CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Domain Specific Continuous Queries Implemented using Database Management Systems

Master's thesis in Computer Science - Algorithms, Languages and Logic

ANTON LINDGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016

Domain Specific Continuous Queries Implemented using Database Management Systems

ANTON LINDGREN

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Computing Science Division
Chalmers University of Technology
Gothenburg, Sweden 2016

Domain Specific Continuous Queries Implemented using Database Management Systems
ANTON LINDGREN

© ANTON LINDGREN, 2016.

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Examiner: Graham Kemp, Department of Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Computing Science Division
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2016

Domain Specific Continuous Queries Implemented using Database Management Systems

ANTON LINDGREN
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

We present an alternative to Continuous Queries, these are queries that deliver streams of data to clients as opposed to common queries which only respond with a single response. We define Domain Specific Continuous Queries as a query that, apart from an initial response with the matching set of records, streams all updates affecting the query. We implement this definition of a Continuous Query for a specific domain where an existing solution and problem definition already exists. The new implementation uses classic query-response database systems to supply the initial response. A persistent data structure containing the full state of the data is kept in memory, which allows our prototype to deduce which queries an incoming update affects. The new implementation is on par with the existing solution's feature set and greatly outperforms it for our metrics. Especially for initial response latency.

Keywords: databases, stream processing, event sourcing, persistent data structures

Acknowledgements

First and foremost I would like to give a huge thank you to Yolean for providing me with an interesting and practical problem definition. Especially I would like to thank my industrial supervisor Staffan Olsson for all the help I've gotten both prior to and during this project. I would also want to thank my academic supervisor Andreas Abel for taking his time to help me find an academic approach to Yolean's problem and for showing great interest in the experiments performed. The same thanks goes out to my examiner Graham Kemp for taking his time to help and guide me all throughout the project.

Anton Lindgren, Gothenburg, September 2016

Contents

List of Figures

1

Introduction

Web Applications that require user collaboration with short feedback-loops between users are becoming more and more common. Google Docs has for long set a high standard on user collaboration in modern text editors, and Trello is one of the leading applications when it comes to project planning on the web. Consequently, a lot of Open Source solutions have sprung up that let developers implement such collaborative web applications more easily. For example there is TogetherJS[1] for distributing user information such as the location of the mouse-pointer, SubstanceIO[2] for text collaboration and SocketIO[3] which provides a reliable real-time API on top of HTTP and Web Sockets. This enables companies to quickly implement such real-time collaboration services with minimal resources.

The company Yolean AB develops applications that digitize management methods. Their products stem from years of research within fields of Product and Production Development. Yolean's most mature product is the Visual Planning (VP) application, see Figure 1.1. The VP application has a few common use-cases with Trello as they are both planning applications. Within both applications, users can create and discuss a common plan or schedule for either projects or daily routines within an organization or group of people. However, what sets the VP application apart from Trello is that it is aimed towards organizations with a higher requirement on their planning process and thus an interest in the Visual Planning method. It differs from Trello's Kanban model (which is common in many Scrum- or agile-based methods) in that it has a time line on the X-axis and thus also resembles a storyline.

[1] `https://togetherjs.com/`
[2] `http://substance.io/`
[3] `http://socket.io/`

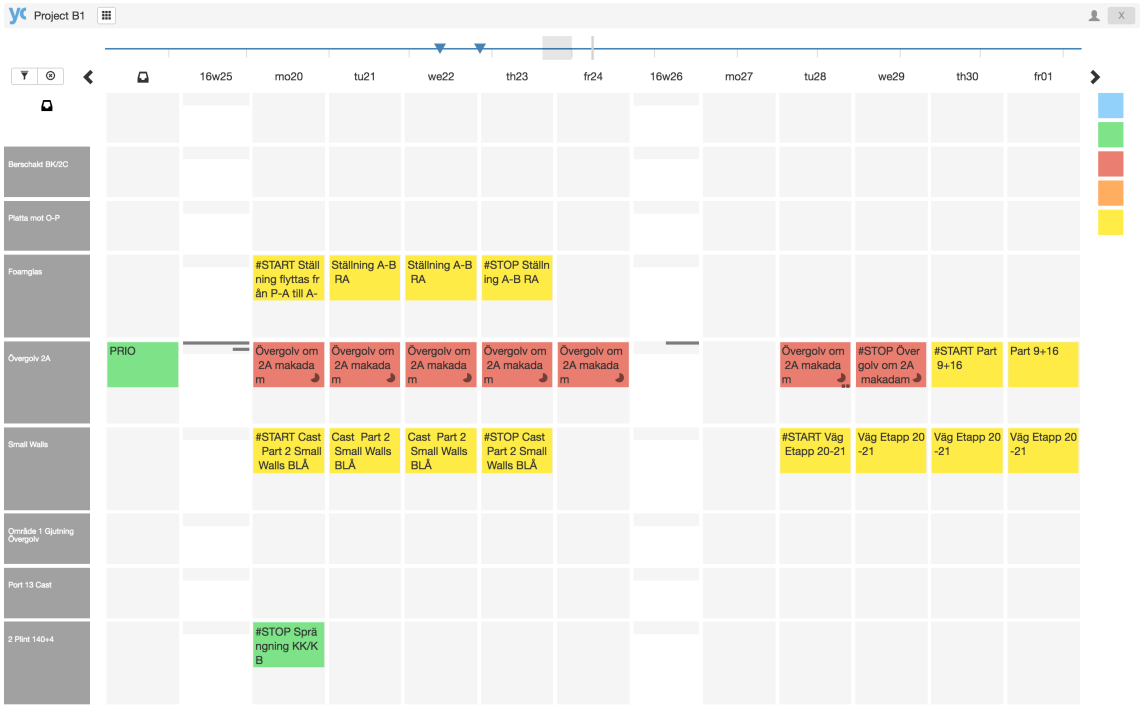

Figure 1.1: Screenshot of Yolean's Visual Planning application. It visualizes planning data with users on the Y-axis and time (here days, but there are different resolutions to choose from) on the X-axis. The colored blocks in the figure are what the VP application refers to as *tasks* which contain information on some job that a group of people has agreed upon.

This thesis develops and evaluates a new prototype for one of Yolean's existing backend services called the *Live Server*. We will start by introducing the Yolean VP application domain and describe the existing implementation of the Live Server. We will propose a method to speed up query-response times significantly while still keeping the same functionality in the Live Server by querying different database solutions as a kind of index for the service. Finally, the pros and cons of the new solution will be discussed while comparing benchmarks between the current and new method.

1.1 Yolean Visual Planning Domain

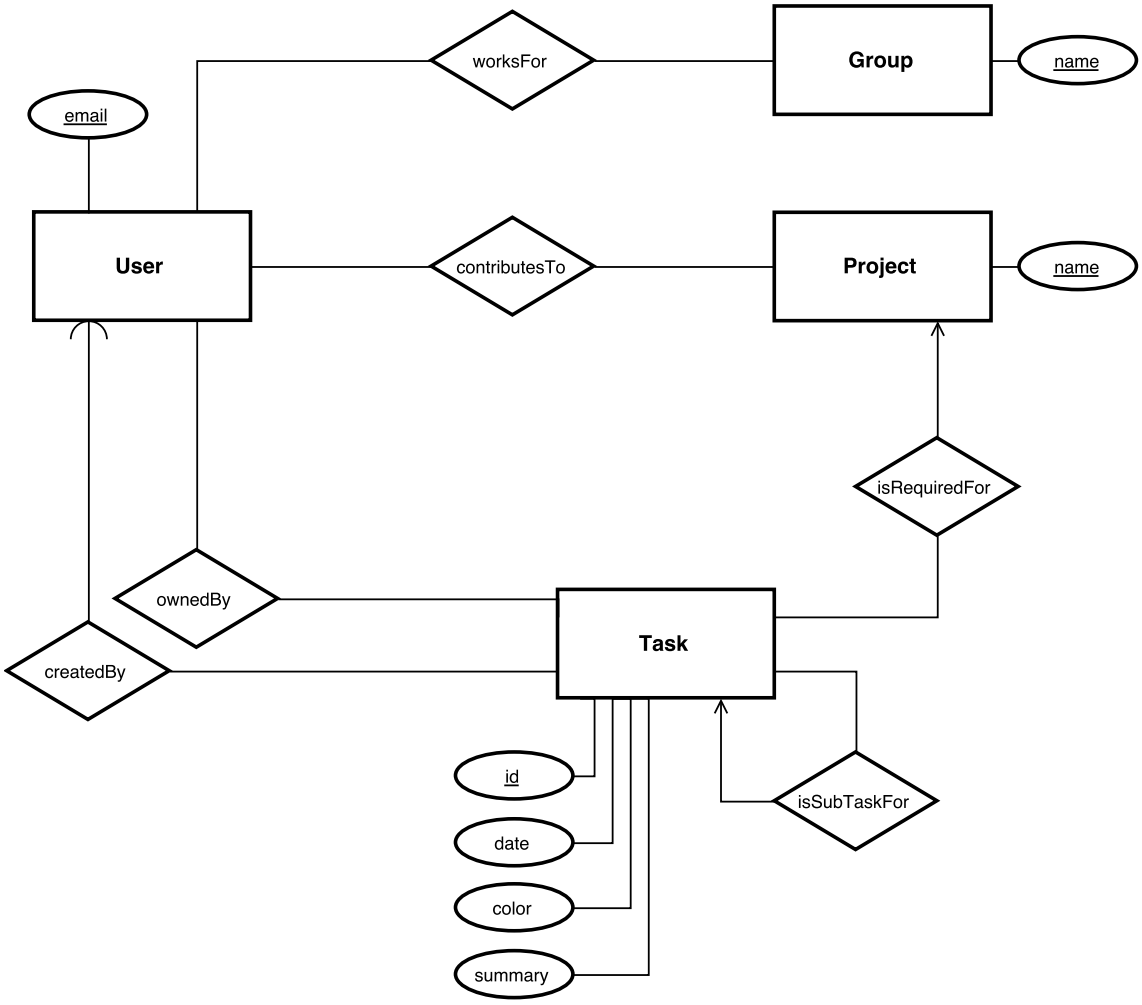

Figure 1.2: Entity-Relationship Diagram for Yolean's Visual Planning Domain.

The VP application domain can be simplified into a handful of entities. These entities and their relations are formalized by the Entity-Relation (ER) Diagram in Figure 1.2. Below follows a short description of each entity:

- **Tasks**, which are visualized as post-it notes in different colors. Preferably, they should deliver some work to a *project*. Tasks can also be split into sub-tasks, which gives them a recursive relation between a parent task and its child-tasks.
- **Users**, which occupy a single row in most views of the application. Users also create (or modify) and own tasks.
- **Groups** and **Projects**, which for our purpose can be simplified as different ways of querying for tasks.

1.2 Yolean Visual Planning Use Cases

The VP application visualizes queries of tasks (i.e. a grid such as the one seen in Figure 1.1). We refer to these views as *boards*. Each board displays a subset of all tasks within one of their customer's database. The subset is defined by querying the database through a collaboration service that Yolean has developed, called the Live Server. The characteristics of a query varies with the type of the board itself. Three common board types are: project boards, which queries for all tasks within a project; group boards, which queries for all tasks for a set of users; and personal boards, which queries for all tasks that are owned by the user.

A project board often contains a lot of tasks per week, grouped by several users (where each user has its own row). A project board is often used for several years in production, accumulating a lot of tasks in total. Therefore, in order to provide a smooth experience for the user, the client application relies on being able to request only tasks that are planned inside the user's current time interval. As in most applications with a time line, the user is able to navigate to different date ranges and choose time resolution (like how calendar applications let you choose between daily, weekly or monthly views). This means that each client connected to a project board may require different subsets of the project's tasks.

The main focus for the VP application is user collaboration. To Yolean, user collaboration requires that whenever an object that matches a client's query is modified, the user must receive this new modification. An intriguing example of this is when user A moves a task X into user B's viewport. Note that user B's client does not have any data for this task prior to the modification. Thus, the Live Server must understand this and send the full object to user B's client (as opposed to when the client already has the most recent state and only requires the modification). A second example, where user A assigns task X to user B from a project board, further requires this change not only to be sent to any of user B's clients connected to this project board, but also to user B's personal board.

To sum up: Many of Yolean's use cases for the Visual Planning application can be realized through the queries that the ER-Diagram in Figure 1.2 supports. The list below contains the most apparent ones:

- Select all tasks for a project/group/user, scheduled within a date range
- Select all tasks that are part of a given task, recursively
- Select all tasks that a given task is part of, recursively
- Select all delayed tasks (where the planned done date has already passed)

These queries are all of the sort that a Relational Database, a Document Store or for the recursive queries even a Graph Database would be suitable as a backend.

However, Yolean's application also expects a stream of all future changes affecting this query!

1.3 Yolean's current Live Server

Yolean's way of supporting collaboration within an organization differs from many other popular tools in that their data is meant to be visualized as different views for different scenarios. So the same set of tasks can be viewed and filtered in many different ways depending on the current use-case, and the applications expect to be notified of any changes to this data. For Yolean, this has lead to a backend solution developed in-house on top of the SocketIO library.

The main focus for the VP application is collaboration on the **task** objects. Tasks are modeled as flat Javascript Object Notation (JSON)[4] records, meaning that they consist of multiple key-value attributes. Each task contains a unique id attribute so they are all easily distinguishable from each other. To keep the problem more general we can imagine tasks as records from now on.

The collaboration service Yolean has developed, the **Live Server**, is responsible for distributing subsets of the set of all records to their clients. A subset S contains the records from the set of all records which passes its predicate P, where P is defined by the client's query. Clients of this service are able to modify existing records and add new records. Modifications trigger update events to connected clients. An update event can be one of three types:

- The addition of a new record to the subset.
- A modification of an existing record.
- The deletion of an existing record from the subset.

An existing implementation based on the Javascript library Backbone exists today. Hence, it runs on the NodeJS platform. This implementation was built during the summer of 2015, in order to be able to pilot the concept Interconnected Boards, a concept which triggered the Visual Planning collaboration scenarios explained earlier in Section 1.2.

1.4 Problems with the current solution

The fact that the existing implementation is based on legacy code causes it to come with a few limitations. These limitations mainly stem from how it stores data

[4] `http://www.json.org/`

by using an open-source issue tracker called Trac, whose query response times are already a bottle-neck in the system today.

The current implementation responds with the initial result set for a client by naively matching each record against its predicate function. So for n records each client requires $O(n)$ setup time. Yolean has not yet investigated at which combination of n records and m users the service will start to cause problem in production. They wish to do this, but for a new implementation that is not bound by legacy. Such an approach suits a master thesis very well as it allows the student to experiment more freely even though it might require a lot of initial work.

1.5 Solution Overview

Arasu et al. [1] has designed a system called STREAM, a Data Stream Management System (DSMS)[5] built around a concept they call *Continuous Queries*. They use this definition for describing queries which are run over a set of data over time, as in opposition to an ordinary request-response query. This definition is also used by Babu and Widom [2] to describe something very similar. Thus, we will distinguish the types of queries that the Live Server handles as continuous queries from now on. However, Arasu et al. [1] and Babu and Widom [2] both focus on designing a query language similar to SQL for which they analyze how to process queries efficiently. To keep the implementation for this project simple and extensible, we decided on a minimal query language based on JSON.

A *Continuous Query* is defined by providing the *name* of the query along with its *parameters*. The parameters are simply key-value mappings that will be translated into a meaningful query by the server. Thus, parameter values can be of any type allowed within the JSON format.

```
{
  name: "name",
  params: {
    ...
  }
}
```

We will use this concept of continuous queries to describe the combination of classical request-response queries and the update events that Yolean's applications demand.

We propose a solution for resolving continuous queries where common query-response database systems such as Relational Database Management Systems (RDBMS) and Search Engines are used as *Indexes* in combination with an in-memory state, implemented as a *Persistent Data Structure* [6]. The indexes will resolve the initial set of

[5] https://en.wikipedia.org/wiki/Data_stream_management_system
[6] https://en.wikipedia.org/wiki/Persistent_data_structure

records for the response and the in-memory state will be used to match updates to queries. In order to implement this and avoid problems such as the *Dual Write* problem [8], a method from Stream Processing[7], namely *event-sourcing* will be applied to construct indexes and the in-memory state.

In short, the thesis uses Apache Kafka[8] as an event log and thus the new collaboration service's only persistent storage. The event log will be read individually by a few different *Index Clients* from within the Live Server. The server component will then route incoming continuous queries to its appropriate index client. The index clients then become responsible for responding to these queries, just as a regular request-response backend service would do. Each index client will accomplish this by mapping named queries into something that its backend (for example a PostgreSQL database) would understand. In addition to the initial response, each index client will stream updates for its registered queries. This will be accomplished by maintaining a minimal set of in-memory states of the dataset using the persistent data structure.

1.6 Thesis Overview

This section provides a short overview of the rest of this thesis. In Chapter 2 the underlying concepts used to implemenent our solution is described. Chapter 3 continues by introducing the Live Server prototype focusing on its protocol and the flow of data within the service. Our Live Server prototype was benchmarked during a set of experiments that we present in Chapter 4. The results from these experiments are discussed in Chapter 5, where we highlight some limitations for the prototype as well as try to predict the impact it would have for Yolean. Finally, Chapter 6 wraps up the thesis by concluding that the prototype is going to be adopted by Yolean thanks to promising results, but that for a more general adoption more work might be needed.

[7] `https://en.wikipedia.org/wiki/Stream_processing`
[8] `http://kafka.apache.org/`

2

Background

This chapter describes four different areas within the field: Stream Processing, Persistent Data Structures, B-Trees and Inverted Indexes. The first two are related to the project as they help build a simple implementation for distributing updates within the system. The other two relates the database solutions used throughout this project by explaining why they were expected to provide a performance boost for queries to the new Live Server prototype.

2.1 Stream Processing and Event Logs

The overall structure of the project is inspired by Kleppmann and his thoughts on Stream Processing and modern database solutions [8]. Kleppmann describes a way to consume event logs into a set of different index, cache and database systems that can operate autonomously and in parallel of each other. In Chapter 3 we will see how the prototype was implemented for this project and how streaming data from a Kafka topic into different data stores helps us both implement and evaluate different databases within Yolean's domain.

While a more classical solution with a single RDBMS could have sufficed to respond with the set of initial records for continuous queries, these stream processing techniques allow for future development to more easily extend and alter the database solutions used within the system. However, we should keep in mind that this solution will add a bit of overhead in order to support this. Kafka, which can be simplified as a write-ahead log for our purposes, will stream updates into PostgreSQL which in turn will also update its own write-ahead log[1] before it writes any changes into the actual database table. Thus, in order to reduce the complexity in our own application code for the Live Server by leveraging several layers of third-party software originally used for persistence, we are sacrificing a bit of processing power.

Kholghi and Keyvanpour [6] summarize a couple of methods for indexing data streams. One indexing technique is implementing a *sliding window* where indexes only consider events within a certain time frame for indexing. Common for all in-

[1] `https://en.wikipedia.org/wiki/Write-ahead_logging`

dexing methods they mention is that they are designed to analyze temporal data like sensor readings. However, while this is very suitable for temporal data which can be disregarded later on, this does not fit Yolean's domain where all history of the data must be available in order to resolve the current state. Instead, Yolean's domain maps quite well to relational databases, which commits to storing your data until it is explicitly removed.

2.2 Persistent Data Structures

Indexing our event stream is not enough to be able to distribute update events to affected clients. Therefore, a partially Persistent Data Structure[2] is implemented by using the **Map** data structure from the library ImmutableJS[3], which, just as the name implies, is built for immutable operations on some common data structures. The in-memory representation of Yolean's data is constructed using hash-array mapped tries (HAMT), a variation of the more commonly known Hash Map, but with better worst-case complexities [3]. The in-memory representation of the data is then used to resolve which updates affect which queries by comparing the old and new states after the update is applied. Keeping a full state in memory is of course nothing one should do without hesitation. Hence, all benchmarks have been conducted on the whole system as opposed to just querying the databases. This way we try to incoroporate as many practical consequences as possible of the in-memory representation.

2.3 B-Trees

This project has evaluated how PostgreSQL can be used to answer queries on the data within Yolean's domain. The table we used to store our data was set up with separate secondary indexes for the few columns that we query against during our experiments. Chapter 4 describes these experiments in detail so we will instead focus on how PostgreSQL indexes were expected to help speed up queries within our prototype. PostgreSQL indexes are implemented as B-Trees (or Balanced Trees) and they have a search complexity of *O(log n)*[9].[4] Recalling the naive method used by Yolean today, which simply filters all records using a predicate function, we can expect a much greater performance when using PostgreSQL to answer our queries.

[2] `https://en.wikipedia.org/wiki/Persistent_data_structure`
[3] `https://facebook.github.io/immutable-js/`
[4] `https://en.wikipedia.org/wiki/B-tree`

2.4 Inverted Indexes

To evaluate a second tool utilizing a second data structure to help answer queries within the domain, we chose the search-engine Elasticsearch[5]. Elasticsearch is based on Lucene indexes which implement the inverted index data structure common in full-text search software [5]. While Inverted Indexes is able to perform full-text search (including not only exact matches on strings) we have not utilized this feature for our experiments. Instead we chose this search engine for comparison against PostgreSQL to get a wider view of how Yolean's domain can be queried. Using the Vector Space Model [6] Lucene Indexes should be able to outperform any naive search thanks to efficient algebraic calculations. However, one should note that these type of index structures are expensive to update[5]. Thus, Chapter 4 also includes update comparisons between PostgreSQL and Elasticsearch.

[5] `https://www.elastic.co/`
[6] `https://en.wikipedia.org/wiki/Vector_space_model`

3

Method

In broad strokes, we have implemented a new prototype for the Live Server that Yolean uses in their Visual Planning application. This chapter provides an overview of the Live Server implementation by describing its protocol as well as the defining components and design choices.

3.1 Live Server Implementation Overview

The Live Server's two main tasks are responding to continuous queries and distributing record changes for these queries. After a continuous query has successfully been registered, its client may send updates to the server. The Live Server persists all updates as *events* in Apache Kafka. This event log is then read in parallel by a number of independent index clients, which, backed by an already optimized and well-tested generic database solution can quickly respond to queries.

Sections 3.1.1 and 3.1.2 provide an overview of the protocol and its implementation respectively. Then, Section 3.1.3 describes the Index Client in detail and in Section 3.1.4 we will see how a persistent data structure lets us keep several different versions of the state in memory with a minimal memory impact.

3.1.1 Live Protocol

The implementation consists of a server component, the **Live Server**, and a client component, the **Live Client**. Clients can communicate with the Live Server through a simple *live protocol*. Figures 3.1, 3.2 and 3.3 display three different scenarios which together capture the different communication flows in this protocol.

3.1.1.1 Successful Communication

A successful run of the protocol is displayed in Figure 3.1. The client connects to the server with a continuous query (CQ). It receives the initial sets of records along

with the query *id*. It is with this id that the client is later allowed to modify the parameters of the query. This relates directly back to the Visual Planning use-case mentioned in Section 1.2 where clients need to be able to quickly update the application's subset of tasks to display different dates. Until the client closes the connection it is allowed to send updates to the server, and it will receive updates affecting its query to the server.

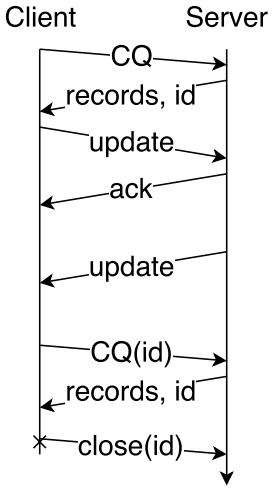

Figure 3.1: A successful run where a client connects to the server, sends an update, receives an update and updates the query itself with new parameters. Finally the connection is closed by the client.

3.1.1.2 Invalid Updates

As visualized in Figure 3.2, when a client sends an update, the server will verify that the update is valid before storing it and distributing it to other clients. If the update is deemed invalid, an error is sent back to the client, which is expected to reset its state by re-sending the initial query and its parameters, along with the query id it was assigned on connection.

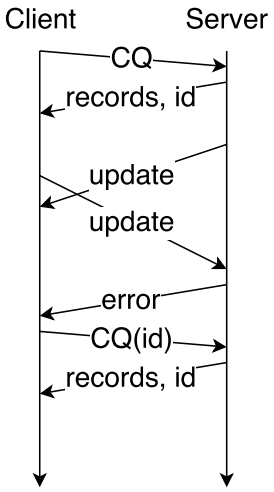

Figure 3.2: The client sends an update which gets rejected by the server as it has become out-dated. The client is now expected to restart the protocol.

3.1.1.3 Invalid Queries

Figure 3.3 displays a client which connects with an invalid query. This might happen for various reasons, for example upon programmer errors or when an index backend becomes unavailable. In this scenario, the protocol is immediately terminated with the error sent back to the client. The application running the client is expected to notify the user of this error.

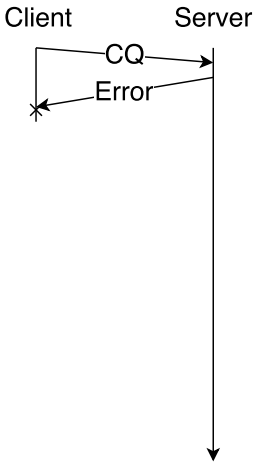

Figure 3.3: A client sends an invalid continuous query (or the index client inside the Live Server is currently unavailable).

3.1.2 Protocol Implementation

The protocol described in Section 3.1.1 is implemented in the Live Server through a few components that will be used in this section to describe the flow of data inside the service.

The most important component of this project is the *Index Client*, which is responsible for a single namespace of *filters* ("postgres/" or "elasticsearch/" for example) and will respond to queries and distribute updates to all Live Clients connected to it. Filters will be explained further along with Index Clients in 3.1.3. The second component is the *State Manager*, which supports the Index Clients by providing an in-memory representation of each state that is generated after merging an update event. For this to be feasible the State Manager implements a persistent data structure to keep the memory usage to a minimum. This section provides an overview of the internal components of the Live Server and then Sections 3.1.3 and 3.1.4 describe the Index Client and State Manager more in detail.

3.1.2.1 State Predicate Index Configuration

Figure 3.4 shows a Live Server with an index client named filter predicate that implements the naive solution where each continuous query is resolved by running the state through a predicate function. This is on par with the current implementation at Yolean. The figure displays the following steps:

1. State Predicate index client reads the initial state.
2. State Predicate index client sets up its own stream of update events.
3. Live Server listens to incoming requests, and receives a continuous query.
4. State Predicate stores the query so that it can be matched against future updates.
5. State Predicate asks the State Manager for the in-memory state and matches each record against the query.
6. Matching records are sent back to the Live Client.

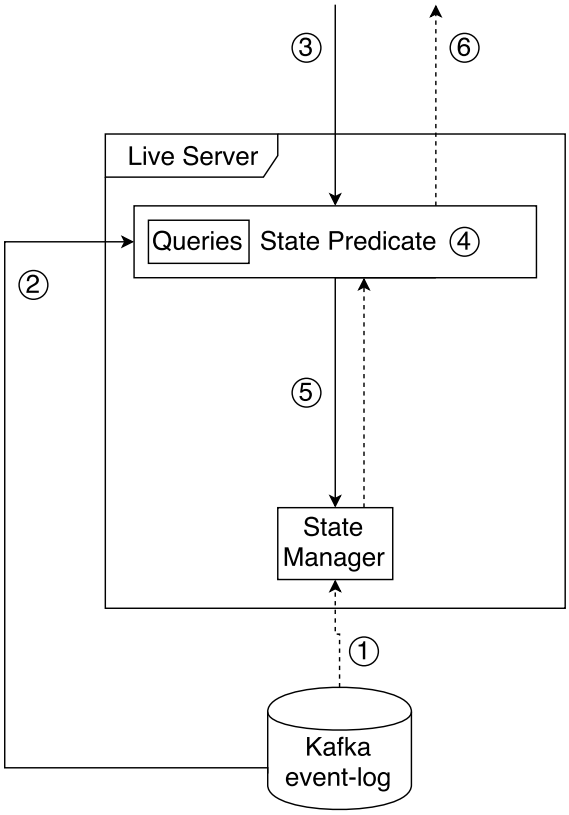

Figure 3.4: The Live Server configured with only the naive filter predicate index client. The figure encapsulates the steps needed for the server to respond with an initial set of records to an incoming query.

Incoming updates are handled in a like-wise manner, where the State Predicate index client matches incoming updates against the registered queries and updates the State Manager according to the record changes from the update. We will see a more detailed description of this later in Section 3.1.2.4. For now we are satisfied by concluding that by defining a few components within the Live Server we are able to satisfy Yolean's requirements on continuous queries.

In the following sections we will see how the State Predicate solution can be generalized into several index clients that can respond to queries with an initial set of records by using an index backend.

3.1.2.2 Initialization Overview

The Live Server process is initialized during an initialization phase as shown in Figure 3.5. The initialization phase consists of the following steps:

1. Existing update events are parsed and stored as an initial state in the State Manager.

2. Each Index Client reads the initial state.
3. Each Index Client resets its backend.
4. Each Index Client indexes the initial state.
5. Each Index Client sets up its own stream of update events.
6. The Live Server starts accepting incoming requests.

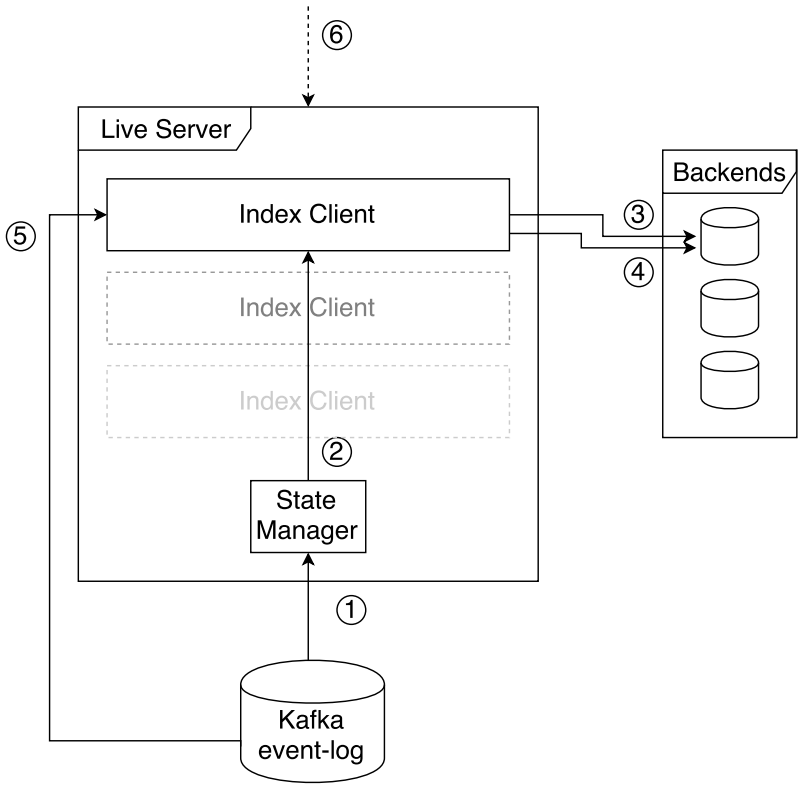

Figure 3.5: The Live Server initialization phase.

3.1.2.3 Query Overview

```
{
  name: postgres/vpInterval,
  params: {
    project: Project A,
    start: 2016-01-01,
    end: 2016-02-01
  }
}
```

Figure 3.6: Example Continuous Query in the Yolean Visual Planning domain. The query would, using the PostgreSQL Index Client, return all tasks for Project A scheduled between the dates 2016-01-01 and 2016-02-01. The Live Server is able to deduce information like return only task-type records from the name of the query.

Once the initialization phase has been completed and the Live Server is listening for incoming requests, Live Clients can register Continuous Queries (as exemplified by Figure 3.6) to the Live Server. Figure 3.7 visualizes the steps performed when registering a new continuous query.

1. Live Client registers a Continuous Query which is passed to the responsible Index Client.

2. The Index Client stores the query so that it can be matched against future updates.

3. The Index Client transforms the query name and parameters into a query for its backend and sends a request to its backend.

4. The Index Clients backend responds with the matching set of ids for the query.

5. The Index Client asks the State Manager for its in-memory state to resolve the set of ids into their corresponding records.

6. The records are sent back to the Live Client.

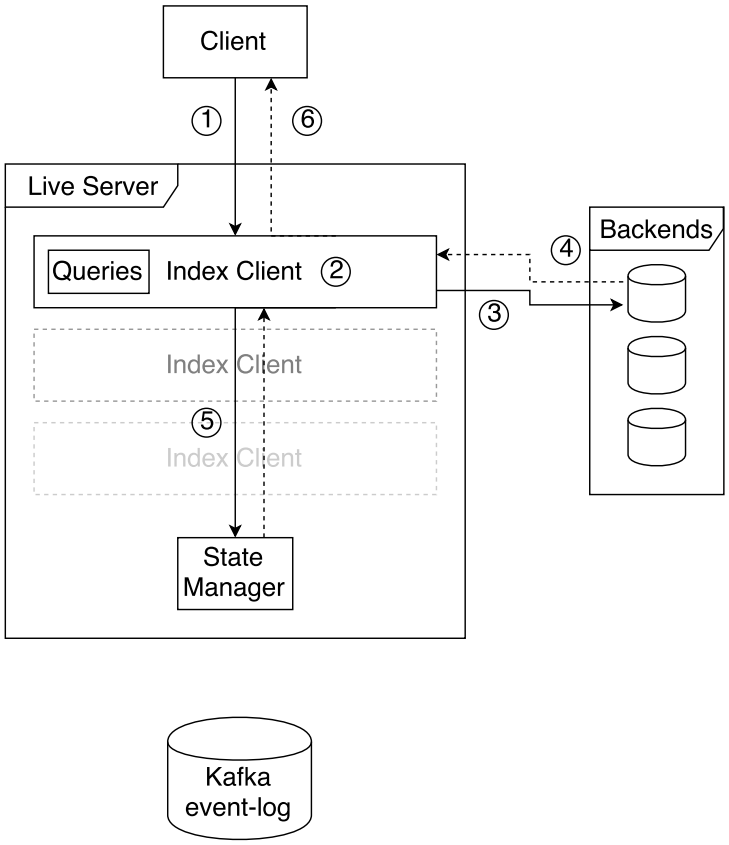

Figure 3.7: The data flow when querying the Live Server.

3.1.2.4 Update Overview

All connected Live Clients are able to send updates to the Live Server, which are distributed to all clients with a registered query that is affected by this update. This process is visualized in Figure 3.8 which contains these steps:

1. A Live Client sends an update to the Live Server.
2. The Live Server validates the update and writes it to the Kafka log (invalid updates are rejected with an error sent back).
3. Each Index Client reads the update through its Kafka stream.
4. Each Index Client decides whether the update affects its index and when needed the update is written to the Index Client's backend.
5. Each Index Client gets its current state (prior to the update) from the State Manager and uses this to resolve which queries are affected by the update.
6. Each Index Client commits to having read the update to the State Manager.

7. The update is distributed to all affected queries.

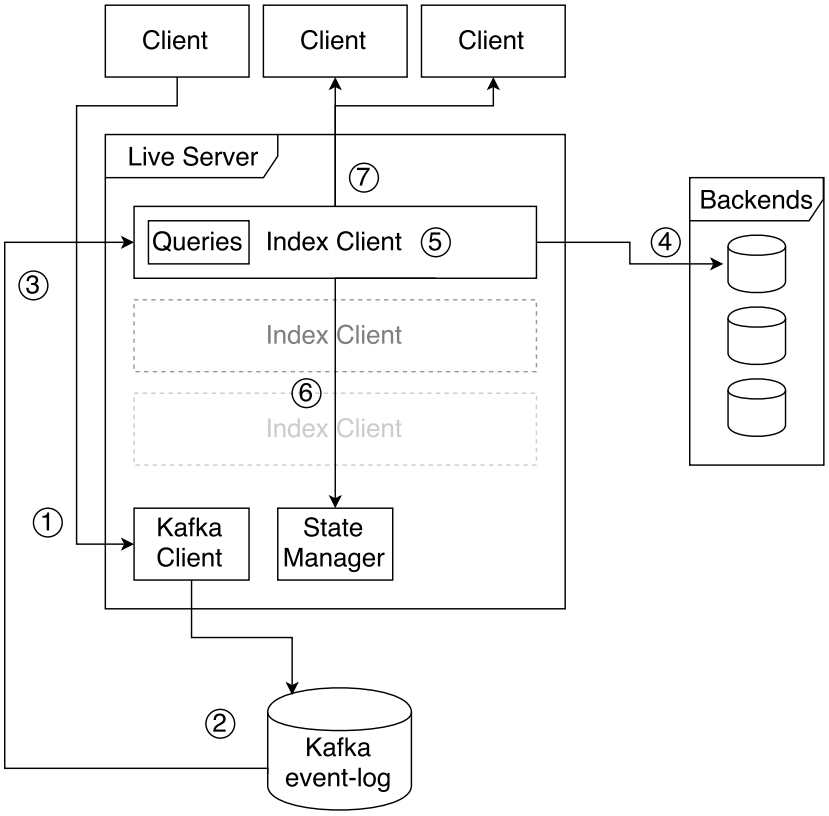

Figure 3.8: The data flow when sending an update to the Live Server.

Step 4 contains an important gotcha that we need to explain a bit further. Each Index Client is responsible for maintaining a state where no race conditions can result in a scenario where it has responded to a query assuming a different state than it last commited to the State Manager. For this proof of concept implementation a great simplification was made to all Index Clients, namely that updates are queued in the Live Server in such a way that there is only one outgoing request at a time to their backends. This way, we can safely know that the state that was commited last by the Index Client is also the state that its backend has been notified of. In order to combat this simplification we can instead simply implement these Index Client such that they ignore updates that don't affect their indexing. This works for our case as many attributes within Yolean's domain that change often are not needed when querying the data.

3.1.3 Index Clients

To formalize the relation between the Live Server and its indexes, we define a concept of Index Clients. Each Index Client can define several parametrized named **filters**, each of which is implemented to respond to a rather narrow set of continuous queries.

All filters expose a common API: a predicate function `matches(record)` which returns true iff the record should be included in the query; and `query()` which queries the index for all matching records by transforming the parameters to a query for its backend. It is then up to the developer of these filters to make sure that these two methods consistently match the same records. Thus $matches(record) \Leftrightarrow record \in query()$ should always hold.

The prototype uses the result of each registered query's `matches(model)` call to define whether changes should be distributed to the client or not. It does so by calling the function once with the old record (if any) and once with the new record (if any, as it might have been removed). Thus it is able to resolve individually for each registered query, whether the change was in fact an add, a change, or a remove event for each query individually.

We describe the implementation of each Index Client by looking at the Continuous Query example mentioned in Section 3.1.2.3 again:

```
{
  name: postgres/vpInterval,
  params: {
    project: Project A,
    start: 2016-01-01,
    end: 2016-02-01
  }
}
```

We will go through the three Index Clients implemented for this project and describe how they relate to their respective backends. We will also motivate why they are suitable candidates for resolving queries for Yolean's Visual Planning domain as formalized by Figure 1.2.

3.1.3.1 Predicate Function

In order to be able to evaluate the new index solutions against the legacy method of matching each record against the query's predicate function, the persistent data structure's state is used straight away without any backend supporting it. This Index Client responds to each query by running all records of the state through the filter's matches function. We see that this Index Client can be formalized as $query() = \{record \mid matches(record)\}$.

Due to NodeJS being single threaded, results against this index should be evaluated while bearing in mind the fact that each call to the query method will occupy the whole Live Server process during this time. There are of course workarounds for this issue, but in order to be able to evaluate a solution close to the legacy implementation of this service, no workarounds were implemented for this problem. Instead, as described in more detail in section 4.3, all metrics are collected by clients as opposed to on the server.

3.1.3.2 Relational Database – PostgreSQL

The postgres database was chosen as it is one of the most popular open-source relational databases.

The Index Client sets up a single table in postgres where it stores only a few columns, namely: `plandone` with type `date`, `project` with type `varchar(50)`, `owner` with type `varchar(30)` and a primary key `id` of type `uuid`. Together these columns suffice to answer the queries that we evaluated during the experiments described in Chapter 4.

In an attempt to optimize the table for reads and thus queries, internal PostgreSQL secondary indexes were created for all columns. As mentioned in Section 2.3 these secondary indexes should result in quicker queries against records in the table.

3.1.3.3 Inverse Index – Elasticsearch

Butcher et al. [5] promotes the Lucene search-engine as one of the most useful implementations of an inverse index. Apache Elasticsearch sits on top of the Lucene engine and provides a mature search API for its Javascript client. Thus it was chosen as the Inverse Index implementation for this project.

As Elasticsearch is built to support few large indexes rather than several small indexes[1], this Index Client creates a single Elasticsearch index that is updated with the fields *plandone*, *project*, *owner* and *id* for all records.

Elasticsearch can perform both string matching (`str1 == str2`) as well as full-text search (which is more geared towards user defined search strings than what this thesis wishes to support). This Index Client configures the Elasticsearch index to perform the first string comparison.

[1] `https://www.elastic.co/blog/index-vs-type`

3.1.4 State Manager

As Index Clients may temporarily become out of sync with each other, we need to keep track of which states that are still needed by an Index Client, and discard states when no Index Clients refers to it after an update. For each update that is consumed by the Index Clients, the State Manager will receive a `commitUpdate(indexName, offset, update)` call. This means that momentarily the Live Server will have to keep two states in memory until all Index Clients have caught up and written the update to their backend. This builds up towards a worst case where we might end up with different state references for each Index Client.

In order to keep the memory usage down while Index Clients are skewed between different states, the State Manager implements a partially persistent data structure using an immutable hash array mapped trie [3] from the ImmutableJS library. This lets all states diverge while only allocating memory for the difference between the states. Once the Index Clients have all caught up, the same state is shared between the Index Clients and all the rest are discarded.

4

Results

This chapter describes the three different experiments we have used to benchmark the Live Server prototype we describe in Chapter 3. The experiments focus on service initialization, query performance and update performance. Each experiment described below will define its own metrics, but so that we know what to expect they are:

- Service initialization time
- Query-response time
- Query throughput
- Update delay
- Update throughput

All three experiments share a common setup and common data input. Section 4.2 describes how the datasets are generated from a real-world set of data from Yolean's domain and Section 4.3 will give a general description of the experiment setup. But to start with we take a look at the hardware and platform we ran the experiments on.

4.1 Environment Specification

All experiments were run on `OS X El Capitan 10.11.1` using a Macbook Pro Retina Laptop with the following specification:

```
MacBook Pro
  Model Identifier: MacBookPro10,1
  Processor Name: Intel Core i7
  Processor Speed:  2,6 GHz
  Number of Processors: 1
  Total Number of Cores:  4
  L2 Cache (per Core):  256 KB
  L3 Cache: 6 MB
  Memory: 16 GB
```

The test framework was implemented by utilizing a containerization tool called Docker.[1] One of Docker's main features is called Docker Compose, which helps you set up and run multiple applications that depend on each other. This procedure also helps when you need reproducible results, as you can simply restart a set of Docker containers through Docker Compose and expect the same result. This reproducibility makes Docker a good platform for our test framework.

The docker version used for these benchmarks was Docker for Mac Version `1.12.1-beta25`. Docker was allowed to use a maximum of `4 CPUs` and `8GB RAM`.

4.2 Dataset Generation

All experiments were run over the same set of initial datasets. We wanted to evaluate the prototype over a growing set of records, so the datasets were generated with the different target sizes $n = \{10000, 15000, 20000, ..., 100000\}$.

The datasets were meant to simulate real-world usage from Yolean's domain. Thus, they were generated from an anonymized set of data from one of Yolean's customers. We will refer to this set of data as *the source* from now on. The source contained $\sim 30,000$ unique records spread over ~ 130 different projects. We wanted to analyze how our implementation of the Live Server behaved over a growing number of records, so we generated datasets for all values of n by picking randomly n records out of the $30,000$. Each sampled record was assigned a new unique id attribute to avoid record duplicates within the generated set. Further, we wanted to maintain the original distribution of records per project within the generated datasets. This was accomplished by scaling the number of projects by a factor of $max(1, \frac{n}{30,000})$.

[1] `https://www.docker.com/`

As we can see in Figure 4.1 our sampling method lets us scale the number of records and projects while keeping the same distribution of number of records per project. Only about 55% of all records actually belong to a project. The remaining 45% are assigned only to a person and not to a project. We still include them in our dataset to model our experiments closer to practice, but they will never be queried for in the experiments.

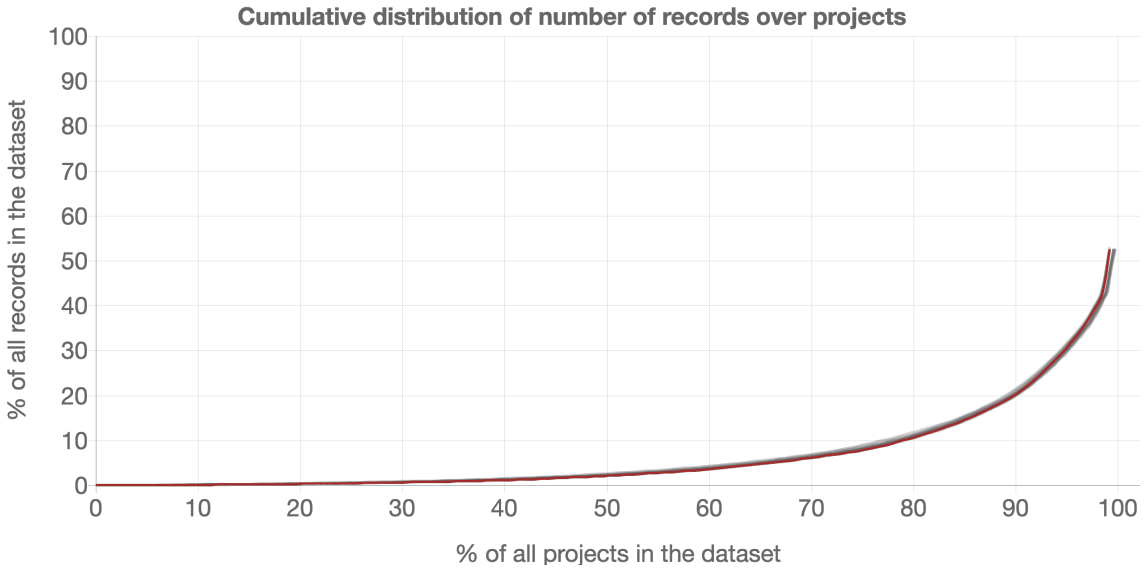

Figure 4.1: Records-per-Project Distribution. The distribution of number of records per project with the source dataset represented by the brown line, and all generated datasets $\{10000, 15000, 20000, ..., 100000\}$ in light grey lines. As we can see the distribution is very similar between all datasets. Most projects contain only a few records: roughly, 80% of the project together own only 20% of the project-associated records.

Note that our datasets are modelling only snapshots of an evolving database. We did not harvest any modification or deletion actions from real-world examples.

4.3 Experiments

In this section we describe the common setup for all three of our experiments. After that, each experiment is described in detail in Section 4.3.1, Section 4.3.2 and Section 4.3.3 respectively. In order to reduce the number of anomalies in our data, each experiment was repeated three times. Thus, all metrics shown in the graphs in this chapter are averages from these three samples.

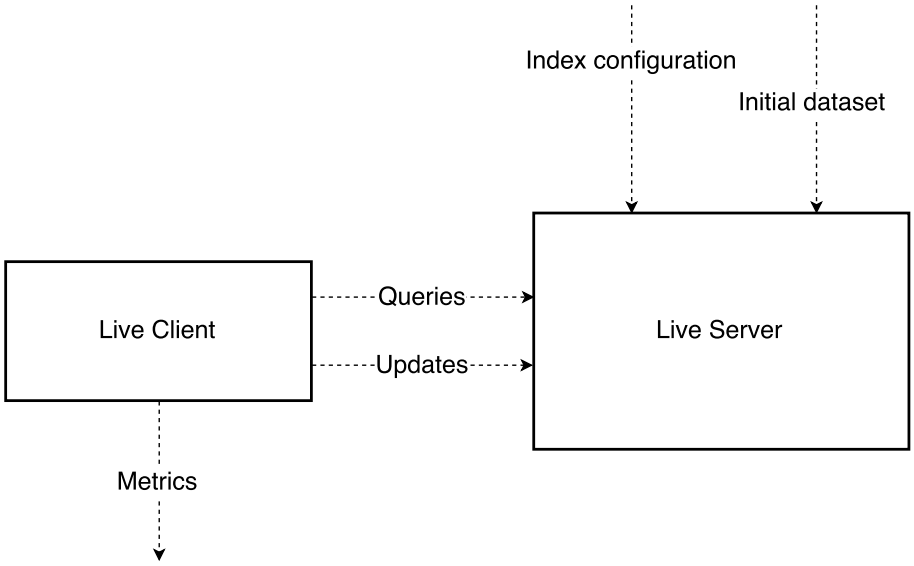

Figure 4.2: The setup of an experiment, where the client outputs its results into Elasticsearch so that it can be queried and visualized in a later stage.

Each experiment consists of a Live Client and a Live Server as can be seen in Figure 4.2. The Live Server is configured to read an initial dataset into the service. The Live Client performs a set of queries and updates depending on the experiment. All results are logged by the client, which means that network latency and message-parsing will also be included in the observed results. However, all experiments were run locally with the client on the same machine as the server. Therefore, network- and message-parsing latencies are considered constant and small.

In Section 3.1.3 we defined the concept of a filter as a structured way for Index Clients to map domain specific continuous queries into a database query (for providing the initial response) and a predicate function (for resolving whether an update affects the query or not). For our experiments a filter named "vpInterval" was implemented for all Index Clients. The filter accepts the parameters "project" and interval- "start" and "end". Throughout all experiments the client thus registers queries against either "postgres/vpInterval", "elasticsearch/vpInterval" or "state-predicate/vpInterval" (which is the legacy method described in Section 3.1.3.1).

4.3.1 Experiment 1 – Service Initialization

The service initialization phase was described in Section 3.1.2.2. As we can recall from there, all events that have been accumulated over time will be streamed once from the event log in Kafka to build an initial state of the data, which in turn is indexed by all configured Index Clients. This experiment will compare how different configurations of Index Clients and number of records in the initial dataset affects the *service initialization time*. The service initialization time metric is measured

from the time it takes for the Live Server process to start until it starts listening to incoming requests.

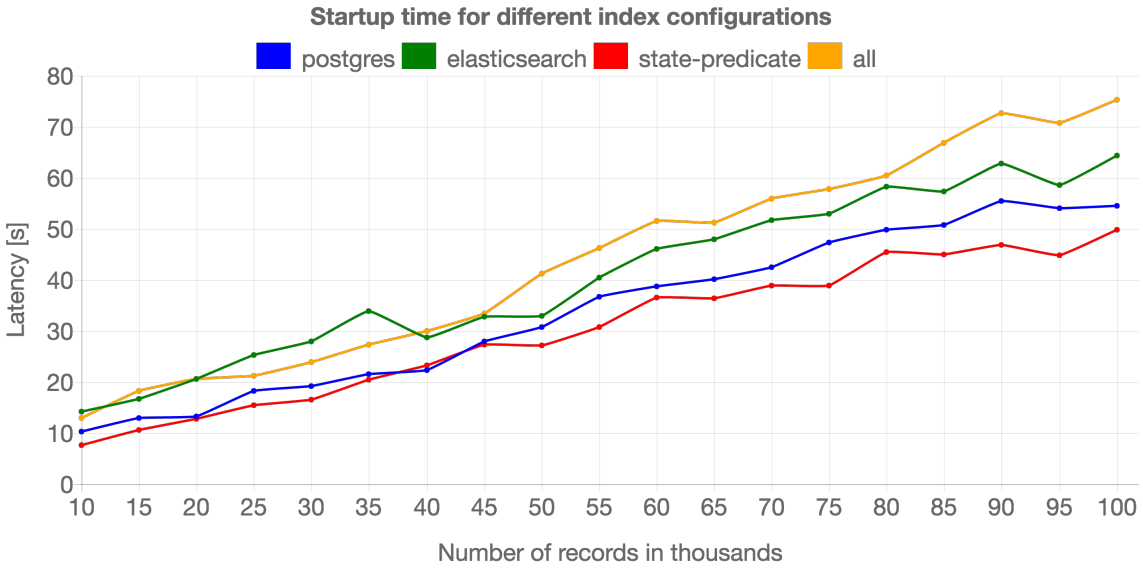

Figure 4.3: Service Initialization Time. Each line describes a configuration of enabled Index Clients for the service. *state-predicate* performs no indexing, *elasticsearch* indexes the dataset to Elasticsearch, *postgres* indexes the dataset to PostgreSQL and *all* indexes the dataset to both Elasticsearch and PostgreSQL.

From Figure 4.3 we can see that all configurations adhere to a linear increase in initialization time. The configuration called "state-predicate" does not perform any indexing (as was described in Section 3.1.2.1), and it is therefore apparent that it requires the least initialization time for all datasets. We can also see that the configuration called "all", which writes to both Elasticsearch and PostgreSQL, takes a bit more time to complete than the configurations that write to only one of these databases.

Let $t_{configuration}$ denote the lines from Figure 4.3. Then, when comparing these configurations we would expect $t_{all} = t_{elasticsearch} + t_{postgres} - t_{state-predicate}$ to hold assymptotically. Roughly, at $n = 100,000$ the right-hand side of this equation equals $65 + 55 - 50 = 70$ seconds, which is quite close to the actual result of the left-hand side. Further, we note that for $20,000 < n < 35,000$ we have $t_{elasticsearch} > t_{all}$, which is unexpected. This can probably be explained by the fact that initializing the Elasticsearch service takes a lot of time, and for the Live Server to discover it can often require several connect retries.

We note here that the datasets were generated as a sequence of updates containing only add operations. Thus, in real-world usage, the initialization time might be higher than what we see here, as a real-world usage would also generate a lot of updates that modify existing records. As the state-predicate configuration (which does not perform any indexing) still takes quite some time to initialize we assume

that reading the event log and constructing the initial state takes up a lot of the initialization time for all configurations.

4.3.2 Experiment 2 – Query Benchmark

The primary concern with the current implementation of the Live Server at Yolean is how performant it is at resolving initial responses as the number of records increases. Improving or at least analyzing *query-response time* and also *query-throughput* was thus one of the highest priorities for this project. Thus, these two metrics were benchmarked during the second experiment.

Query-response times were measured as the time it took between registering the query and receiving the set of records. The average was then calculated for 100 queries registered from the same client.

Query throughput was measured as the number of queries the service could respond to per second. This was accomplished by registering 100 queries in sequence on a single client and then measuring the time passed until all queries had received a response.

These metrics were benchmarked over which Index Client that was included in the configuration as well as how many records there were in the initial dataset.

The parameter values (project, interval-start and interval-end) for each continuous query were sampled with a normal distribution from the dataset, which means that projects containing a large number of records would have been queried more often.

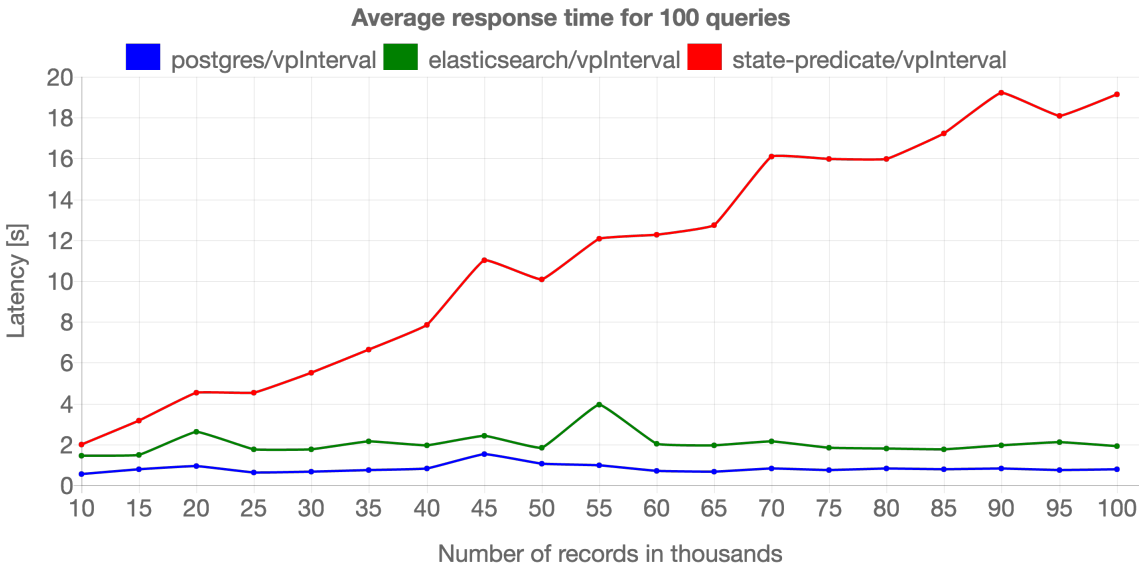

Figure 4.4: Query Response Time over a growing dataset. Each line describes which Index Client was queried. The other Index Clients were disabled (received no updates or queries).

The goal of this project was to implement a new prototype of the Live Server which would still adhere to Yolean's requirements on Continuous Queries, but allow for optimizations on the initial responses. Figure 4.4 shows a clear improvement when third party software is used to help with these initial responses. The legacy implementation *state-predicate* which naively filters all records in the dataset for each query has, as expected, a linear relation between response time and the size of the dataset, whereas both *postgres* and *elasticsearch* have an almost constant response time in comparison. We should keep in mind however that the data volumes for these benchmarks and also Yolean's domain are still very small in comparison to what modern databases are built to handle. As we can recall from Chapter 2, these databases use indexes implemented as B-Trees and Inverted Indexes, which means these lines would likely resemble logarithmic curves for larger values on n.

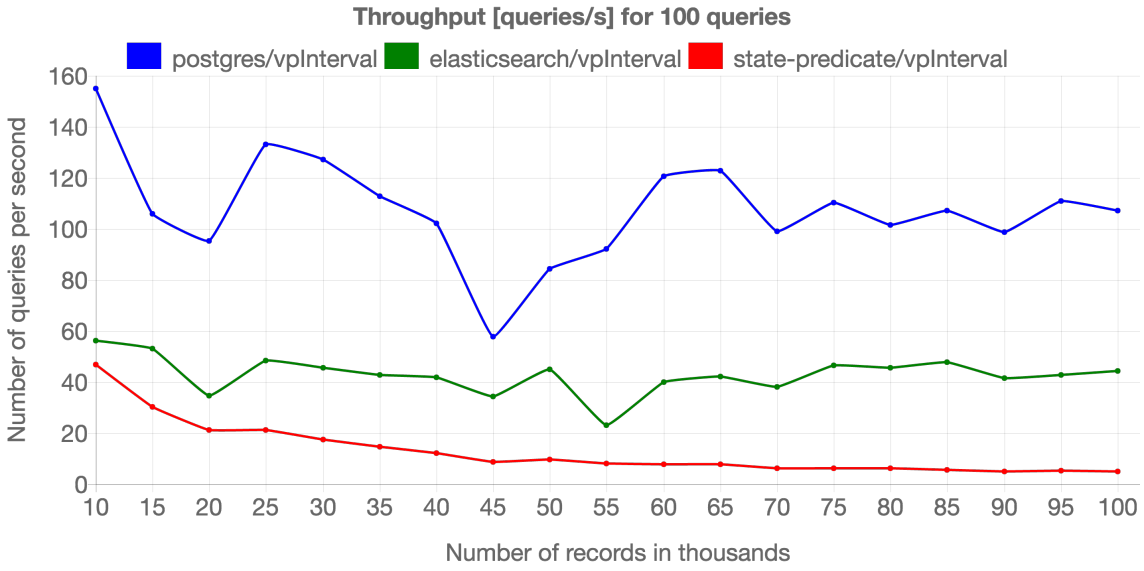

Figure 4.5: Query Throughput over a growing dataset. Each line describes which Index Client was queried. The other Index Clients were disabled (received no updates or queries).

Looking at Figure 4.5, we expect the line for "state-predicate" to match a hyperbolic curve ($\frac{1}{n}$), which is not far from the actual outcome here. Further, for both "postgres" and "elasticsearch" we would expect them to be almost constant, which apart from some noise, they are.

In this experiment we notice anomalies for $n = \{45000, 55000, 70000, 90000\}$. They are represented by spikes in Figure 4.4 and dips in Figure 4.5. All configurations of the service were run independent of each other, which might explain why we see anomalies for different configurations at different values of n. To be certain that these anomalies are not actually due to some problems with the implementation, further experiments would need to be conducted. For example, repeating the experiment more times in a more stable and controlled environment would allow us to compute standard deviations and confidence intervals for our results. However,

for this thesis we settle with being able to show the difference in query performance between the naive configuration and the database-supported configurations.

Figure 4.4 and Figure 4.5 tell us that PostgreSQL outperforms Elasticsearch for this query. If we consider the type of queries described for Yolean's domain in Section 1.1 we see that they closely map to the SQL queries that PostgreSQL is implemented for. We also note that Elasticsearch, which is first and foremost a Search Engine, supports complex full text search queries, which is not at all utilized by the queries in these benchmarks.

4.3.3 Experiment 3 – Update Benchmark

As was described in Section 3.1.2.4, upon receiving a new update from the event log, our implementation of the Live Server waits for the Index Client to write the changes from the update to its backend before it sends updates to affected queries. It does so in order to be able to guarantee that the Index Client can keep its backend in sync with the in-memory state it has committed to. This limits each Index Client to only handle one update at once. With that in mind, we wish to benchmark the *update delay* and *update throughput*. The parameters that we vary for this experiment are:

- The total number of records in the dataset
- The Index Client used by the configuration
- Whether the updates were indexed by the Index Client or not

Update delay was measured as the time it took for an update sent from one client to all affected queries' clients.

Throughput was measured as the number of operations the service could distribute per second. Much like the query throughput benchmarks, this was achieved by registering a set of queries, and after receiving the initial data for all queries, a set of operations was sent to the service in sequence and we waited for all affected queries to receive all updates.

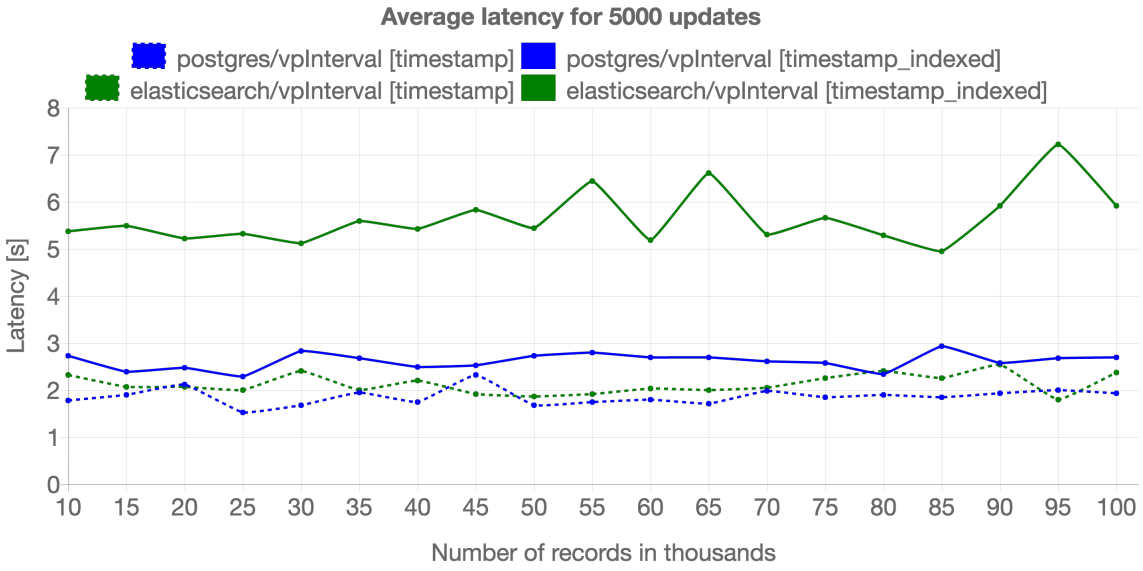

Figure 4.6: Update Delay over a growing dataset. Dotted lines represent configurations where updates are not written to any Index Client's backend, while filled lines represent configurations where they are.

The Live Server prototype proposed in this thesis is biased towards read efficiency, rather than write efficiency (as we can conclude from the Update Overview in Section 3.1.2.4). This becomes apparent when looking at Figure 4.6 and Figure 4.7 where we can see how the Elasticsearch Index Client becomes slower when it has to write updates into its Elasticsearch backend before distributing incoming updates. We note however that dataset size does not notably affect either update delay or update throughput. This is of course both an expected and desired behaviour of any database solution.

A final note on the results, especially apparent in Figure 4.6, is that even at a dataset containing only $10,000$ records, all configurations for the Live Server take about two full seconds before they have distributed all update events. So we should keep in mind that these benchmarks represent spike load on the service, and all metrics include computation times of a single client that registers 100 queries. Thus, when looking at these graphs, we are mostly interested in the comparison between the different Index Client implementations over a growing number of records in the dataset.

Looking at Figure 4.6 and Figure 4.7 we see that writing to PostgreSQL has much smaller impact on performance compared to when writing to Elasticsearch. This result can be explained by reflecting over how Elasticsearch is built for seldom and bulk writes and when we force it to update its internal index with many small updates it is forced to update its index after every write. PostgreSQL on the other hand can, even when configured to use secondary indexes for our columns, choose to defer updating the index, and answer queries using its table instead.

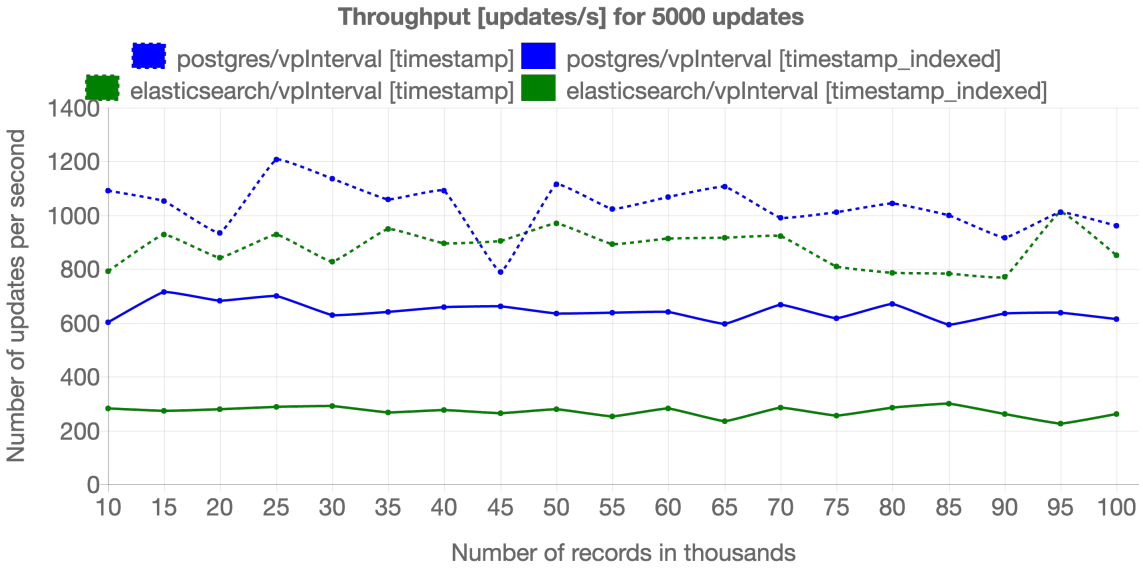

Figure 4.7: Update Throughput. Dotted lines represent configurations where updates are not written to any Index Client's backend, while filled lines represent configurations where they are.

5

Discussion

In this chapter we discuss the results presented in Chapter 4. We will sum up the results and relate them to the expectations and requirements of Yolean as well as highlight some limitations on the current implementation and how they could be improved by future work.

The results presented clearly show that this new implementation outperforms Yolean's naive solution which is in production today. It does so by using a simple method of querying well-tested database solutions like PostgreSQL on a subset of all attributes within Yolean's domain, while keeping the full dataset in memory. With this in-memory representation the prototype is then able to distribute the minimal set of updates that affect each registered query.

Currently, Yolean's largest dataset in production contains $30,000$ records which means that Yolean could decrease the query time down to about 14% of today's query response times when the site is under heavy load. Even more interesting and reassuring for Yolean is the low decrease in performance over a growing set of records, with one exeption perhaps: the initialization time.

5.1 Reducing Initialization Time via Data Snapshots

The initialization process was implemented in order to allow bulk writes to both PostgreSQL and Elasticsearch which for many database solutions is more efficient than writing each record separately. With the results shown in Figure 4.3 we may conclude that a major part of the initialization process is reading all the update events from the Kafka log and constructing the initial state for the data. Reducing the initialization time could be accomplished by snapshotting the data at regular intervals, thus reducing the number of events that have to be parsed when the service starts [7].

5.2 Removing the In-Memory Dependency

The most unconventional design choice with the proposed solution is the dependency on keeping the full state in memory. While the idea is motivated by the fact that Yolean's domain does not work with huge data volumes and that RAM is considered cheap these days, this design decision imposes limitations on the system. The persistent data-structure described in Section 3.1.4 is implemented using only synchronous calls and performs efficient lookups for all records in $O(1)$ time using the ImmutableJS library. While this method was shown to work for a dataset about 10 times larger than Yolean's current situation, it does so by sequencing all writes to each Index Client in order to be able to handle keeping all possible states in memory. Instead, an exernal immutable object store such as Interplanetary File System (IPFS) could be used to remove the in-memory layer in the Live Server prototype. IPFS implements a data structure that allows for efficient lookup but most importantly it does so on immutable data [4]. Hence, it could be used to resolve the state of a record at a certain version in our system. For convenience and to improve performance on common usage something simple as a Least Recently Used-Cache could be implemented to keep the most sought-for records in memory.

5.3 Internalizing all Indexes

In contrast to Section 5.2, under the assumption that RAM will continue to be a cheap resource, we could remove the usage of external database solutions for indexing the records. Instead B-Trees and Inverted Indexes could be used internally by the Live Server. This would reduce the overhead with running external services in several ways (less CPU and RAM overhead, reduced latency from communicating with HTTP between services). Downsides that would speak for going in another direction is however that you would risk building a larger and more monolithic solution, with fewer swappable components.

6

Conclusion

We have implemented a proof of concept prototype for a service which resolves what we defined as Domain Specific Continuous Queries in Chapter 1. A Continuous Query is a query that along with the initial response of matching records, also provides a stream of all updates affecting the query's response. The implementation makes a few simplifications such as keeping an in-memory representation of the full dataset and sequencing database writes. The prototype was benchmarked against a growing set of records and metrics for querying and updating the dataset were measured. The naive state-predicate method matching all records against a predicate function was greatly outperformed by the implementation we proposed which uses relational DBMS and Search Engine solutions to respond with the initial set of records. The prototype is useful within the domain of Yolean's VP application and during our experiments we observed query times 7 times faster than the original approach. Yolean is thus on their way to adopt the solution in production. However, for datasets with millions of records the in-memory representation would likely have to be removed or at least support asynchronous writes.

Bibliography

[1] A. Arasu et al. *STREAM: The Stanford Data Stream Management System*. Technical Report 2004-20. Stanford InfoLab, 2004. URL: http://ilpubs.stanford.edu:8090/641/.

[2] Shivnath Babu and Jennifer Widom. "Continuous Queries over Data Streams". In: *SIGMOD Record* (2001), pp. 109–120. URL: http://doi.acm.org/10.1145/603867.603884.

[3] Phil Bagwell. "Ideal Hash Trees". In: *Es Grands Champs* 1195 (2001). URL: http://lampwww.epfl.ch/papers/idealhashtrees.pdf.

[4] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: *CoRR* abs/1407.3561 (2014). URL: http://arxiv.org/abs/1407.3561.

[5] Stefan Büttcher, Charles Clarke, and Gordon V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010.

[6] Mahnoosh Kholghi and Mohammad Reza Keyvanpour. "Comparative Evaluation of Data Stream Indexing Models". In: *CoRR* abs/1208.0684 (2012). URL: http://arxiv.org/abs/1208.0684.

[7] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2016. ISBN: 978-1-4493-7332-0.

[8] Martin Kleppmann. *Making Sense of Stream Processing*. O'Reilly, 2016.

[9] Gregory Smith. *PostgreSQL 9.0 High Performance*. Packt Publishing, 2010.