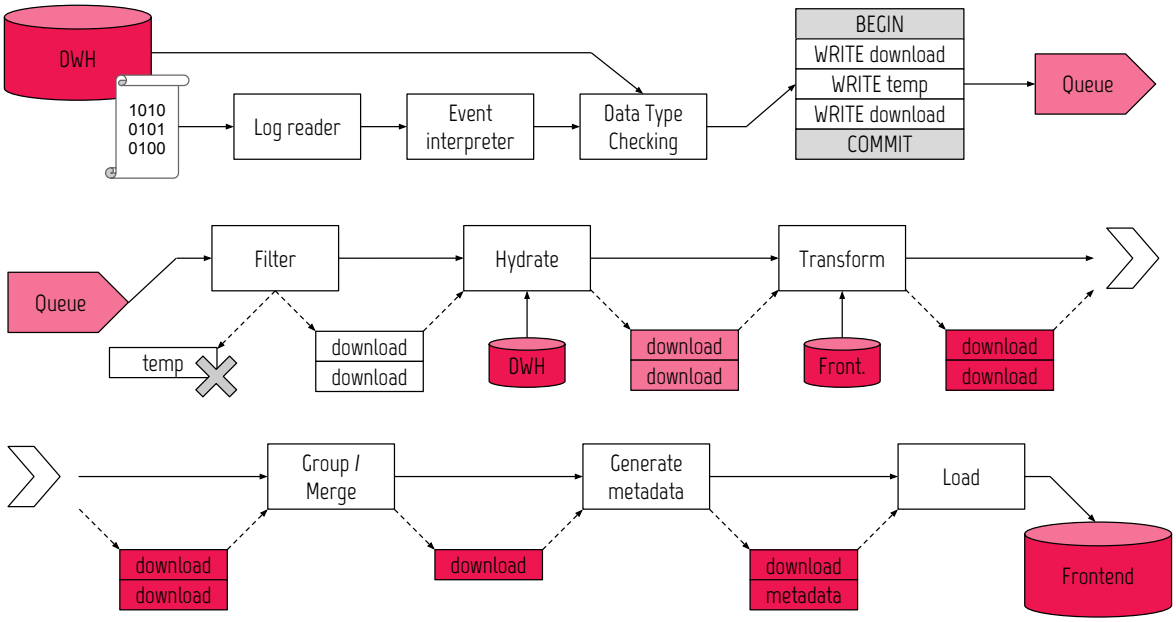

Figure C.1: Full process

C.1 Design limitations

C.1.1 Handling DDL query events

A special case of events are those containing DDL queries. In other words: **ALTER TABLE** queries. These queries only occur at times when a version of the software is deployed or a data migration takes place with the use of a temporary table. During the extraction step described in section 4.2.1, validation of the data takes place. The data captured from the log is validated against a cached table definition. This cache is built as soon the log sync starts and must be invalidated as soon a DDL events occurs. Consecutive write events will then be validated against an up-to-date definition. This situation is illustrated in fig. C.2.

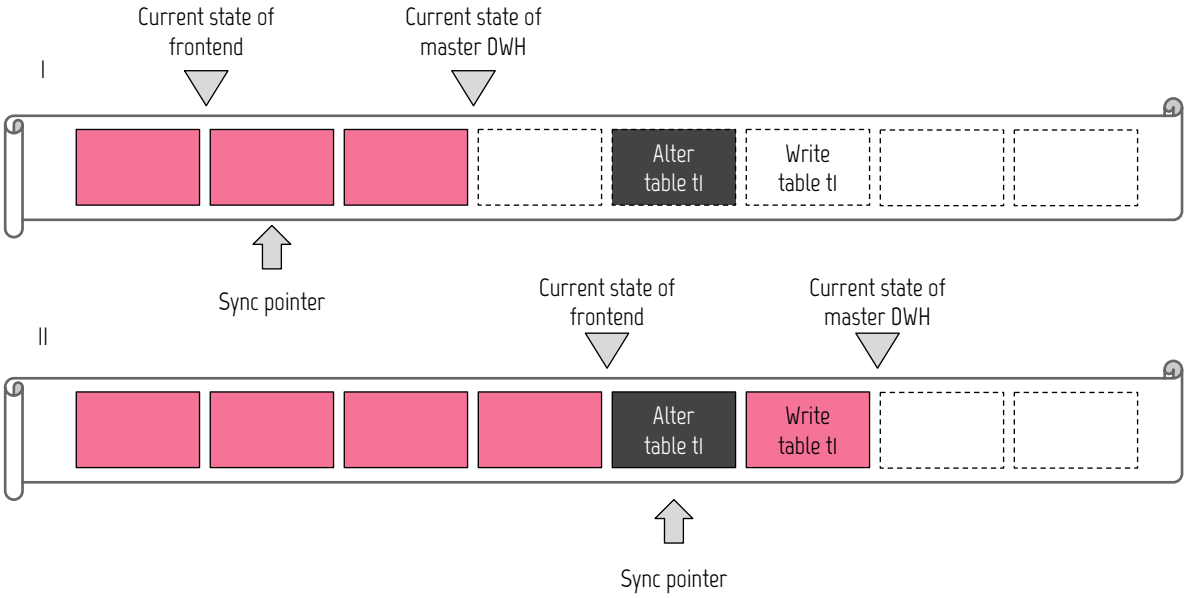

Figure C.2: **Phase I**: The log sync starts and caches a table structure of table t1 as defined by the current state of the DWH. **Phase II**: Table t1 is modified by a DDL query. The log-sync refreshes its cache with an up-to-date definition of table t1.

Validating the log's event structures against the current state of the database could result in problems if the database structure was changed over time and the log file is outdated. Figure C.3 shows this situation where the log sync has just started and lags behind on the state of the master. The sync is going to catch up with the tail of the log by processing all events from left to right in the picture. The write operation on table t1 will be compared with the current schema of table t1 on the master. The schema of t1 is however inconsistent with the write event in the log because the table was altered somewhere between the sync pointer and the log tail.

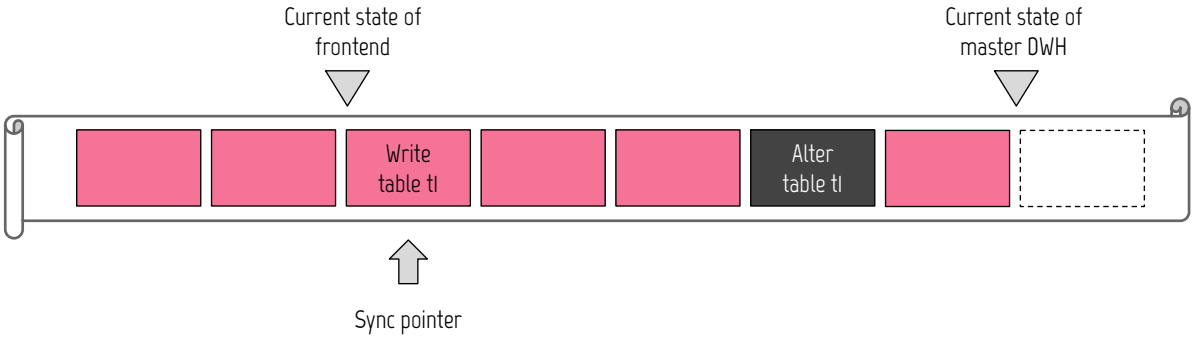

Figure C.3: The binlogsync validates an event using a new (and possibly incompatible) table structure set by the **ALTER TABLE** event.

The best thing to do is to make sure this situation never happens. This can be done by making sure that the sync is not running behind while doing the **ALTER TABLE**. The sync should pass the alter statement in

the log to make sure the situation in fig. C.3 cannot happen in the future. In case that sync was not able to catch up with the alter statement, the good old 'sync state sync' will need to recover consistency across the databases from where the log sync can take over.

Appendix D

Performance statistics by Munin

A performance assessment of the sync framework on a production environment is near to impossible to do correctly. The target database is used by a large number of ad-hoc processes and this harms proper test isolation. Although the regular system monitoring software is not an appropriate tool to draw conclusions upon, the software helped to gain additional insight in the way the sync framework operates. Figure D.1 displays one year of data retrieved from the monitoring software Munin.

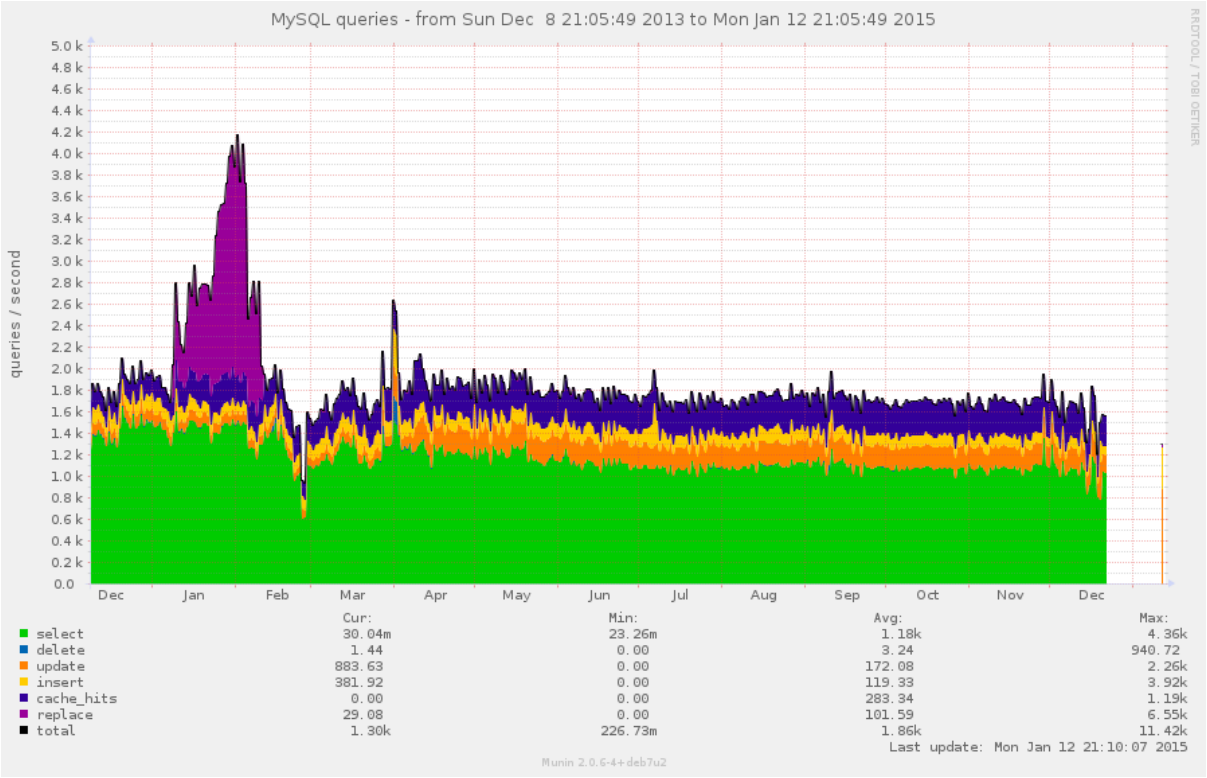

Figure D.1: Number of queries per second on the target database, retrieved from system monitor Munin.

The graphic displays the number of queries per second processed by the target database, itemized by query type. The large purple spike ranging from Jan - Feb was an early prototype that was geared towards **REPLACE** queries. For the majority of queries the net result was zero because the **REPLACE** was

issued regardless of whether the record had to be updated or not. This inefficient behavior was eventually corrected and the more efficient **ON DUPLICATE KEY UPDATE** was used in relevant cases. By the end of March the definitive processing pipeline was deployed on the production environment. This can be identified based on the increase of **UPDATE** queries (orange). The automated cache and metadata maintenance contributed for hundreds of extra updates per second. This seems like a heavy performance penalty, but the queries themselves are very efficient. Their task is to update only one record at a time based on the full primary key of the record. This immediately exposes the problem of graphs like these: the number of queries per second is not a good measure for actual performance. This is the reason that Munin was not used for further analysis of the framework design.

Bibliography

S. W. Ambler. The object-relational impedance mismatch. **Agile Database Techniques, Wiley Publishing**, 2003.

R. M. Bruckner, B. List, and J. Schiefer. Striving towards Near Real-Time Data Integration for Data Warehouses. In Y. Kambayashi, W. Winiwarter, and M. Arikawa, editors, **Data Warehousing and Knowledge Discovery**, number 2454 in Lecture Notes in Computer Science, pages 317–326. Springer Berlin Heidelberg, Jan. 2002. ISBN 978-3-540-44123-6, 978-3-540-46145-6. URL `http://link.springer.com/chapter/10.1007/3-540-46145-0_31`.

E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In **Proceedings of the 2008 ACM SIGMOD international conference on Management of data**, SIGMOD '08, pages 739–752, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376691. URL `http://doi.acm.org/10.1145/1376616.1376691`.

S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. **ACM Sigmod record**, 26(1):65–74, 1997. URL `http://www.acm.org/sigmod/record/issues/9703/chaudhuri.ps`.

L. Chen, W. Rahayu, and D. Taniar. Towards Near Real-Time Data Warehousing. In **2010 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)**, pages 1150 –1157, Apr. 2010. doi: 10.1109/AINA.2010.54.

J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. **Commun. ACM**, 51 (1):107–113, Jan. 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL `http://doi.acm.org/10.1145/1327452.1327492`.

R. Dhamane, M. P. Martínez, V. Vianello, and R. J. Peris. Performance Evaluation of Database Replication Systems. In **Proceedings of the 18th International Database Engineering & Applications Symposium**, IDEAS '14, pages 288–293, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2627-8. doi: 10.1145/2628194.2628214. URL `http://doi.acm.org/10.1145/2628194.2628214`.

M. Eccles, D. Evans, and A. Beaumont. True Real-Time Change Data Capture with Web Service Database Encapsulation. In **2010 6th World Congress on Services (SERVICES-1)**, pages 128 –131, July 2010. doi: 10.1109/SERVICES.2010.59.

P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. **ACM Comput. Surv.**, 35(2):114–131, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857078. URL `http://doi.acm.org/10.1145/857076.857078`.

M. Fowler. Event Sourcing, Dec. 2005. URL http://martinfowler.com/eaaDev/EventSourcing.html.

S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In **Proceedings of the nineteenth ACM symposium on Operating systems principles**, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450.

R. Hodges. tungsten-replicator - A high performance, open source, data replication engine for MySQL - Google Project Hosting, Oct. 2012. URL http://code.google.com/p/tungsten-replicator/.

P. Hofmann. Multi-Master Replication Manager for MySQL [MMM for MySQL Wiki], Nov. 2012. URL http://mysql-mmm.org/doku.php.

W. H. Inmon. **Building the data warehouse**. Wiley, 2005. ISBN 9780764599446.

M. Kifer, A. J. Bernstein, and P. M. Lewis. **Database Systems: An Application-oriented Approach**. Addison-Wesley, 2005. ISBN 9780321228383.

T. M. Nguyen and A. M. Tjoa. Zero-latency data warehousing (ZLDWH): the state-of-the-art and experimental implementation approaches. In **2006 International Conference on Research, Innovation and Vision for the Future**, pages 167 – 176, 2006. doi: 10.1109/RIVF.2006.1696434.

Oracle Corporation. MySQL :: MySQL 5.6 Release Notes, Jan. 2015. URL http://dev.mysql.com/doc/relnotes/mysql/5.6/en/.

S. Perez. App Annie Acquires Competitor Distimo, Raises Another $17 Million From Existing Investors, May 2014. URL http://social.techcrunch.com/2014/05/28/app-annie-acquires-competitor-distimo-raises-another-17-million-from-existing-investors

K. Pettersson and M. Kindahl. MySQL Replication Listener in Launchpad, Apr. 2010. URL https://launchpad.net/mysql-replication-listener.

R. W. Schulte. Introducing the zero-latency enterprise. **Gartner Research, June**, 1998.

J. Shi, Y. Bao, F. Leng, and G. Yu. Study on log-based change data capture and handling mechanism in real-time data warehouse. In **Proceedings - International Conference on Computer Science and Software Engineering, CSSE 2008**, volume 4, pages 478–481, 2008.

J. Shi, Y. Bao, F. Leng, and G. Yu. Priority-Based Balance Scheduling in Real-Time Data Warehouse. In **Ninth International Conference on Hybrid Intelligent Systems, 2009. HIS '09**, volume 3, pages 301–306, Aug. 2009. doi: 10.1109/HIS.2009.275.

J. Song, Y. Bao, and J. Shi. A Triggering and scheduling approach for ETL in a real-time data warehouse. In **Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010**, pages 91–98, 2010.

Y. Wang and J. Chiao. Data Replication in a Distributed Heterogeneous Database Environment: An Open System Approach. In **, IEEE 13th Annual International Phoenix Conference on Computers and Communications, 1994**, page 315, Apr. 1994. doi: 10.1109/PCCC.1994.504132.

L. Wong, N. Arora, L. Gao, T. Hoang, and J. Wu. Oracle Streams: A High Performance Implementation for Near Real Time Asynchronous Replication. In **IEEE 25th International Conference on Data Engineering, 2009. ICDE '09**, pages 1363 –1374, Apr. 2009. doi: 10.1109/ICDE.2009.121.

J. Xu. open-replicator - A high performance MySQL binlog parser written in Java - Google Project Hosting, Aug. 2012. URL `https://code.google.com/p/open-replicator/`.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In **Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing**, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1863103.1863113`.