UiO : Department of Informatics
University of Oslo

Identifying Higher-Level Semantics in Football Event Data

Eigil Håkedal Skjæveland

Master's Thesis Autumn 2017

Identifying Higher-Level Semantics in Football Event Data

Eigil Håkedal Skjæveland

July 31, 2017

Abstract

Quantitative analysis is used in an increasing amount to get an edge over competitors in football, and one of the largest providers of data is Opta Sports. In particular, they provide "event data", which are annotations of every event in a match.

In this thesis, we ask whether it is possible to extract higher-level events from the Opta event data, and whether it is possible to do efficiently in a database. We also examine the performance characteristics of some options of mapping the XML data to databases, and querying it, to find which of the options is the most efficient for storing and querying the Opta data in general.

We find that it is indeed possible to extract higher level-events from the data, and in particular find that a relational approach using recursive common table expressions is an efficient way of doing it.

We further find the most efficient of the evaluated options for storing and querying our data to be a relational approach, using PostgreSQL's JSONB column type for storing sparse attributes.

Acknowledgments

I would like to thank my supervisors, Pål Halvorsen and Kenneth Wilsgård, for their guidance and for the opportunity to work them. Further, I want to thank the Norwegian Center of Football Excellence for allowing me to use their data.

I would also like to thank my family for their support, and especially my girlfriend, Silje, for her helpful advice and for her patience.

Eigil H. Skjæveland
Oslo, Norway
July, 2017

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Quantitative analysis has become an important part of the football analysis ecosystem the last decade. Football has become more competitive, and better tools are available now than were before. By using novel player tracking technology, or by manual collection, data from matches are available in larger volumes and at an increasing granularity.

Such detailed data has a wide range of possible uses. It may be used by trainers and team analysts to improve their player training and team tactics. But it is also used by bookmakers for setting odds, and even by gamblers to reduce their risk and increase earnings.

Some of the most detailed data available today is *match event data*, which are annotations of every on-ball event throughout a match. One supplier of such data is *Opta Sports*, which, through an agreement with the *Norwegian Center of Football Excellence*, has supplied us with data from the Norwegian Tippeligaen, the Europa League and Champions League.

For producing match event data, Opta employs two operators covering a team each, later checked by a third. The data is then distributed in *feeds* with different focus and levels of detail.

This kind of data can be used to create new and novel statistics, but the more novel the statistic, the more computationally demanding the query becomes.

1.2 Problem Definition

The overall goal of this thesis is to aid the *Norwegian Center of Football Excellence* in using the Opta event data efficiently for analysis, which could ease their work in helping Norwegian football teams succeed abroad and at home. Opta data is already beginning to be used to locate simple events for match analysis. It could relieve analysts of considerable amounts of manual classification work if it is possible to also extract higher-level semantics, that

is situations defined by patterns of events, in matches using the event data provided by Opta.

Because of the potential scale of the data, performance will also be an issue. When working with data from several seasons of a competition, and possibly many competitions, the amount of data will be in the order of tens of gigabytes. Querying data on this scale is usually most practically done with some sort of database system.

Therefore, the right choice and use of database tools will be essential for computing novel statistics to gain new insights in reasonable time.

In summary, we will try to answer the following questions:

1. Is it possible to extract higher-level events from the Opta event data, and can it be done efficiently?

2. How should one map the Opta event data to a database to get the most performant querying?

1.3 Limitations

While the overall goal of this thesis is to find the most efficient way to represent Opta data, we cannot test every database system or every schema, so the scope is limited to the databases and schemas outlined in section 1.6.

Furthermore, the results are only applicable to our benchmark workload, and while we chose the benchmark queries specifically to try to emulate a probable real-world workload, it will realistically only cover a small subset of possible workloads. Specifically, the benchmarking is geared towards offline analysis, and we concern ourselves only with read operations. All data is assumed to have been batch loaded prior to the benchmarks.

The result of the thesis will likely only be of use as a guideline to developers for interfacing with the Opta data. It will likely be of little use to non-developers.

1.4 Research Method

This thesis will follow the *design paradigm* proposed by the *Association for Computing Machinery's (ACM) Task Force on the Core of Computer Science* [1]. We will design a system for benchmarking our database options and querying our data, then implement the system and database schemas, then use the benchmarking system to measure our database options, and lastly compare our results.

1.5 Main Contributions

Through this work, we provide insight into the performance and characteristics of different ways to represent Opta XML data in database systems. We develop a benchmark, and test several approaches to storing and querying Opta's event data.

We explore whether it is possible to extract higher-level semantics from the event data, and we evaluate approaches to doing so efficiently.

In section 1.2, we asked two questions which we have answered as follows:

1. *Is it possible to extract higher-level events from the Opta event data, and can it be done efficiently?*
   Yes, we have found that this is possible. We have constructed queries, both for a relational and a graph based database, to extract two types of situations that are characterized by a certain pattern of events. Firstly, we have shown that we can identify compound events of a constant length, and in addition we found that it was possible to identify compound events of unbounded length. We have also shown that it can be done efficiently with the use of Recursive Common Table Expressions.

2. *How should one map the Opta event data to a database to get the most performant querying?*
   We found that using a relational approach was the more performant than using a graph based approach. Specifically, we found that using the JSONB column type of PostgreSQL was a very efficient way of storing the sparse attributes that are "qualifiers" in the opta event data.

1.6 Outline

This thesis consists of 7 chapters, and 2 appendices:

Chapter 2: Background

We begin by giving an overview of the current state of football analytics, before detailing the structure of the Opta data. Then we look at different types of database systems, and ways of measuring their performance, before we look at some previous work in the field.

Chapter 3: Experiment Design

In chapter 3, we describe our benchmarking system, and design our benchmark queries. Then we look at which metrics our benchmark should measure, and list our test environment.

Chapter 4: Relational Approach

Chapter 4 is where we benchmark our relational database, PostgreSQL. We design two schemas, then accompanying queries, before benchmarking them and analyzing the results.

Chapter 5: Graph Approach

In chapter 5, we benchmark our graph database, Neo4j. Like in the previous chapter, we design the schema and accompanying queries, then benchmark them and analyze the results.

Chapter 6: Comparison

In this chapter, we compare the results of the graph approach with the best schema from the relational approach.

Chapter 7: Conclusion

Finally, in chapter 7, we conclude the thesis by summarizing our findings and listing some possible future work.

Chapter 2

Background

This chapter will present the state of football analytics leading to the need for performant database solutions. Further, it will provide a thorough introduction to the format of the Opta XML match event data and common approaches for querying XML data. Then it will discuss approaches to benchmarking databases and database systems, before providing an overview of previous relevant work.

2.1 The Current State of Football Analytics

Football in general, and Norwegian football specifically, has evolved from an amateur pastime to a big-business, professional sport through the last half century. TV advertising ensure that teams have billion-euro budgets, but also the stakes that come with it. Fortunes can be made in the sport now, both by the participants and, owing to the sport's increased exposure, also by bookmakers and gamblers.

Football is a relatively unstructured sport, so detailed statistics and data-based analysis has not evolved as naturally as in more structured sports, like baseball, where events are easily defined and counted, and thorough use of statistics already has proven success [2].

In recent years, however, increased data availability and the sport's ballooning economy has brought about many innovations in the field. Today conferences are held [3][4], and new ideas like Expected Goals[5] and using machine learning to classify playing styles [6] are emerging fast.

Because of the game's nature, it is hard to analyze situations without having a lot of data. For example, evaluating a player's action in a situation is not only dependent on easily observable metrics like the current score and the match clock, but also on the player's location on the pitch and, even more complicating, the locations and movements of all of the *other* players on the pitch. E.g. a player's actions may be limited by an opposing player approaching, and his passing options are defined by the locations of his teammates.

Opta records only the location of *events*, and their definition of events is usually limited to events involving the ball. As such, we only know the location of the event, and therefore only the location of the players involved in the event.

Opta is limited by the manual data collection, but recent developments have been made to automate player tracking. Video analysis has been leveraged to track players through camera arrays through systems like Stats LLC's SportVU [7], but although these systems are steadily improving, the nature of video analysis means they are still inaccurate and computationally demanding. Bagadus [8][9][10] and its ZXY Sport Tracking subsystem [11] aims to remove these inaccuracies by using wearable sensors to track the players. The obvious drawback to this is its limited prevalence, as this is a system that requires physical setup in each stadium.

Opta's data is reliant only on a regular broadcast feed of each match it covers, and in addition to event locations also includes the *types* of events that occurred. So although analysis requiring complete player tracking remains out of bounds for the Opta event data, their event classification makes the data an indispensable source for football analysts.

2.2 Opta Data Structure

This section gives an overview of the relevant parts and structure of the Opta data. The data is distributed as XML [12] files grouped into *feeds*, and each subsection below will explain each of the relevant feeds, and the significance of each XML element and some important attributes.

The XML files make heavy use of numerical values which have to be cross-referenced with proprietary documentation [13] provided by Opta to have any meaning.

2.2.1 The F24 Event Feed

The F24 event feed is the most detailed of the feeds. It contains information about every event in a match.

Games

The most basic grouping in the event feed is the `Game` element, of which each file contain only one. Its attributes hold most of the match metadata, and the element itself contains all the events. A short excerpt can be seen in listing 2.1.

Events

The most interesting part of the feed is the events, of which an example is provided in listing 2.2. An event will always have one of currently 70 possible

Listing 2.1: The opening Game tag from the 2016 Champions League final

```
<Game id="851530" away_team_id="175" away_team_name="Atlé
    tico de Madrid" competition_id="5" competition_name="
    Champions League" game_date="2016-05-28T19:45:00"
    home_team_id="186" home_team_name="Real Madrid" matchday
    ="13" period_1_start="2016-05-28T19:50:04"
    period_2_start="2016-05-28T20:53:08" period_3_start
    ="2016-05-28T21:47:22" period_4_start="2016-05-28T22
    :06:21" period_5_start="2016-05-28T22:29:19" season_id
    ="2015" season_name="Season 2015/2016">
```

type_ids to signal its event type. It will also contain the time at which the event occurred relative to the period's kick-off and the time and date it was recorded by Opta, what player was involved (if any), and the x and y coordinates on the pitch. It will often also contain several sub-elements, which will be introduced in the next section.

Listing 2.2: The opening tag of a sample event

```
<Event id="33868766" event_id="4" type_id="5" period_id="1"
     min="0" sec="8" player_id="19927" team_id="186" outcome
    ="0" x="69.8" y="101.8" timestamp="2016-05-28T19
    :50:13.199" last_modified="2016-05-28T20:06:50" version
    ="1464462410263">
```

Events are recorded for every touch of the ball, substitutions, passes, goals, delays, formation changes, out of play balls, and so on. Each game will contain an average of about 1700 events.

Qualifiers

Qualifiers are sub-elements added to events to provide more detail. For a pass, the event will contain information like the coordinates of the passer. Additionally qualifiers will be associated with the event to show details like the intended recipient of the pass and his coordinates, the angle of the pass relative to the direction of play, whether the pass was chipped, or if it was an assist for a goal. There are currently 292 possible types of qualifiers, some of which may signify different things depending on the type of their parent event.

Listing 2.3: A sample qualifier

```
<Q id="495608885" qualifier_id="56" value="Left" />
```

An example of how qualifiers are applied to events in a real match, can be seen in listing 2.5, which shows a *foul* event (`type_id="4"`) followed by a *card* event (`type_id="17"`) with a *yellow* qualifier (`qualifier_id="31"`).

Listing 2.4: The beginning of a match in an F24 event feed

```
1  <Game id="851530" away_team_id="175" away_team_name="Atlético
       de Madrid" competition_id="5" competition_name="Champions
       League" game_date="2016-05-28T19:45:00" home_team_id
      ="186" home_team_name="Real Madrid" matchday="13"
      period_1_start="2016-05-28T19:50:04" period_2_start
      ="2016-05-28T20:53:08" period_3_start="2016-05-28T21
      :47:22" period_4_start="2016-05-28T22:06:21"
      period_5_start="2016-05-28T22:29:19" season_id="2015"
      season_name="Season 2015/2016">
2    <Event id="2002501076" event_id="1" type_id="34" period_id
       ="16" min="0" sec="0" team_id="186" outcome="1" x="0.0"
       y="0.0" timestamp="2016-05-28T18:31:20.871"
       last_modified="2016-05-28T19:41:33" version
       ="1464460892677">
3      <Q id="1844398382" qualifier_id="194" value="17861" />
4      <Q id="1971617116" qualifier_id="44" value="1, 2, 2, 3,
         2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5" />
5      <Q id="1687357007" qualifier_id="197" value="239" />
6      <Q id="1983652265" qualifier_id="131" value="1, 2, 3, 4,
         5, 6, 7, 8, 9, 10, 11, 0, 0, 0, 0, 0, 0, 0" />
7      <Q id="1071094016" qualifier_id="130" value="4" />
8      <Q id="1300398263" qualifier_id="59" value="1, 15, 12,
         14, 3, 4, 19, 8, 9, 11, 7, 6, 10, 13, 18, 20, 22, 23"
         />
9      <Q id="283877491" qualifier_id="227" value="0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0" />
10     <Q id="1508462225" qualifier_id="30" value="28411, 88483,
          39563, 61256, 18522, 17861, 37055, 44989, 19927,
         36903, 14937, 88477, 60025, 39790, 107593, 93127,
         80209, 100180" />
11   </Event>
12   ...
```

Listing 2.5: Example of foul and subsequent card event in an F24 feed

```
<Event id="1227128966" event_id="105" type_id="4" period_id
    ="1" min="9" sec="44" player_id="88483" team_id="186"
    outcome="0" x="56.2" y="10.7" timestamp="2016-05-28T19
    :59:48.996" last_modified="2016-05-28T20:00:39" version
    ="1464462038194">
  <Q id="155783154" qualifier_id="152" />
  <Q id="1837853712" qualifier_id="13" />
  <Q id="1762606589" qualifier_id="285" />
  <Q id="819241028" qualifier_id="56" value="Right" />
  <Q id="1569714366" qualifier_id="233" value="80" />
  <Q id="199342494" qualifier_id="265" />
</Event>
<Event id="1721898852" event_id="106" type_id="17" period_id
    ="1" min="10" sec="9" player_id="88483" team_id="186"
    outcome="1" x="0.0" y="0.0" timestamp="2016-05-28T20
    :00:13.890" last_modified="2016-05-28T23:45:51" version
    ="1464475551081">
  <Q id="185245862" qualifier_id="13" value="243" />
  <Q id="967262012" qualifier_id="31" />
</Event>
```

2.2.2 The F9 Match Results Feed

The F9 feed contains match results, or summaries. It is mostly extended match metadata and precomputed aggregations of the F24 event feed, but also contains some useful information not easily available elsewhere, like the number of minutes each player played.

The aggregation provided is mostly counts of each event type per player, e.g. showing that the home team's goalkeeper touched the ball 38 times and had 17 accurate passes out of a total of 35.

2.2.3 The F40 Squad Feed

The F40 feed has one entry per season per competition, and contains the team squads for the season. It also lists any transfers, in or out, and team and player metadata. An example is show in appendix A.1.

2.3 Overview of Database Systems

Choosing the right database model for the task is the first and most fundamental thing to consider, but also not straightforward. In later chapters, we will use both a graph database and a relational database. This section gives an overview of the merits of each of the two classes of databases.

2.3.1 Relational Database

The relational database is the most popular choice of database worldwide [14], and there is little reason to think that football analysis is any different. In fact, Opta themselves use a relational database for their internal database [15][16]. Therefore, it is an obvious choice for our benchmark.

Relational databases store their data in tables of columns and rows, and most often use Structured Query Language (SQL) for querying.

2.3.2 Graph Database

Graph databases represent data in terms of *nodes*, *edges* and *properties* instead of tables of columns and rows. Because it allows for traversing the data directly instead of searching for foreign keys, it has a theoretical advantage on queries with many *join*-operations.

If we, for a named player, want to find the names of every player he has successfully dribbled, the relational approach and graph approach would be quite different. The relational approach would be to search the *players* table by name, then use the player's ID to search the *events* table for dribbling events involving the player, then use these IDs to search the *qualifiers* table for any qualifiers linked to these events and containing a player id. Finally, we can scan the *players* table to find the names of the players. This amounts to

$$\frac{2e}{p}log(p) + log(e) + log(q)$$

comparisons, where e is the number of events, p is the number of players and q is the number of qualifiers.

A graph database, on the other hand, has the property that following links or edges happens in constant time, so it would start by locating the player node with the matching name, then follow the links to each event, then from each dribbling event follow the links to any qualifiers, and finally from the qualifiers with a link to a player, follow the link to this player's node. This should amount to roughly

$$log(p) + \frac{e}{p}$$

comparisons.

While both approaches have the same general complexity of $\mathcal{O}(log(n))$, the constant factors should give the graph database a clear theoretical advantage, which will increase with queries involving more tables/joins and node types. On the other hand, as we will see in later chapters, this may change when indexes and other query optimization techniques are applied.

2.3.3 Query Complexity

When analyzing query performance, it is useful to be able to say something about its complexity. This section outlines some of the most common operations in query executions in databases, along with their complexity.

Scans

The most basic operation, is the **sequential scan**. It involves traversing the entire table, and consequently has a complexity of $\mathcal{O}(N)$.

If the table has an index for the columns that we query on, an **index scan** is often used. Since the index is ordered, this scan can use a binary search, which is only $\mathcal{O}(log(N))$.

Joins

A **hash join** is a type of join that creates a hash map from the smallest table, then scans the larger table doing look-ups in the hash map. This has a complexity of $\mathcal{O}(M + N)$, since both creating the hash map of the smaller table and scanning the larger are linear operations [17].

Another approach to joining, is the **merge join**. It is efficient if the tables both have indexes on the joining columns, because it can then just scan the two tables in parallel, leading to a complexity of $\mathcal{O}(M + N)$. However, if a table doesn't have an index, it will have to sort the table first, leading to $\mathcal{O}(M + Nlog(N))$ if one table has an index or $\mathcal{O}(Mlog(M) + Nlog(N))$ if none of them have [17].

The final join type we will describe, is the **nested loop join**. It is theoretically a very expensive join, because it scans the entire second table for every row in the first, giving an $\mathcal{O}(MN)$ complexity. However, if one of the tables have an index, it can use an index scan instead, and have the complexity of $\mathcal{O}(Mlog(N))$. This can be quite fast if the unindexed table is small.

2.4 Previous Work

In this section, we will present previous work related to mapping XML data to relational databases, then look at work related to comparison of different databases for specific applications.

2.4.1 Mapping XML to Relational Databases

Florescu and Kossmann [18] compare four simple approaches to mapping XML to relational databases. Their approaches are based on representing the

XML document as a graph, where each element is a node, and each element-sub-element relationship is an edge. Attributes on elements are treated as sub-elements.

The **edge approach** stores all of the document's edges in a single table. Each row represents an edge, and has an ordinal number to record the order of the elements, a flag indicating the target type, which could be a reference to another edge to indicate a sub-element, or a value type such as string or integer. The rows also have references to its source and target.

The values, or leafs, are stored in separate tables, with one for each value type.

The **universal table** approach stores each distinct path from the root node to a leaf as a row. This means that the universal table has two columns for each element type in the document: an ordinality column and a target column. The target column points to a value in the value table or to a sub-element in the universal table.

They note that this approach is denormalized. It stores a lot of redundant information, and each row is likely to store many *nulls* for columns it doesn't use.

The **binary approach** groups all edges with the same value in the same table. Each of these tables contain four columns: an ordinal number to record the order of the elements, a target type in the same way as for the *edge approach*, and references to the edges source and target, The values are stored in separate tables in the same way as in the previous approaches.

The **binary inline approach** is the same as the *binary approach*, but instead of referencing values in separate tables, the values are stored directly in the edge tables. This means having a column for each possible data type, and removing the flag column indicating the target type. Leaves will be distinguished by having *null* as their target reference value.

Their experiments found the *binary inline* approach to outperform the others in almost every way. It required the least disk space, and it was the most performant for nearly all of their benchmark queries.

2.4.2 Comparing Databases

Angles et al. [19] compare five different databases for application to social networks. They create a micro-benchmark with several domain-specific queries modelled after common interactions with known social networks. Their data-schema consists of *persons* with a few attributes, and *friend* connections between each other. Each person may also *like* any number of *web pages*, which also has some attributes. They measure the data loading and query execution time. Several different types of databases are benchmarked, including both relational and graph databases.

They found that all of their tested databases were able to complete the queries within reasonable time, except for reachability queries. In the reachability queries, like "Get the friends of the friends of a given person P",

they found a gap between the graph databases and the relational databases emerged when the number of hops was larger than 4.

Jouili and Vensteenberghe [20] present GDB, a distributed graph database benchmarking framework, and compare the performance of Neo4j, Titan, OrientDB and DEX. They test both load and traversal workloads, and additionally benchmark the databases' performance with a large number of concurrent requests.

They found Neo4j to outperform all of the other databases in traversal workloads. In a read-only intensive workload where they queried for vertices by property, they found Neo4j and OrientDB slightly ahead of the others. In intensive write workloads, they found Neo4j and Titan significantly inferior to the others.

2.5 Summary

In this chapter, we have gone through an overview of the background for the thesis. We presented the current state of football analytics, then described the Opta event data in some detail. Further, we looked at the classes of database that will be relevant for the later chapters, and we discussed how one can benchmark databases. Finally we presented some relevant work done in the field previously.

Chapter 3

Experiment Design

In this chapter we will present our benchmarking system. We will describe the overall system structure, then give a short introduction to each of our database systems. Further, we will describe how our system sets up the databases, then give an overview of the data we will be using for our benchmarks. Finally, we will describe the queries that make up our benchmark and what it will measure.

3.1 The Benchmarking System

To easily benchmark the different schemas and databases, we have designed a modular system where new databases and schemas can easily be added [21].

The system consists of three main parts: The databases, the schemas and their populating scripts, and the profiling scripts that run the benchmarks themselves.

3.2 The Database Systems

For these experiments, we have chosen PostgreSQL as our relational database system and Neo4j as our graph database system. This section gives a brief introduction to the two databases.

3.2.1 PostgreSQL

PostgreSQL [22] is a relational database, initially developed at the University of California, Berkeley, and was first released in 1996 [23]. Today, it is one of the most popular choices among open source relational databases [14]. It is written in C, uses SQL as its query language, and has a modern query planner that uses dynamic programming and a number of heuristics to decide on an efficient execution plan [24].

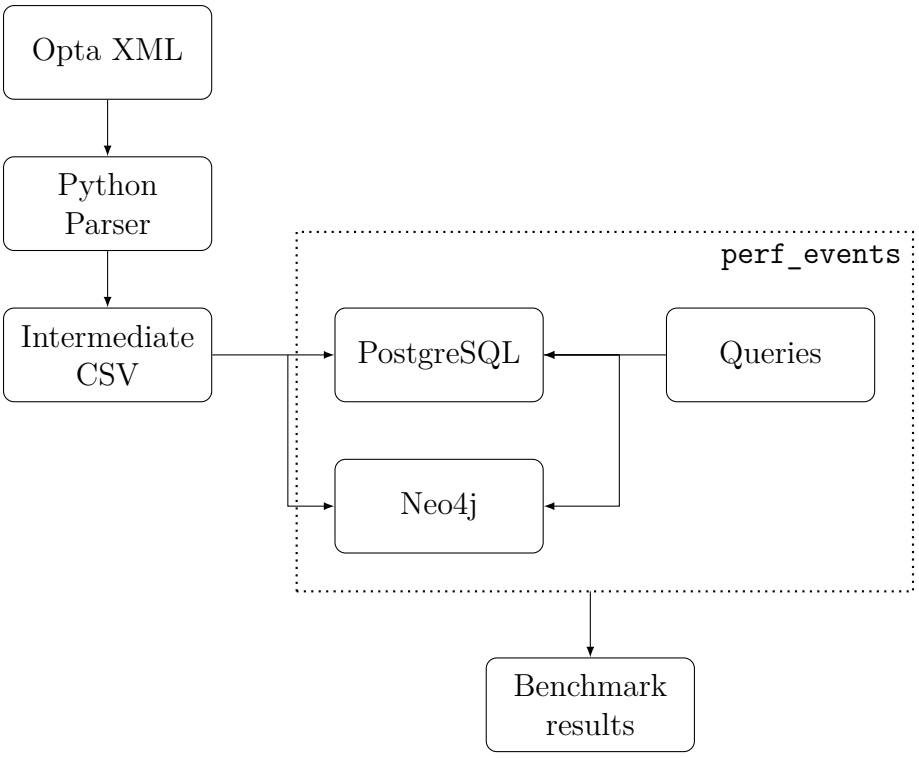


Figure 3.1: Overview of our benchmarking system.

As a relational database, it represents its data as rows in tables. The tables are stored in "heap files", and each heap file consists of several sub-units which are called "pages". The pages have a fixed size of 8KB, and are the smallest units that the database reads and writes. It is these pages that contain the rows. A page, if we disregard its header, has the rows' data—which are called tuples—at the end, and the indices and offsets of the tuples at the beginning, the two growing towards each other. The tuples may be shuffled around when data is deleted and added, but the pointers at the beginning of the page stays in the same place. These files have a maximum size of 1GB, meaning that the table will be split between files if it gets larger than that including its metadata.

This structure means that variable length fields has little impact on performance, as long as the fields are not too long. It does not matter too much how the rows themselves are aligned, since the database only reads and writes 8KB pages. However, rows have a maximum size of 2KB, so should a row be bigger than that, there will be some differences. First, the database will try to compress the fields of the row. If the row is still too big, it will use a system called *The Oversized-Attribute Storage Technique* (TOAST) which offloads the largest fields to a separate TOAST table until the row is below the 2KB limit [25].

3.2.2 Neo4j

Neo4j [26] is a native graph database developed by Neo Technology, now Neo4j Inc., and first released in 2007. It represents data as nodes and relationships, which can both have properties.

It is written in Java, and has its own query language: *Cypher*. It stores its data mainly in a number of intertwined linked lists. Each node or relationship has a pointer to the first of its properties, with the rest stored as a linked list extending from the first. Each node also has a pointer to its first relationship, with the rest stored as a doubly linked list extending from the first.

Each relationship is a part in *two* doubly linked lists, one for the start node's relationships, and one for the end node's. Additionally it has pointers to its start and end node [27][28].

3.3 Setting up the Databases

As the Opta data is delivered to us as XML files, some parsing is needed to be able to populate our databases. Since we want to be able to compare multiple schemas in each of the database systems, we will also need a way of keeping these schemas separate.

This section will describe how our system achieves this for each of our database systems.

3.3.1 PostgreSQL

To interact with PostgreSQL from our parsing scripts, we use the *psycopg2* package [29], which allows us to easily execute SQL commands through a Python method call. To execute queries, we use the default `psql` shell.

We start by creating the tables, keeping the different schemas separate using PostgreSQL's "schemas" feature. This means we can keep different schemas in the same database, allowing us to access them all from a single connection, which is convenient when we want to compare different schemas.

For parsing the XML, we use the `xml.etree` module from Python's standard library. This allows us to easily iterate over the elements in each XML file. Initial experiments with executing an SQL `INSERT` statement for each element parsed, proved to be prohibitively slow for our 16 million row data set. Therefore, we use PostgreSQL's `COPY FROM` bulk load feature, which is able to load entire tables from CSV files in a single go.

From the XML files, we therefore generate temporary CSV files adhering to the format specified by PostgreSQL's documentation [30]. The CSV files are generated with the `csv` module from Python's standard library.

Since our workload is an offline analytical one, and we will only have a single connection to the database, we can safely increase the work memory available to each query from the default 8 MB to a more generous 1024 MB,

which should ensure that most sorting operations can be executed without writing to disk.

3.3.2 Neo4j

Neo4j doesn't have a "schemas" feature like PostgreSQL, which means we have to create different databases for each of our schemas. This means changing from one schema to another isn't as simple either. In fact, we have to edit the main Neo4j configuration file and restart the database server.

We automated this through our database populating scripts, which other than this follows much of the same approach as for PostgreSQL. The XML is parsed with Python's `xml.etree` module. For interfacing with Neo4j, as with PostgreSQL, initial experiments using the "py2neo" package to insert each XML element separately as we parsed the XML files proved prohibitively slow, as each insertion would be done over HTTP. Therefore, we use a feature equivalent to PostgreSQL's `COPY FROM`, namely the `neo4j-import` tool, which also bulk loads from CSV files.

For executing queries from our benchmarking scripts, we use the `cypher-shell` [31].

3.4 The Benchmarking Data

Opta's data sets are rich, and includes information about not only every event in a match through the F24 feed, but also, through the F40 and F9 feeds, every team transfer, players on loan to other clubs, information about team staff and match officials, and about the teams' stadia, etc. While the breadth of the information from the F40 and F9 feeds open for a wealth of interesting queries, the small size of this data means it is not immediately interesting for performance benchmarking. Therefore we will limit our schemas to events, qualifiers, players, matches, teams, squads, seasons and competitions.

3.5 Queries

To benchmark the different approaches, we will construct a collection of queries with different levels of complexity. The first level should aggregate simple events of a single or few event types. The second level considers compound events, or *situations*, of constant length, like an event of type a followed by a successful event of type b. The third and last level should query for situations made up of an unbounded number of events.

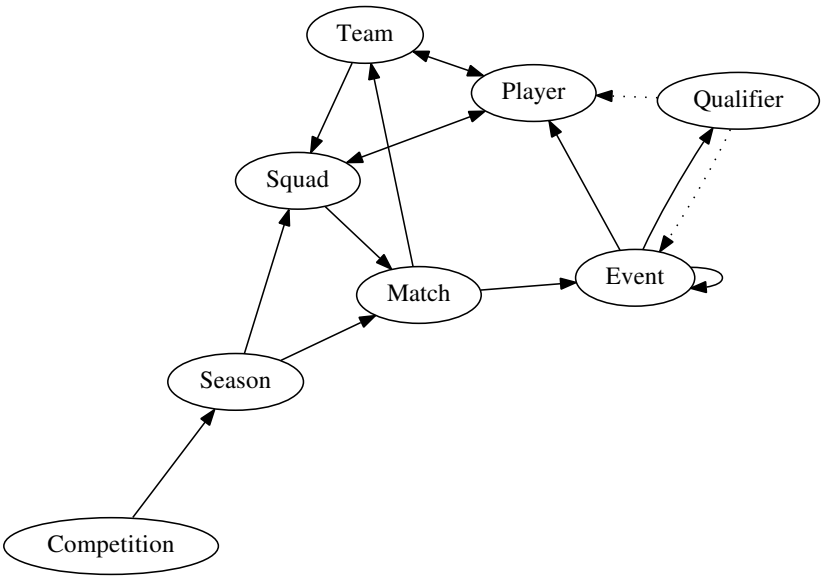


Figure 3.2: The Benchmark Data Structure

3.5.1 Level 1: Top 10 long ball passers in Norwegian Tippeligaen 2016

For out first-level query, we want to create a top 10 list of the players with the most completed long passes in Norwegian Tippeligaen 2016. This is a simple matter of finding all pass events that have a qualifier with a `qualifier_id` of 1, indicating a *long pass*, then averaging the boolean `outcome` attribute of the events for each player. The pass events are identified by a the attribute `type_id` having a value of 1.

To make the results more interesting, we exclude players that have fewer than 100 attempts at long passes.

3.5.2 Level 2: Corners cleared by defender heading

For the second-level query, we want to benchmark the databases ability to identify fixed-length compound queries. One such situation is where a corner is cleared by a defender heading, and that is what we will identify in this query.

This means that we want to identify a specific sequence of events: a sequence containing a taken corner followed by a successful, headed clearance by a player on the defending team.

Such a situation must begin with a pass event (`type_id` = 1) with an associated "Corner Taken" qualifier (`qualifier_id` = 6). It must also end with a "Clearance" event (`type_id` = 12) with an appropriate "Body Part: Head" qualifier (`qualifier_id` = 15).

Between the "Corner Taken" and "Clearance" events, there may also be some other events, which we will limit to the possibility of two "Aerial Duel"

events (`type_id` = 44); one for each player involved.

While we want to test the identification of these situations, we will only actually return the number of them from our queries. This makes it easier to check that the result is correct, and that the queries are not limited by having to visualize large amounts of data. However, we will take care in constructing our queries in such a way as to only require minor edits to extract the underlying events for each situation counted.

3.5.3 Level 3: Counter-Attacks

For this last level, we chose to expand on the previous level where we wanted to identify compound situations of a fixed number of events, and now make our goal to identify compound situations of an *unknown* number of events, namely counter-attacks.

A counter-attack is not as easily defined as the header-cleared corner situation of the previous section, but there are a couple of ground rules we can use:

- It has to start with a change of possession.
- The following offense should have a fast pace.

Our event data makes it easy to detect the first one, as each event has a `team_id` attribute. The second one can be approximated by using the x and y pitch coordinate attributes together with the match clock, which is found in the `min` and `sec` attributes of each event. By calculating the pitch distance between the dispossession event and the counter-attack-ending event and dividing it by the time between the events, we get an approximation of the attacking velocity.

The counter-attack-ending event, however, is not as easily defined. The counter-attack ends if the defending team takes possession of the ball, e.g. through an interception, a tackle, a block, or a keeper save; if the attacking team scores; if the referee awards the defending team a free-kick; if the ball goes out for a goal kick or if the defending team somehow regains balance and the attacking pace is reduced.

Since we have no information about the location of the defending players, we can't say anything about the defense's balance. It is, however, likely that the attacking pace will be slowed down if the defense regains balance, so by introducing a lower bound on the pace, we should exclude most of these situations. We choose our lower bound as 6% of the field per second, and define the pace as the percentage of the field traveled towards the goal line divided by the difference in time. 6% should correspond to about $6m/s$, which should be a good enough approximation to the running speed of an athlete.

Most of the other counter-attacking events implies either a change of possession or a significant slowdown of the attack. The former is easily tested

in the same way as the beginning of the counter-attack, while the latter is taken care of by our lower bound on pace.

Finally, we want to exclude situations that are so short that it would not count as an attack at all, meaning situations where a team may take possession and move quickly forward a short distance before being dispossessed again. We achieve this by introducing a lower bound on distance traveled towards the goal line, which we set to 20% of the pitch length. Towards the same goal, and also to avoid division by zero, we also exclude situation shorter than 3 seconds in time.

This is still only a rough approximation to finding counter-attacks, but includes the most computationally demanding parts, which is that the number of events is unbounded.

In this query, like the previous, and for the same reasons, we will only output the total count from the query, but will also here take care to write the queries in such a way as to be able to extract the underlying events for each situation with only minor edits to the query.

3.6 Measurements

Databases are often the bottlenecks of applications, so many database systems include means to benchmark and analyze queries [32][33]. Both schema and query construction can have considerable impact on application performance. However, when comparing different database systems, these tools cannot be trusted to provide comparable measurements, so we will have to use an external tool for measuring.

For our experiments the operating system will be Linux, which provides a profiler tool called "perf"[34], which we will describe below.

3.6.1 Linux' `perf_events`

The `perf_events` subsystem is a part of the Linux kernel that counts different kinds of "events". It uses special purpose CPU registers to count hardware events, like CPU cycles and branch mispredictions, but can also count software events like system calls and page misses. Compared to user space profilers, the `perf_events` subsystem introduces little overhead, and it allows us to count events that are otherwise not available.

The user space tool for interacting with `perf_events` is called `perf`. More specifically, a program may be profiled by running `perf stat <program_name>`, upon which `perf` will fork off the program and start measuring it as a new process.

`perf stat` also has an option for attaching to an already running process, which is necessary for our use case. Most database systems use a client-server model, and so when we pass a query to the database shell, it is not the same process that executes the query.

We can get around this by using the `-p` option, which allows us to specify the process ID of an already running process that we want to attach to, in our case the database server. It also allows us to use the client process as a timer, stopping the measurements when the client process exits.

3.6.2 Metrics

Our first and foremost goal is to be able to execute complex queries in as little time as possible. We are interested in simulating an analyst workload, where the database has a single user, so we will not measure large amounts of concurrent queries.

We will measure the number of instructions executed, the number of cycles elapsed, and the "task clock", which is a measure of the total time the measured task, in our case the query, has spent on the CPU. Additionally, to be able to better understand the reasons behind the measured performance, we will measure page faults, number of branches and branch misses, context switches, CPU migrations, cache references and misses, memory loads and memory stores. All of these events are measured through the `perf stat` Linux command.

`perf` will monitor any number of events that we want, but it is limited by the number of counting registers are available in the CPU. Therefore, `perf` uses event multiplexing [35]. It divides the register time between all the hardware events we want to measure, and then scales the numbers to provide an estimate. It also lists how big a part of the register time each event got, and we will report this as well.

We will execute each query 5 times, then report the mean values and their relative standard deviation.

3.7 Test Environment

The benchmarks were run under 64-bit Debian Linux buster/sid, on a 2 core, 4 thread Intel(R) Core(TM) i7-6500U CPU running at 2.50GHz. The machine was equipped with 16GB RAM, and the data was stored on an SSD.

We are using PostgreSQL version 10beta2 compiled with GCC 6.4.0, and Neo4j version 3.3.0-aplha04 running on OpenJDK 1.8.0.141.

To measure, we are using perf version 4.11.11 on Linux 4.11.0-2-amd64.

3.8 Summary

In this chapter, we have seen the structure of our benchmarking system. We have studied the different database systems we will use, and we have seen the data we will benchmark them on and how we will structure it. We have specified the queries we will use for our benchmark, and we have described the

system we will use for measuring it. Finally, we have listed the specifications of the system our benchmark will be run on.

Chapter 4

Relational Approach

In this chapter, we will describe the design and execution of two relational approaches to storing and querying our data. We will expand on the general data structure design from the previous chapter, and describe two PostgreSQL-specific schemas for representing our data. Further, we will design queries for each of our three levels described in section 3.5, then analyze and compare the results from each of the two schemas.

4.1 Schema Design

To construct schemas for our relational approach, we will start from Florescu and Kossmann's binary inline approach which we outlined in section 2.4.1. But since we know beforehand which elements will be leafs, and we know the type of every attribute, we can make our tables even more compact than they outlined.

4.1.1 The Base Schema

Expanding on the inline binary approach from section 2.4.1, we construct this schema by creating a table for each type of XML element with appropriate references between them. That is, an `events` table, a `qualifiers` table, a `players` table, and so on, with foreign keys set up so we can query relations.

The resulting Events and Qualifiers tables can be seen in listing 4.1, and the remaining tables are listed in appendix B.1.

4.1.2 Using the JSONB column type

By far, the largest tables of the base schema will be the *events* and *qualifiers* tables, and it is important to notice that the *qualifiers* table will never be used on its own—it has to be joined with the *events* table to provide meaningful information. But joining large tables is computationally expensive.

There are hundreds of different qualifier types, but each event will only have something in the area of 5 to 10 of them. The qualifiers are essentially

Listing 4.1: The F24 tables for the base schema.

```
CREATE TABLE events (
       id                INTEGER PRIMARY KEY,
       match_id          INTEGER REFERENCES matches(id),
       team_id           INTEGER REFERENCES teams(id),
       player_id         INTEGER REFERENCES players(id),
       event_id          INTEGER,
       type_id           INTEGER,
       period_id         INTEGER,
       min               INTEGER,
       sec               INTEGER,
       outcome           BOOLEAN,
       keypass           BOOLEAN,
       assist            BOOLEAN,
       x                 FLOAT,
       y                 FLOAT,
       timestamp         TIMESTAMP,
       last_modified     TIMESTAMP,
       version           BIGINT
);

CREATE TABLE qualifiers (
       id                SERIAL PRIMARY KEY,
       event_id          INTEGER REFERENCES events(id),
       qualifier_id      INTEGER,
       value             TEXT
);
```

Listing 4.2: The single F24 event table for the JSONB schema.

```
CREATE TABLE events (
       id                INTEGER PRIMARY KEY,
       match_id          INTEGER REFERENCES matches(id),
       team_id           INTEGER REFERENCES teams(id),
       player_id         INTEGER REFERENCES players(id),
       qualifiers        JSONB,
       event_id          INTEGER,
       type_id           INTEGER,
       period_id         INTEGER,
       min               INTEGER,
       sec               INTEGER,
       outcome           BOOLEAN,
       keypass           BOOLEAN,
       assist            BOOLEAN,
       x                 FLOAT,
       y                 FLOAT,
       timestamp         TIMESTAMP,
       last_modified     TIMESTAMP,
       version           BIGINT
);
```

a sparse set of attributes on the events, but finding them will nevertheless mean joining two large tables.

We can look back to Florescu and Kossmann again, and see that the *edge approach* is a different take on this problem. This structure would be equivalent to the result of a full outer left join between the Events and the Qualifiers. This would, as they note, create large amounts of duplicate information. We would have several rows for each event, only differing by the `qualifier_id` and `value` columns from the Qualifiers table, and their conclusion showed that it was not very efficient.

From this situation, it would be tempting to aggregate the two columns from the Qualifiers table into an array of key-value pairs. This leads us to our second schema.

PostgreSQL offers a few types that allow storing unstructured data in table columns, and among them is JSONB. JSONB stores data as a binary representation of a JSON object, providing a dictionary-like structure.[1] This means we can replace the entire *qualifiers* table with a new JSONB column in the *events* table, then use the `qualifier_id` as key, and the qualifier's optional `value` attribute as the value. This seems a middle way between the base model and the edge approach, and as we can see in table 4.1 it saves us about 50 percent in table sizes.

[1]While this looks and acts like a hash map, it is actually not. The keys are stored in a sorted array [36] and queried with binary search [37].

Table 4.1: Table sizes for different the base and JSONB schemas, including indexes.

Table \ Schema	Base	JSONB
Events	365 MB	529 MB
Qualifiers	728 MB	0
Total	**1093 MB**	**529 MB**

4.2 Optimization

4.2.1 Indexing

Indexes are integral to database performance, as it allows us to avoid complete table scans for filtering the indexed tabled on the indexed columns. For our base schema, the largest table by far is the Qualifiers table, and we know that it will always be filtered or sorted by the `event_id` for joining with the Events table. Therefore we create an index on the Events table over the `event_id` column.

In both the base and the JSONB schemas, the Events table will be queried extensively. It is the biggest table by two orders of magnitude in the JSONB schema, and the second biggest after the Qualifiers table in the base schema. We will most often be filtering the events table by its `type_id` column, and therefore we create an index on that in both of our schemas. In addition, we expect that `match_id` and `player_id` are columns that will be used to filter and order the Events table a lot, so those indexes are also created. The complete set of indexes can be seen in listing 4.3.

All of the other tables are very small compared to the Events and Qualifiers tables, so we wont be considering any indexes on those.

Listing 4.3: Adding indexes to our PostgreSQL database

```
1 CREATE INDEX ON qualifiers (event_id);
2 CREATE INDEX ON events (type_id);
3 CREATE INDEX ON events (match_id);
4 CREATE INDEX ON events (player_id);
```

4.3 Implementation of Queries in PostgreSQL

In this section we will describe the reasoning behind the design of our PostgreSQL queries. For each level we will list the resulting queries for both the Base schema and the JSONB schema, except for the last level where the queries are the same.

Listing 4.4: PostgreSQL query for finding top 10 long ball passers in Norwegian Tippeligaen for the 2016 season

```
SELECT
    name,
    avg(outcome::int) AS rate,
    count(*) AS total
FROM events
JOIN qualifiers ON events.id = qualifiers.event_id
JOIN players ON player_id = players.id
JOIN matches ON match_id = matches.id
            AND competition_id = 90
            AND season_id = 2016
WHERE type_id = 1 AND qualifier_id = 1
GROUP BY player_id, name
HAVING count(outcome) > 100
ORDER BY rate DESC
NULLS LAST
LIMIT 10;
```

4.3.1 Level 1

The Base Schema

This query is pretty straight-forward, but does require a few `JOIN`s, as can be seen in listing 4.4.

We join the `events` table to the `qualifiers` table on `event_id`, to `players` on `player_id` to get the players' names, and to `matches` on `match_id`, `season_id` and `competition_id` to limit our events to only Norwegian Tippeligaen 2016. Then we filter the results by `type_id` and `qualifier_id`, group them by `player_id`, filter the groups to only leave those with more than 100 successes, then we aggregate the groups with average over the outcome column to create the `rate` which we order the results by, limiting to the top 10.

The JSONB Schema

The JSONB query for this level only differs in that we have removed the join to the `qualifiers` table, and instead of filtering on the `qualifier_id` column, we use the JSONB "contains" operator "?" to check for the proper qualifier ID in the JSONB column.

4.3.2 Level 2

In this query, we want to find situations where a corner is cleared by a defender heading, as we described in section 3.5.2.

We initially explored using a window function to implement this query, which would allow us to look at the current row and the following three rows

Listing 4.5: PostgreSQL query for finding top 10 long ball passers in Norwegian Tippeligaen for the 2016 season, using the JSONB schema

```
1  SELECT
2      name,
3      avg(outcome::int) AS rate,
4      count(*) AS total
5  FROM events
6  JOIN players ON player_id = players.id
7  JOIN matches ON match_id = matches.id
8              AND competition_id = 90
9              AND season_id = 2016
10 WHERE type_id = 1 AND qualifiers ? '1'
11 GROUP BY player_id, name
12 HAVING count(outcome) > 100
13 ORDER BY rate DESC
14 NULLS LAST
15 LIMIT 10;
```

to see if it matched the pattern we were looking for. However, we found this approach to be cumbersome and inefficient compared to the approach using *recursive common table expressions*.

Common table expressions (CTE) are a way to construct temporary, named results, and are used through the `WITH` construct. They are useful to avoid recomputation of a subquery.

The Base Schema

We start our query with a CTE in which we first annotate each event row with the ID of its preceding row, then left join the result with the Qualifiers table filtered by the relevant `qualifier_id`s, which are 6 for signifying that a pass event is a corner and 15 for signifying that a clearance event was executed with the head. This operation only makes sense because we know that no event will have both of these qualifiers, as that would indicate taking a corner kick with the head. We filter the result by `type_id` and `qualifier_id`, such that the only events remaining are corner kicks, aerial duels and headed clearances.

Next, on line 12 in listing 4.6, we define another CTE. This CTE is recursive: It filters the previous CTE, `pertinent_events`, to only include the headed clearances. Then, it proceeds to `UNION` itself with the result of `JOIN`ing itself with the `pertinent_events` table such that it effectively walks from each clearance event, backwards in time adding new aerial duel or corner events until the set no longer changes. At that point we have gathered all the events of every one of the situations we are looking for, and finally we can simply count the number of corner events in our set to see how many of these situations there are.

Listing 4.6: PostgreSQL query for counting corners cleared by heading

```
WITH RECURSIVE pertinent_events AS (
     SELECT id, type_id, prev_id
     FROM (SELECT *, lag(id) OVER () AS prev_id
           FROM events) e
     LEFT JOIN (
         SELECT event_id, qualifier_id FROM qualifiers
         WHERE qualifier_id IN (6, 15)
     ) q ON q.event_id = e.id
     WHERE type_id = 44
        OR (type_id = 1 AND qualifier_id = 6)
        OR (type_id = 12 AND qualifier_id = 15 AND outcome)
), cleared_corners AS (
     SELECT * FROM pertinent_events
     WHERE type_id = 12
   UNION
     SELECT e.* FROM cleared_corners c,
                     (SELECT * FROM pertinent_events
                      WHERE type_id IN (1, 44)) e
     WHERE e.id = c.prev_id
)
SELECT count(*)
FROM cleared_corners
WHERE type_id = 1;
```

The JSONB Schema

The JSONB variant of this query, is very similar to the Base variant. The only difference is that we can skip joining the Qualifiers table, and that the checks on `qualifier_id` in the `WHERE` clause of the first CTE are replaced by key-existence checks on our JSONB column. The resulting query is shown in listing 4.7.

4.3.3 Level 3

In the third level query, we want to identify and count the counter-attacks in our data, as we explained in section 3.5.3.

Since the number of events involved is unbounded, our initial idea of using window functions from the previous query would not work here, because PostgreSQL's window functions can only have a fixed size. The recursive CTE approach, however, will work in this case too.

Our query consists of three main parts:

- First, we annotate each event row with the `id` and `team_id` of its successor;
- then, we use a recursive CTE to "walk" from each possession-switch to the next while adding the first event's `id` as an `attack_id` column to

Listing 4.7: PostgreSQL query for counting corners cleared by heading, using the JSONB schema

```
WITH RECURSIVE pertinent_events AS (
     SELECT id, type_id, prev_id
     FROM (SELECT *, lag(id) OVER () AS prev_id
           FROM events) e
     WHERE type_id = 44
        OR (type_id = 1  AND qualifiers ? '6')
        OR (type_id = 12 AND qualifiers ? '15' AND outcome)
), cleared_corners AS (
     SELECT * FROM pertinent_events
     WHERE type_id = 12
   UNION
     SELECT e.* FROM cleared_corners c,
                     (SELECT * FROM pertinent_events
                      WHERE type_id IN (1, 44)) e
     WHERE e.id = c.prev_id
)
SELECT count(*)
FROM cleared_corners
WHERE type_id = 1;
```

all rows to be able to group events by event chains of possession;

- and finally, we join the annotated events to our recursive `possessions` CTE while filtering based on the pace, distance and time criteria outlined in section 3.5.3.

For the first part of our query, we initially used a CTE to avoid recomputing the subquery all of the three times it is referenced. However, through PostgreSQL's `EXPLAIN` tool which shows the query plan, we saw that the planner chose to do a "merge join" in all three instances. Since a merge join requires the tables to be ordered by the joining column, and because PostgreSQL does not optimize across CTE borders [38], this entailed three full sorts of the entire `events` table. Therefore, we dropped the CTE, and used three identical subqueries instead. This led the planner to choose a "hash join" instead, which does not need sorting. To avoid cluttering the query by repeating the subquery, we used psql's `\gset` feature, which works as a text-substituting macro.

Since we don't use the qualifiers at all in this query, the queries for the base and JSONB schemas are identical.

4.4 Results

In this section we will analyze and compare the results of benchmarking the queries from the previous section. The queries are run through our

Listing 4.8: PostgreSQL query for counting counter-attacks

```
SELECT $$(SELECT id, team_id, min, sec, x,
                 lead(team_id) OVER () AS next_team,
                 lead(id) OVER () AS next_id
            FROM events)$$ events_1 \gset
WITH RECURSIVE possessions AS (
    SELECT id attack_id, *
      FROM :events_1 e
     WHERE team_id != next_team
  UNION
    SELECT p.attack_id, e.*
      FROM :events_1 e, possessions p
     WHERE e.id = p.next_id
       AND (e.team_id = p.team_id OR p.id = p.attack_id)
)
SELECT count(DISTINCT p.attack_id)
  FROM events e, possessions p
 WHERE e.id = p.attack_id
   AND (p.sec - e.sec + (p.min - e.min)*60 >= 3
        AND p.x - (100-e.x) > 20
        AND (p.x - (100-e.x)) /
            (p.sec - e.sec + (p.min - e.min)*60) > 6);
```

benchmarking system, outlined in section 3.1, and we use PostgreSQL's query plan analyzer `EXPLAIN` [32] and its `ANALYZE` profiling parameter to analyze the query execution. It is worth noting that when split times are presented, they are from `EXPLAIN ANALYZE`, and might not add up to the time measured by `perf` through our benchmarking system.

4.4.1 Level 1

The Base Schema

From table 4.2, we see that the query is measured to execute in 1,430 $\pm$ 4 ms of CPU time, which is aggregated across all cores used. Dividing by the "CPUs utilized" value of 2.908, we get that the actual elapsed wall time is 492 $\pm$ 1 ms.

Using PostgreSQL's `EXPLAIN ANALYZE`, we see that the execution plan has some interesting features. The query spends just over 99% of its time joining the four tables, most of which is from filtering the `events` table by `type_id` and joining it with the `qualifiers` table. It does a sequential scan on the `events` table filtering it from 1.8 million rows down to 1 million rows in about 100 ms. Then, it joins it with the `matches` table to filter by `competition_id` and `season_id`, which returns about 200,000 rows in about 50 ms. These 200,000 rows are then joined with the `qualifiers` table using a "nested loop join", which means that it does 200,000 index scans on the `qualifiers` table, and it does it in around 350 ms. This takes a long time

because it also filters on the `qualifier_id`. It returns around 46,000 rows which are joined with the `players` table to annotate the rows with player names before the rows are sorted with Quicksort to prepare them for being grouped by `player_id` and `name`. After the grouping, there are about 1000 rows left, and all that is left is to filter them by number of successes and sort them by their average.

Using the complexities we described in section 2.3.3 and the number of rows passed to each query plan component relative to the number of rows in the events table, we can estimate that the query should have a complexity of $0.15E\ log(E) + 1.65E$, where E is the size of the `events` table, normalized to 1 for our current table size. This assumes that all other tables have constant ratios of size to the `events` table.

The JSONB Schema

From table 4.3, we see that the JSONB variant is measured to execute in 689 ± 0.7 ms of CPU time, which is aggregated across all cores used. Dividing by the "CPUs utilized" value of 2.838, we get that the actual elapsed wall time is 243 ± 0.2 ms.

This variant also starts by filtering the `events` table, but it filters both by `type_id` *and* by checking if the `qualifiers` JSONB column contains the proper key. This takes us from 1.8 million to 182,000 rows in around 190 ms. The rows are then joined with the `matches` and `players` tables in around 40 ms, before passing the same resulting 46,000 rows as last time to Quicksort. The `players` table is joined with a nested loop join.

For comparison, we can calculate an estimate of this query's complexity using the same base as for the last one, and will then get a complexity of $0.06E\ log(E) + 0.35E$.

Summary

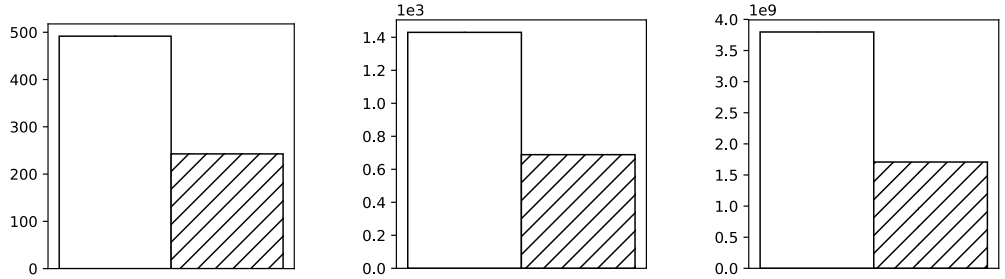
Both queries have a nested loop join with one side indexed which takes them to $\mathcal{O}(Nlog(N))$, but joining with the big `qualifiers` table moves the constant factors much in favor of the JSONB schema. It executes in under half the time for our data. The complexity of our queries are very similar, and in fact the limit of the ratio between the two estimated complexities is 2.5, indicating that the relative difference will stay about the same for any size of data set.

Table 4.2: Query 1 benchmark results after 5 runs for the base schema.

Metric	Mean Count	RSD	Measured	Note
cycles	3,797,879,935	0.11%	24.97%	2.656 GHz
instructions	5,224,539,005	0.57%	31.76%	1.38 insn per cycle
task-clock	1,430	0.26%	100.00%	2.908 CPUs utilized
context-switches	77	29.29%	100.00%	0.054 K/sec
cpu-migrations	21	2.61%	100.00%	0.015 K/sec
branches	1,056,526,679	0.66%	32.44%	738.863 M/sec
branch-misses	6,500,381	0.51%	33.11%	0.62% of all branches
minor-faults	28,970	0.20%	100.00%	0.020 M/sec
major-faults	0	0.00%	100.00%	0.000 K/sec
page-faults	28,970	0.20%	100.00%	0.020 M/sec
cache-misses	39,177,553	0.82%	33.71%	41.165% of all cache refs
bus-cycles	33,854,299	0.17%	33.91%	23.675 M/sec
mem-loads	133	5.97%	26.87%	0.093 K/sec
mem-stores	881,304,573	0.78%	26.37%	616.325 M/sec
cache-references	95,171,672	0.86%	25.67%	66.557 M/sec
dTLB-load-misses	1,955,976	0.86%	24.82%	
iTLB-load-misses	47,626	2.40%	24.12%	
LLC-load-misses	4,914,193	0.88%	24.63%	38.97% of all LL-cache hits
L1-dcache-loads	1,527,320,025	1.35%	24.20%	1068.104 M/sec
L1-dcache-load-misses	71,002,920	0.69%	24.54%	4.65% of all L1-dcache hits
LLC-loads	12,609,523	0.81%	24.72%	8.818 M/sec

Table 4.3: Query 1 benchmark results after 5 runs for the JSONB schema.

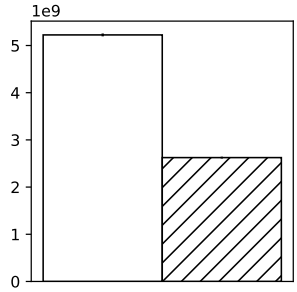
Metric	Mean Count	RSD	Measured	Note
cycles	1,708,073,577	0.10%	23.81%	2.480 GHz
instructions	2,624,675,534	0.73%	31.17%	1.54 insn per cycle
task-clock	689	0.10%	100.00%	2.838 CPUs utilized
context-switches	51	13.15%	100.00%	0.074 K/sec
cpu-migrations	12	10.17%	100.00%	0.018 K/sec
branches	546,872,727	0.24%	32.76%	794.081 M/sec
branch-misses	2,699,244	0.25%	34.22%	0.49% of all branches
minor-faults	23,828	0.18%	100.00%	0.035 M/sec
major-faults	0	0.00%	100.00%	0.000 K/sec
page-faults	23,828	0.18%	100.00%	0.035 M/sec
cache-misses	12,520,688	0.33%	35.65%	40.436% of all cache refs
bus-cycles	15,665,312	0.13%	36.68%	22.747 M/sec
mem-loads	88	6.62%	28.93%	0.127 K/sec
mem-stores	453,853,074	0.27%	28.59%	659.013 M/sec
cache-references	30,964,087	1.07%	28.23%	44.961 M/sec
dTLB-load-misses	269,395	0.54%	27.77%	
iTLB-load-misses	19,550	8.94%	26.44%	
LLC-load-misses	783,933	2.12%	23.07%	33.00% of all LL-cache hits
L1-dcache-loads	744,466,416	0.60%	23.01%	1080.995 M/sec
L1-dcache-load-misses	23,855,588	0.51%	21.54%	3.20% of all L1-dcache hits
LLC-loads	2,375,645	0.85%	22.35%	3.450 M/sec



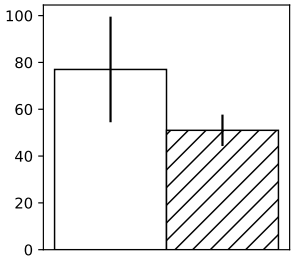
(a) `elapsed`

(b) `task-clock`

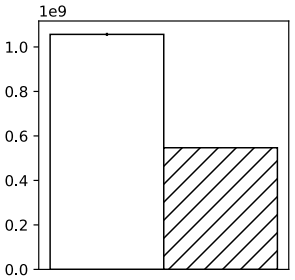
(c) `cycles`



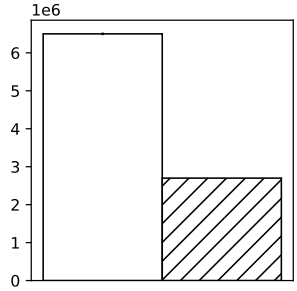
(d) `instructions`



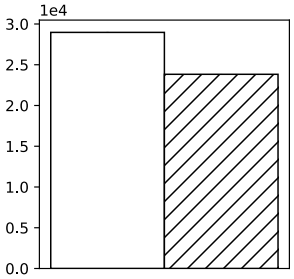
(e) `context-switches`



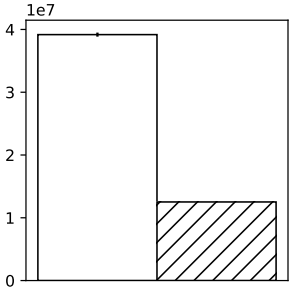
(f) `branches`



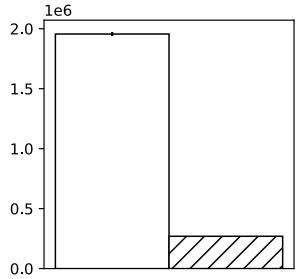
(g) `branch-misses`



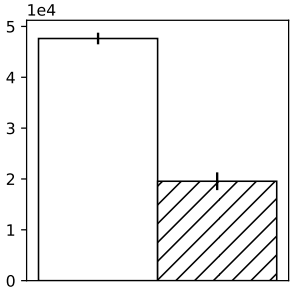
(h) `page-faults`



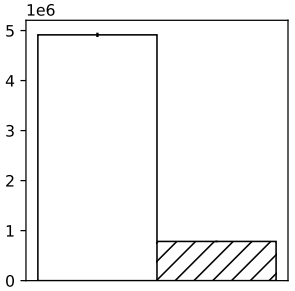
(i) `cache-misses`



(j) `dTLB-load-misses`



(k) `iTLB-load-misses`



(l) `LLC-load-misses`

Figure 4.1: Comparison of mean event counts for Query 1 after 5 runs. The white bars represent the base schema, while the striped represent the JSONB schema. The error bars show the standard deviation.

4.4.2 Level 2

The Base Schema

From table 4.4, we see that the query is measured to execute in 2,253 ± 5 ms of CPU time, which is aggregated across all cores used. Dividing by the "CPUs utilized" value of 1.502, we get that the actual elapsed wall time is 1,500 ± 3 ms.

Again using `EXPLAIN ANALYZE`, we see that the query starts, as we would expect, with running a window function over the `events` table to annotate the rows with the `id` of the preceding row in about 750 ms. Then, the `type_id` and `outcome` filters are applied, reducing the row count from 1.8 million to 1.1 in about 200 ms. The `qualifiers` table is scanned and filtered, in parallel, from the initial 8 million rows down to 14,000 in about 350 ms. The filtered qualifier and event rows are now hash joined while filtering on the *combination* of `type_id`s, `qualifier_id`s and `outcome`s, taking the row count from 1.1 million event rows to 90,000. At this point the `pertinent_events` CTE is complete.

The 90,000 rows are then filtered by `type_id=12`, leaving 22,000 rows after about 500 ms. These rows are then hash joined to the `pertinent_events` table five times in the recursive union in the `cleared_corners` CTE, expanding the 22,000 rows to 35,000 in about 50 ms. Finally these rows are filtered by `type_id` to leave only the corner kicks, which are then counted.

Applying the same types of complexity estimation as we did for level 1, looking at the types of the query components and their complexity, we find the complexity to be linear. The recursive CTE could introduce higher order complexity if the number of joins was unbounded, but since the situation we are looking at is limited to four events in a row, we avoid that.

The JSONB Schema

From table 4.5, we see that the JSONB variant is measured to execute in 944 ± 5 ms of CPU time, which is aggregated across all cores used. Dividing by the "CPUs utilized" value of 0.995, we get that the actual elapsed wall time is 949 ± 5 ms.

The big differences in the JSONB schema for this level, as for the previous, is the missing hash join to the `qualifiers` table. It starts with the window function, as before, but instead of the multiple-stage filtering and joining from the base query, it applies all of the filters in one go—completing the `pertinent_events` CTE in 1,100 ms.

As with the base variant, there are only linear time components in this query, and so the only difference is in lower constant factors.

Summary

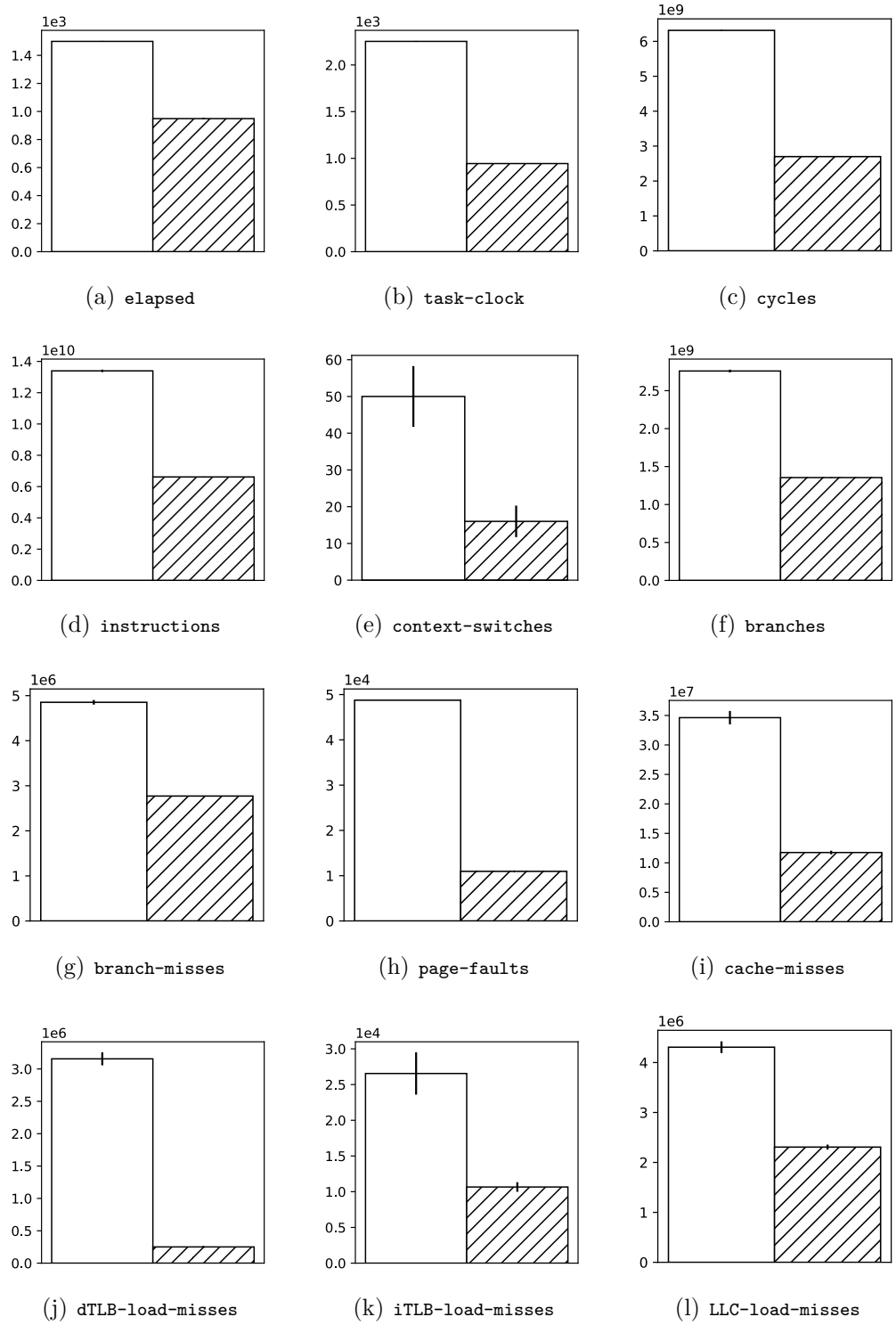
The relative difference in execution time is not as great for this level as it was for the previous, but it is clear that the JSONB is the more efficient of the approaches. The difference is smaller because the base schema processed the `qualifiers` table in parallel for this query, and we can see the difference in "CPUs utilized" in 4.4. The extra threads are likely also the cause for the high number of context switches relative to the JSONB schema that we can see in 4.2(e).

Table 4.4: Query 2 benchmark results after 5 runs using the base schema.

Metric	Mean Count	RSD	Measured	Note
cycles	6,314,381,880	0.21%	24.55%	2.803 GHz
instructions	13,395,105,709	0.59%	30.94%	2.12 insn per cycle
task-clock	2,253	0.23%	100.00%	1.502 CPUs utilized
context-switches	50	16.55%	100.00%	0.022 K/sec
cpu-migrations	20	4.45%	100.00%	0.009 K/sec
branches	2,758,902,405	0.66%	31.13%	1224.811 M/sec
branch-misses	4,852,026	1.05%	31.36%	0.18% of all branches
minor-faults	48,765	0.07%	100.00%	0.022 M/sec
major-faults	0	0.00%	100.00%	0.000 K/sec
page-faults	48,765	0.07%	100.00%	0.022 M/sec
cache-misses	34,625,128	3.27%	32.17%	53.789% of all cache refs
bus-cycles	52,040,787	0.23%	32.85%	23.103 M/sec
mem-loads	0	0.00%	26.04%	0.000 K/sec
mem-stores	2,311,701,643	0.57%	25.84%	1026.277 M/sec
cache-references	64,371,913	1.05%	25.67%	28.578 M/sec
dTLB-load-misses	3,156,378	3.18%	25.48%	
iTLB-load-misses	26,550	11.15%	25.32%	
LLC-load-misses	4,306,909	2.71%	24.54%	49.07% of all LL-cache hits
L1-dcache-loads	4,439,199,266	1.18%	25.05%	1970.776 M/sec
L1-dcache-load-misses	34,690,280	1.29%	24.95%	0.78% of all L1-dcache hits
LLC-loads	8,777,308	2.64%	24.81%	3.897 M/sec

Table 4.5: Query 2 benchmark results after 5 runs using the JSONB schema.

Metric	Mean Count	RSD	Measured	Note
cycles	2,699,938,316	0.14%	24.28%	2.860 GHz
instructions	6,615,328,081	0.32%	30.69%	2.45 insn per cycle
task-clock	944	0.55%	100.00%	0.995 CPUs utilized
context-switches	16	26.81%	100.00%	0.017 K/sec
cpu-migrations	1	31.62%	100.00%	0.001 K/sec
branches	1,355,126,470	0.31%	31.00%	1435.381 M/sec
branch-misses	2,771,830	0.20%	31.34%	0.20% of all branches
minor-faults	10,955	0.00%	100.00%	0.012 M/sec
major-faults	0	0.00%	100.00%	0.000 K/sec
page-faults	10,955	0.00%	100.00%	0.012 M/sec
cache-misses	11,751,656	2.61%	31.68%	58.049% of all cache refs
bus-cycles	22,596,208	0.57%	31.83%	23.934 M/sec
mem-loads	0	0.00%	25.42%	0.000 K/sec
mem-stores	1,157,677,940	0.27%	25.42%	1226.239 M/sec
cache-references	20,244,315	4.15%	25.42%	21.443 M/sec
dTLB-load-misses	250,742	5.74%	25.40%	
iTLB-load-misses	10,660	6.29%	25.31%	
LLC-load-misses	2,306,835	2.15%	24.18%	68.34% of all LL-cache hits
L1-dcache-loads	2,194,472,803	0.43%	24.84%	2324.436 M/sec
L1-dcache-load-misses	16,022,848	1.34%	24.55%	0.73% of all L1-dcache hits
LLC-loads	3,375,728	3.36%	24.30%	3.576 M/sec



(a) `elapsed` (b) `task-clock` (c) `cycles`

(d) `instructions` (e) `context-switches` (f) `branches`

(g) `branch-misses` (h) `page-faults` (i) `cache-misses`

(j) `dTLB-load-misses` (k) `iTLB-load-misses` (l) `LLC-load-misses`

Figure 4.2: Comparison of mean event counts for Query 2 after 5 runs. The white bars represent the base schema, while the striped represent the JSONB schema. The error bars show the standard deviation.

4.4.3 Level 3

The Base Schema

From table 4.6, we see that the query is measured to execute in 7,077 ± 12 ms of CPU time, which is aggregated across all cores used. Dividing by the "CPUs utilized" value of 0.999, we get that the actual elapsed wall time is 7,784 ± 13 ms.

PostgreSQL starts the query with running a window function over the `events` table to annotate each row with the `id` and `team_id` of following row, taking about 900 ms to do so. The result is then filtered by `team_id` and `next_team` to find all the changes of possession, which reduces the number of rows from 1.8 million to 680,000 in about 100 ms. These rows are the starting point for the recursive `possessions` CTE, and they are now hash joined 63 times with a subquery that runs the same window function over the events table as was explained above. The whole construction recursive `possessions` CTE takes about 5 seconds to complete, ending with 2,5 million rows.

The `possessions` CTE is then hash joined to the `events` table, while filtering on the `min`, `sec` and `x` columns, completing in about 4 seconds. The resulting rows are then counted to provide the final result.

Regarding complexity, the level 3 query has much of the same characteristics as the level 2 query, but unlike the corner situation, the counter-attack is technically unbounded. There is theoretically no limit to how many consecutive events a team can have, although in practice it is unusual for a team to keep possession of the ball an entire period, and even if a team managed to do this, there is a practical upper limit for how many events the team can manage during 45 minutes.

If we assume that there is no relation between the size of the events table and the maximum number of consecutive events from the same team, which we think is reasonable, we can say that the query is linear in the size of the events table.

The JSONB Schema

The query is identical for the two schemas, but it nonetheless executed slightly slower in the JSONB schema. From table 4.7, we see that it was measured to execute in 7,259 ± 6 ms of CPU time, which with the "CPUs utilized" value of 0.999, gives us the wall time of 7,266 ± 6 ms.

Summary

The schemas had very similar performance, which was expected as the queries were identical. However, the base schema had a slight edge. The difference is most likely attributable to the fact that the `events` table has an extra column, and therefore is bigger, in the JSONB schema. We can see this

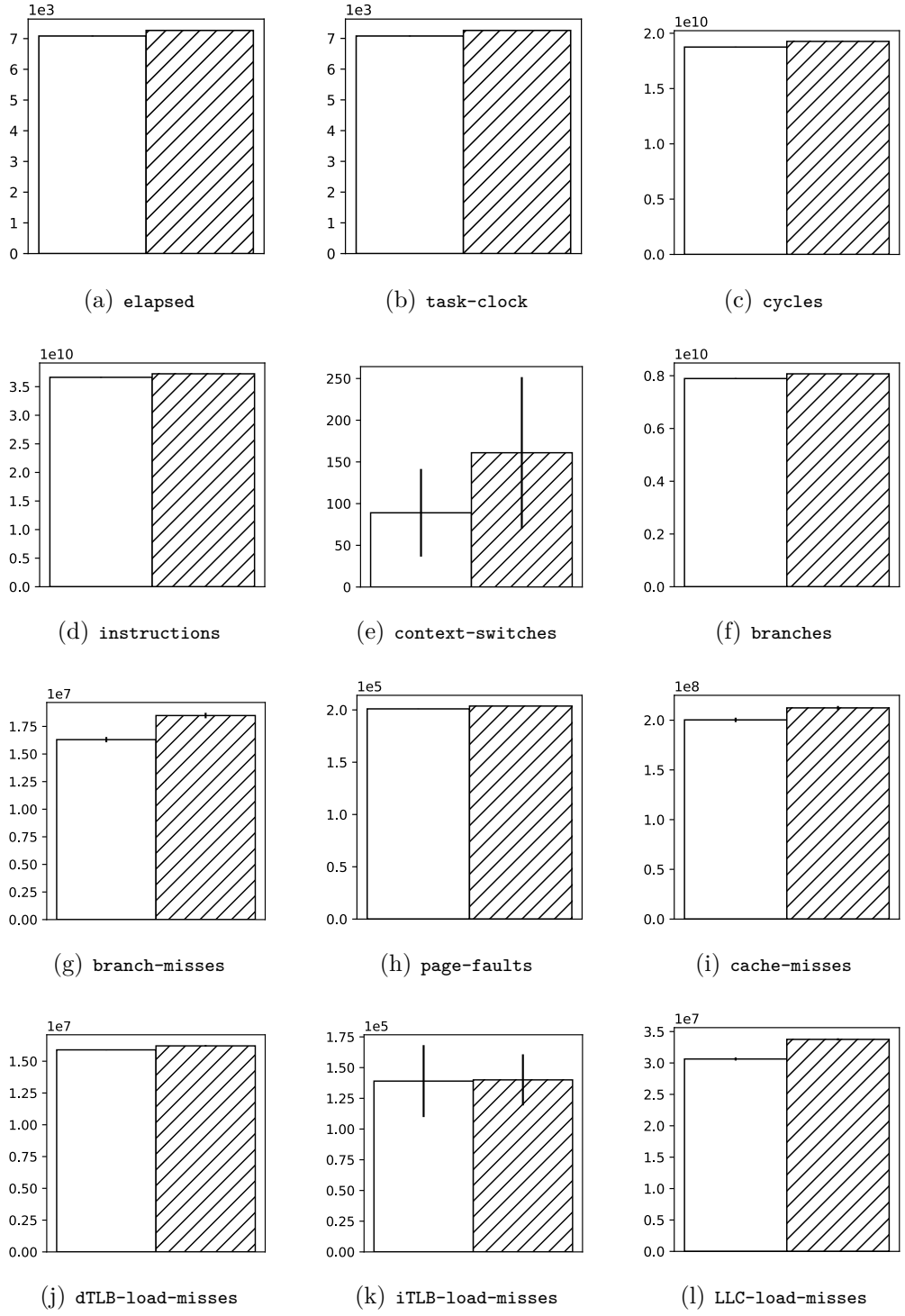
manifested in a higher number of memory store events when comparing tables 4.6 and 4.7.

Table 4.6: Query 3 benchmark results after 5 runs using the base schema.

Metric	Mean Count	RSD	Measured	Note
cycles	18,753,339,797	0.15%	24.97%	2.650 GHz
instructions	36,639,518,454	0.16%	31.27%	1.95 insn per cycle
task-clock	7,077	0.17%	100.00%	0.999 CPUs utilized
context-switches	89	59.04%	100.00%	0.013 K/sec
cpu-migrations	2	11.11%	100.00%	0.000 K/sec
branches	7,892,510,232	0.21%	31.31%	1115.188 M/sec
branch-misses	16,301,005	1.46%	31.34%	0.21% of all branches
minor-faults	200,998	0.05%	100.00%	0.028 M/sec
major-faults	0	0.00%	100.00%	0.000 K/sec
page-faults	200,998	0.05%	100.00%	0.028 M/sec
cache-misses	200,306,195	1.10%	31.36%	66.703% of all cache refs
bus-cycles	169,428,487	0.17%	31.34%	23.940 M/sec
mem-loads	0	0.00%	25.01%	0.000 K/sec
mem-stores	5,605,380,302	0.19%	25.00%	792.023 M/sec
cache-references	300,296,482	0.82%	24.99%	42.431 M/sec
dTLB-load-misses	15,893,029	0.23%	24.97%	
iTLB-load-misses	139,021	21.06%	24.96%	
LLC-load-misses	30,633,737	0.79%	24.94%	66.48% of all LL-cache hits
L1-dcache-loads	10,925,477,537	0.12%	24.96%	1543.737 M/sec
L1-dcache-load-misses	223,332,158	0.36%	24.96%	2.04% of all L1-dcache hits
LLC-loads	46,082,865	0.35%	24.95%	6.511 M/sec

Table 4.7: Query 3 benchmark results after 5 runs using the JSONB schema.

Metric	Mean Count	RSD	Measured	Note
cycles	19,261,924,945	0.04%	24.94%	2.654 GHz
instructions	37,252,645,438	0.07%	31.22%	1.93 insn per cycle
task-clock	7,259	0.08%	100.00%	0.999 CPUs utilized
context-switches	161	56.33%	100.00%	0.022 K/sec
cpu-migrations	2	11.11%	100.00%	0.000 K/sec
branches	8,071,694,786	0.08%	31.28%	1111.989 M/sec
branch-misses	18,484,952	1.34%	31.32%	0.23% of all branches
minor-faults	203,688	0.04%	100.00%	0.028 M/sec
major-faults	0	0.00%	100.00%	0.000 K/sec
page-faults	203,688	0.04%	100.00%	0.028 M/sec
cache-misses	212,371,887	0.91%	31.36%	66.948% of all cache refs
bus-cycles	173,752,253	0.08%	31.38%	23.937 M/sec
mem-loads	0	0.00%	25.07%	0.000 K/sec
mem-stores	5,696,586,274	0.12%	25.04%	784.784 M/sec
cache-references	317,218,602	0.33%	25.02%	43.701 M/sec
dTLB-load-misses	16,208,159	0.36%	24.99%	
iTLB-load-misses	140,009	14.86%	24.97%	
LLC-load-misses	33,758,002	0.51%	24.91%	67.23% of all LL-cache hits
L1-dcache-loads	11,075,792,460	0.08%	24.93%	1525.845 M/sec
L1-dcache-load-misses	261,950,912	0.42%	24.92%	2.37% of all L1-dcache hits
LLC-loads	50,216,091	0.39%	24.92%	6.918 M/sec



(a) `elapsed` (b) `task-clock` (c) `cycles`

(d) `instructions` (e) `context-switches` (f) `branches`

(g) `branch-misses` (h) `page-faults` (i) `cache-misses`

(j) `dTLB-load-misses` (k) `iTLB-load-misses` (l) `LLC-load-misses`

Figure 4.3: Comparison of mean event counts for Query 3 after 5 runs. The white bars represent the base schema, while the striped represent the JSONB schema. The error bars show the standard deviation.

4.5 Comparison

The JSONB schema is significantly faster than the base schema for the two first levels, while the base schema takes a slight edge in the third level. The conclusion we draw from these results, is that the JSONB schema is a good solution for storing sparse attributes, and that it is the better choice of the two for almost any workload resembling our benchmarks. We find it hard to imagine an analyst workload that would never make use of the qualifiers, and it is only when the qualifiers are not involved that the base schema takes a slight edge. In fact, a real application of the counter-attacks query should probably add filters on which events are allowed to start and end a counter-attack to exclude some edge cases, and that would almost certainly make use of the qualifiers.

4.6 Summary

In this chapter we have seen the construction and benchmarking of our relational approaches to storing and querying the Opta data, and we have proven that PostgreSQL is able to extract higher-level events from the Opta event data.

We described the design of the base and JSONB PostgreSQL schemas, then designed queries for the three levels specified in our benchmarking system design, and finally analyzed and compared the results, marking the JSONB schema as the most performant of the two.

Chapter 5

Graph Approach

In this chapter, we will design and and benchmark a graph approach to storing and querying our data. We will construct a schema to hold the data, then design and implement queries for each of the three levels outlined in section 3.5. Finally, we will benchmark the queries through our benchmarking system, and analyze the results.

5.1 Schema Design

In section 2.4.1 we saw that Florescu and Kossmann used a graph as a conceptual intermediate stage of mapping XML to a relational database. They described the graph as having all elements as nodes, each element-sub-element relationship as an edge, and treating each attribute on an element as a sub-element.

This is a natural starting point for our graph schema. But, as we saw in section 3.2.2, Neo4j also allows attributes on the nodes and relationships themselves. This means we have to find a balance of what should be stored as attributes, and what should be stored as relationships to other nodes. The obvious approach, and also what we will do, is to map XML attributes directly to Neo4j attributes, and only make edges for the element-sub-element relationships.

This leaves some gaps, however, as many elements have references to other elements that are not their sub-elements. An example of this is the `Event` elements, which have references to both the player involved and the player's team in its attributes, in addition to having several `Qualifier` sub-elements. These cases are obviously already relationships, and we will create edges for these as well.

For our data, this means we will have nodes for all events, qualifiers, players, games, teams and seasons, which are separated by "labels" in Neo4j. We will then link the events to their qualifiers, players and games, and the games to their seasons, etc., using edges, which are called relationships in Neo4j.

Additionally, one would want to create edges between successive elements where the order matter, as Neo4j doesn't have any internal ordering to rely on, unlike tables in most relational databases. In our case, this means creating directed edges with a "preceding" label from each event to the following.

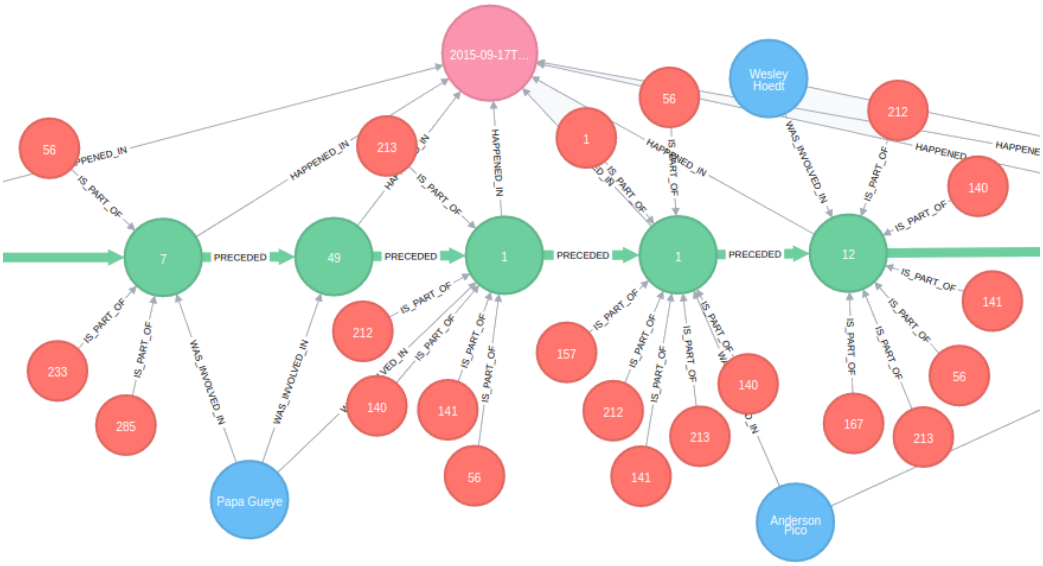


Figure 5.1: Excerpt from a match as modeled in Neo4j. Green nodes are events, red nodes are qualifiers, blue nodes are players, and the pink node is the game node.

5.2 Optimization

5.2.1 Warm-up

Where PostgreSQL relies heavily on the operating system's file cache, Neo4j mainly uses its own cache [39]. This, in combination with the fact that our benchmarking system restarts the Neo4j server between every query benchmarked, means that the database has a significant cold start problem.

While this is a real concern, and should be taken into account when comparing the databases, it is not a real problem for our target workload. The Opta data analyst will likely only use data sets that fit in memory, so the warm-up is really only a startup cost of populating the cache.

We will counter the cold start problem with a warm-up before each query benchmark, where we execute the query to be benchmarked one time without measuring after restarting the server.

5.2.2 Indexes

Indexes in Neo4j behaves much as indexes in other databases, in that they sacrifice disk space for read performance on nodes with a certain label when queried by one or more specified attributes, in the same way as indexes on tables and columns work in most relational databases. The indexes are mainly used in the beginning of queries, to collect the nodes that we will follow relationships from.

Many of the columns we created indexes over for PostgreSQL in section 4.3 are replaced by relationships in our Neo4j schema. The only remaining is the index on the events over the `type_id` attribute. We will create this, and also add an index on the qualifiers and its `qualifier_id` attribute:

```
1 CREATE INDEX ON :Event(type_id);
2 CREATE INDEX ON :Qualifier(qualifier_id);
```

5.3 Implementation of Queries in Cypher

5.3.1 The Cypher Query Language

The Cypher Query Language, or just *Cypher* for short, is a declarative query language created by Neo Technologies for use in their Neo4j database. Since 2015 it has had an open specification [40]. It now has several implementations in addition to Neo4j, most of them being external modules to provide graph querying to existing database projects [41].

The language bears some resemblance to SQL in that it is declarative and has familiar constructs like `ORDER BY`, `LIMIT` and `WHERE`, but is differentiated by the `MATCH` clause, which enables matching patterns of nodes and relationships.

5.3.2 Level 1

For the first level query, we take advantage of Cypher's graph pattern matching, and query for every sub-graph starting with a player node connected to an event node with a `type_id` of 1, which signifies a pass event, where that event node is connected to a game node, which further is connected to a season node with the correct `season_id`. Since we can only specify two-dimensional chains, we have to add a second part to our `MATCH`, specifying that the event node should be connected to a qualifier node with a `qualifier_id` of 1, signifying a "long ball".

Then we aggregate the average outcome and the event node count for each of our sub-graphs, filter the results on event count, and finally return the limited, ordered list.

Listing 5.1: Neo4j query for finding top 10 long ball passers in Norwegian Tippeligaen for the 2016 season

```
MATCH (p:Player)
      -[:WAS_INVOLVED_IN]-> (e:Event {type_id: 1})
      -[:HAPPENED_IN]-> (g:Game)
      -[:IS_A_GAME_IN]-> (s:Season {season_id: '201690'}),
      (q:Qualifier {qualifier_id: 1}) -[:IS_PART_OF]-> (e)
WITH p, avg(e.outcome) AS a, count(e) AS c
WHERE c > 100
RETURN p.name, a, c
ORDER BY a DESC
LIMIT 10;
```

Listing 5.2: Neo4j query for counting corners cleared by heading.

```
MATCH p = (e2:Event {type_id: 12, outcome: 1})
      <-[:PRECEDED*0..3]- (e1:Event {type_id: 1})
WHERE EXISTS((:Qualifier {qualifier_id: 15})
             -[:IS_PART_OF]-> (e2))
      AND EXISTS((e1) <-[:IS_PART_OF]-
                 (:Qualifier {qualifier_id: 6}))
      AND (EXISTS((e1) -[:PRECEDED]-> (e2))
           OR NONE (ex IN nodes(p)[1..-1]
                    WHERE ex.type_id <> 44))
RETURN count(*);
```

5.3.3 Level 2

In the second level query, we start by querying for every sub-graph starting with an event of `type_id` 12 with an `outcome` of 1, which corresponds to a successful clearance event, and specify that it has to be preceded by an event node with `type_id` 1, meaning a pass event, after *at most* three hops. Then we add further conditions in the `WHERE` clause, specifying that the clearance event node must be connected to a qualifier node with a `qualifier_id` of 15, meaning that the clearance must be conducted with the head. We further specify that the pass event must be connected to a qualifier node with a `qualifier_id` of 6, indicating a corner kick, and that the corner event must either immediately precede the headed clearance event, or that all of the intermediate event nodes have a `type_id` of 44, indicating an aerial duel. Finally, we count the number of matching sub-graphs.

5.3.4 Level 3

For our third level, we start the query with identifying all switches of possession, querying for all sub-graphs starting with an event node which is directly connected to another event node with a different `team_id` attribute.

Then, we query for all sub-graphs starting with one of the ending nodes of

Listing 5.3: Neo4j query for finding number of counter attacks in the dataset.

```
MATCH (e1:Event) -[:PRECEDED]-> (e2:Event)
WHERE e1.team_id <> e2.team_id
WITH e1, e2
MATCH p = (e2:Event) -[:PRECEDED*0..]-> (e3:Event)
WHERE ((e3.sec + e3.min*60) - (e1.sec + e1.min*60)) >= 3
      AND (e3.x - (100-e1.x)) > 20
      AND (e3.x - (100-e1.x)) /
      ((e3.sec + e3.min*60) - (e1.sec + e1.min*60)) > 6
      AND NONE (ex IN nodes(p) WHERE ex.team_id = e1.team_id)
RETURN count(DISTINCT e1);
```

our possession switch sub-graphs from before, further connected to what will be our counter-attack ending node through any number of hops, but filtered by the pace and distance conditions from the query definition in section 3.5.3, and ensuring that all of the event nodes in the same sub-graph have the same `team_id`.

Finally, we count all the distinct start nodes from our possession switch sub-graphs to get the number of counter-attacks.

5.4 Results

Neo4j has a `PROFILE` prefix that can be added to queries for profiling them, much like PostgreSQL's `EXPLAIN ANALYZE` that we used in the previous chapter. `PROFILE`, however, does not give quite as detailed output as its PostgreSQL counterpart—e.g. it does not show any split times or planner estimates—but it does show the number of "db hits" for each part of the execution.

5.4.1 Level 1

From table 5.1 we see that the first level query is measured to execute in 1.3 ± 0.2 s of CPU time, aggregated across all cores used. Dividing by the "CPUs utilized" value of 0.88, we get that the actual elapsed wall time is 1.6 ± 0.2 s.

The `PROFILE` tool shows us that the query starts with an index scan on the Qualifier nodes, filtering the 8 million nodes down to 184,000 nodes using as many "db hits". Next, it follows the `:IS_PART_OF` connection from the Qualifier nodes to find all the associated events, using 368,000 "db hits" because it must first get the relationship, then get the event node. Then it filters on the event nodes' `type_id` attributes and the qualifier nodes' `qualifier_id` attributes, which only reduces the sub-graph count by 1% to 182,000, but uses 366,000 "db hits" to look up the properties on both node types. Following the `:HAPPENED_IN` relationship, it adds a game node to each

sub-graph using “364,000” db hits for looking up relationships and the game nodes.

It then does a hash join between these sub-graphs and the sub-graphs with the game nodes that satisfy our league and season criteria, the latter of which we will not detail the query for because of its insignificant size. The sub-graph count is now 46,101. Next, it follows the `:WAS_INVOLVED_IN` relationship backwards from each event node to add player nodes to each sub-graph, using 92,000 “db hits”, and then it applies all of the attribute filters to every sub-graph again, which does not change anything, but still uses 46,000 “db hits”. Finally, it runs an aggregate over the sub-graphs and runs some insignificant last steps on the resulting 363 rows to produce the final list.

Examining the complexity of each step and the number of rows between each step, we estimate the complexity to

$$0.375E + 2log(E) - 3.3$$

which is $\mathcal{O}(E)$.

Table 5.1: Query 1 benchmark results after 5 runs using Neo4j.

Metric	Mean Count	RSD	Measured	Note
cycles	3,342,529,229	12.79%	26.28%	2.436 GHz
instructions	5,978,503,761	7.43%	33.95%	1.79 insn per cycle
task-clock	1,372	12.72%	100.00%	0.880 CPUs utilized
context-switches	799	11.01%	100.00%	0.582 K/sec
cpu-migrations	119	6.45%	100.00%	0.087 K/sec
branches	1,008,297,594	9.05%	35.36%	734.890 M/sec
branch-misses	8,770,654	33.26%	36.77%	0.87% of all branches
minor-faults	159	10.86%	100.00%	0.116 K/sec
major-faults	0	0.00%	100.00%	0.000 K/sec
page-faults	159	10.96%	100.00%	0.116 K/sec
cache-misses	43,134,723	6.47%	38.69%	27.392% of all cache refs
bus-cycles	31,637,407	12.55%	37.25%	23.059 M/sec
mem-loads	0	0.00%	27.98%	0.000 K/sec
mem-stores	797,053,407	6.65%	27.42%	580.926 M/sec
cache-references	157,473,914	18.56%	26.37%	114.774 M/sec
dTLB-load-misses	570,927	20.26%	25.96%	
iTLB-load-misses	113,714	45.81%	25.35%	
LLC-load-misses	1,655,288	9.52%	24.96%	16.93% of all LL-cache hits
L1-dcache-loads	1,646,307,749	10.63%	24.63%	1199.899 M/sec
L1-dcache-load-misses	107,771,767	13.41%	24.11%	6.55% of all L1-dcache hits
LLC-loads	9,775,708	28.88%	24.24%	7.125 M/sec

5.4.2 Level 2

From table 5.1 we see that the second level query was measured to execute in 1.4 ± 0.2 s of CPU time, aggregated across all cores used. Dividing by the "CPUs utilized" value of 0.944, we get that the actual elapsed wall time is 1.5 ± 0.3 s.

The execution starts with an index scan on the event nodes to find the clearance events, which returns 52,000 rows. These nodes are then filtered by their `outcome` and by connection to a qualifier node with the right `qualifier_id`, to leave 22,000 rows. These rows are then made into sub-graphs with their four preceding event nodes, producing 88,000 rows, which are then filtered by `type_id` and connections to qualifier nodes with the corner kick `qualifier_id` to produce 3,060 rows which are counted to produce the final output.

Adding up the complexities while using the row counts as estimates for the constant factors, gives us a query complexity of around

$$0.1E + log(E)$$

which is, as the previous query, also $\mathcal{O}(E)$.

Table 5.2: Query 2 benchmark results after 5 runs using Neo4j.

Metric	Mean Count	RSD	Measured	Note
cycles	3,605,076,988	17.21%	26.26%	2.551 GHz
instructions	6,434,104,079	10.28%	33.84%	1.78 insn per cycle
task-clock	1,413	16.80%	100.00%	0.944 CPUs utilized
context-switches	800	12.25%	97.76%	0.566 K/sec
cpu-migrations	137	13.34%	97.76%	0.097 K/sec
branches	1,136,270,385	10.59%	33.54%	803.910 M/sec
branch-misses	10,851,417	29.07%	34.29%	0.96% of all branches
minor-faults	123	17.02%	96.99%	0.087 K/sec
major-faults	0	0.00%	96.99%	0.000 K/sec
page-faults	126	17.40%	96.99%	0.089 K/sec
cache-misses	45,323,775	12.23%	34.98%	26.402% of all cache refs
bus-cycles	34,479,652	15.14%	34.09%	24.394 M/sec
mem-loads	0	0.00%	25.59%	0.000 K/sec
mem-stores	848,104,182	6.63%	26.37%	600.033 M/sec
cache-references	171,665,656	17.46%	25.77%	121.453 M/sec
dTLB-load-misses	543,444	23.92%	24.99%	
iTLB-load-misses	163,888	30.62%	25.02%	
LLC-load-misses	1,181,148	28.73%	24.55%	10.08% of all LL-cache hits
L1-dcache-loads	1,858,802,808	11.29%	23.95%	1315.101 M/sec
L1-dcache-load-misses	92,067,506	19.65%	24.27%	4.95% of all L1-dcache hits
LLC-loads	11,718,482	32.96%	24.28%	8.291 M/sec

5.4.3 Level 3

From table 5.1 we see that the last level query was measured to execute in 40 ± 4.5 s of CPU time, aggregated across all cores used. Dividing by the "CPUs utilized" value of 1.537, we get that the actual elapsed wall time is 26 ± 3 s.

The query execution is started with a scan on the event nodes, which are then expanded to sub-graphs by following their `:PRECEDED` relationship to the following event node. These sub-graphs are then filtered by `team_id` to produce only sub-graphs which represent possession changes. The sub-graphs are then actually expanded by following `:PRECEDED` until the end of the current period, instead of to the next change of possession, as we would expect. This grows the number of sub-graphs from 700,000 to 1.8 million, and these rows are now hash joined to the event nodes again. It seems it must do this because until now the rows have only contained the relationships. The hash join leaves the same 1.8 million rows, which are then filtered by our pace and distance limitations specified in section 3.5.3 to leave only the rows, or sub-graphs, corresponding to counter-attacks. The rows are then counted to return the result.

This query, like its relational counterpart, has only linear components if we assume that the number of events to scan from each possession switch event is not proportional to the number of event nodes. However, since the query we constructed was unable to stop at the next possession switch event, it is only saved by the fact that it stops at the end of the current period. This gives us a very large constant factor, which is likely what slows this query down.

Just as we theoretically cant guarantee any upper limit on the number of events in a row from the same team, we can only be sure that the number of events from the current to the period end is at least as many, and because of this the theoretical class of the query is also here $\mathcal{O}(E^2)$.

Table 5.3: Query 3 benchmark results after 5 runs using Neo4j.

Metric	Mean Count	RSD	Measured	Note
cycles	115,272,000,335	11.57%	25.05%	2.863 GHz
instructions	182,141,250,649	14.64%	31.45%	1.58 insn per cycle
task-clock	40,267	11.28%	100.00%	1.537 CPUs utilized
context-switches	7,183	1.53%	100.00%	0.178 K/sec
cpu-migrations	702	3.86%	100.00%	0.017 K/sec
branches	31,468,157,275	14.34%	31.53%	781.484 M/sec
branch-misses	240,142,664	6.79%	31.68%	0.76% of all branches
minor-faults	1,719	47.91%	100.00%	0.043 K/sec
major-faults	0	0.00%	100.00%	0.000 K/sec
page-faults	1,723	47.84%	100.00%	0.043 K/sec
cache-misses	1,292,630,549	10.54%	31.75%	45.390% of all cache refs
bus-cycles	953,549,563	11.27%	31.71%	23.681 M/sec
mem-loads	0	0.00%	25.30%	0.000 K/sec
mem-stores	22,304,178,177	15.56%	25.24%	553.904 M/sec
cache-references	2,847,837,604	13.34%	25.21%	70.724 M/sec
dTLB-load-misses	51,974,302	4.72%	25.13%	
iTLB-load-misses	1,184,105	16.71%	25.05%	
LLC-load-misses	59,151,702	8.72%	24.94%	24.32% of all LL-cache hits
L1-dcache-loads	54,563,692,355	12.81%	25.02%	1355.041 M/sec
L1-dcache-load-misses	1,633,553,879	16.53%	25.00%	2.99% of all L1-dcache hits
LLC-loads	243,204,441	11.41%	24.95%	6.040 M/sec

5.5 Summary

In this chapter we have constructed and benchmarked the graph approach to storing and querying our data, and we have seen that Neo4j is indeed capable of extracting higher-level events from the Opta data, although we did not find it very efficient at the most complex of our benchmark queries.

We designed a schema for storing the data, and then constructed queries to extract the data we have specified in the three levels from section 3.5. Finally, we benchmarked the queries in our benchmarking system, and analyzed the results.

Chapter 6

Comparison

In the two previous chapters, we have looked at relational and graph based approaches to storing and querying our event data. We saw that both of the approaches made it possible to extract higher-level events from the Opta data. In this chapter we will compare the benchmark results of the two approaches to further identify which option gives the most performant querying.

We will limit the comparison to the most performant schemas from each approach, which means we will only compare the graph schema to the JSONB schema of the relational approach.

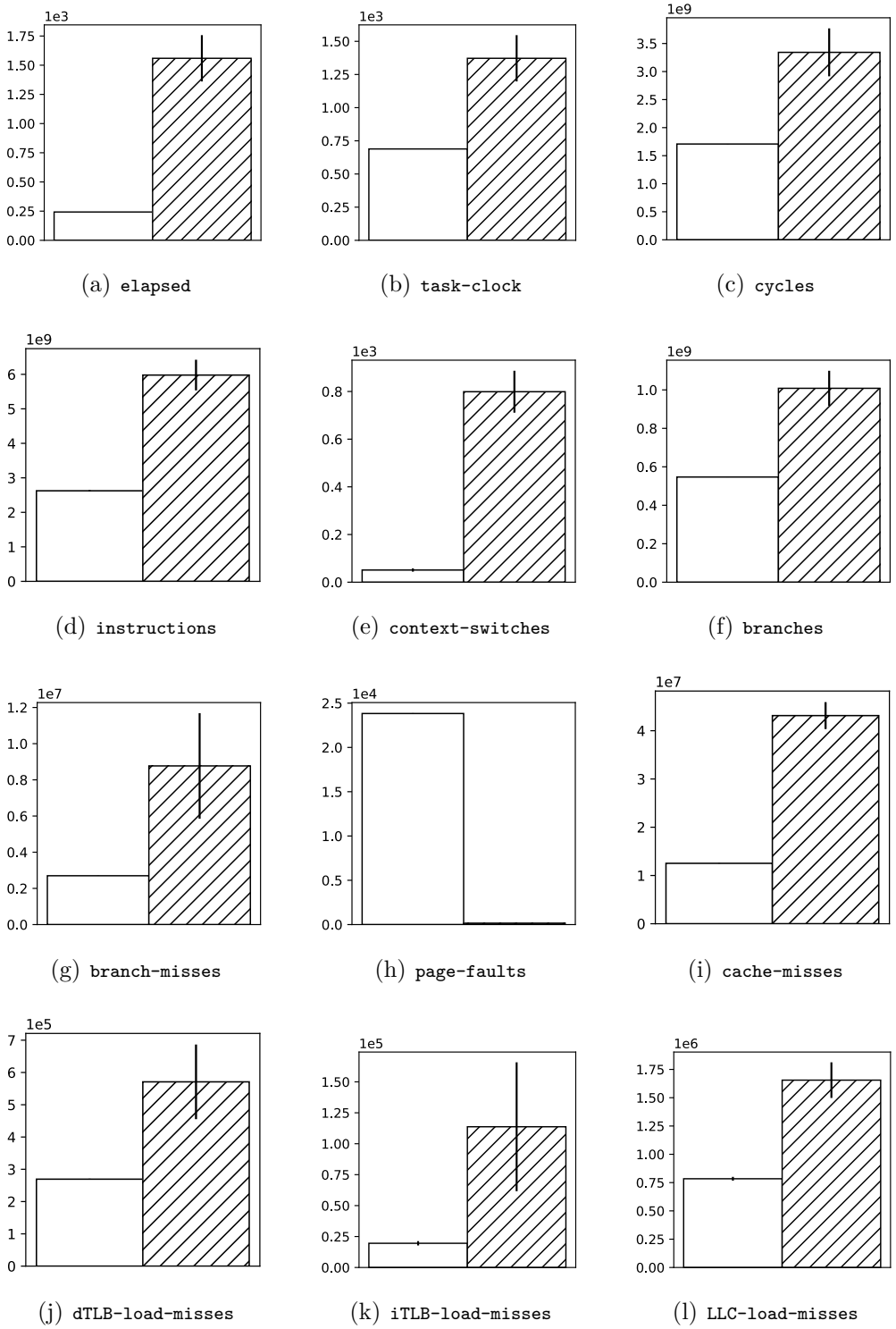
6.1 Level 1

The storage structures of the two benchmarked databases are markedly different, as we have seen in sections 3.2.1 and 3.2.2. This, combined with the different query languages, provide some interesting results.

For the first level, PostgreSQL is clearly the fastest of the two. It executes in 243 ms, which is less than a fifth of Neo4j's 1.5 s. Some of this is explained by PostgreSQL's parallel execution of parts of the query, and we can indeed see in figure 6.1(b) that the difference in aggregated CPU time is a lot smaller.

Another noteworthy point, is can be noticed in figure 6.1(h). PostgreSQL has many orders of magnitude more page faults. By examining table 4.3, however, we see that none of them are "major". These minor page faults are caused by PostgreSQL's reliance on the operating system's file cache [42], and a minor page fault will be triggered every time PostgreSQL needs something from the operating system's cache.

Even though PostgreSQL is faster, we see that because of PostgreSQL's use of a nested loop join, it has a complexity of $\mathcal{O}Nlog(N)$ in the size of the events table, while Neo4j is linear. However, it is likely that the PostgreSQL optimizer would choose a different strategy if for a bigger data set, so it would be unwise to attribute too much to this fact.



(a) `elapsed` (b) `task-clock` (c) `cycles`

(d) `instructions` (e) `context-switches` (f) `branches`

(g) `branch-misses` (h) `page-faults` (i) `cache-misses`

(j) `dTLB-load-misses` (k) `iTLB-load-misses` (l) `LLC-load-misses`

Figure 6.1: Comparison of mean event counts for Query 1 after 5 runs. White is PostgreSQL and striped is Neo4j. The error bars show the standard deviation.

6.2 Level 2

At the second level, PostgreSQL is still the fastest of the two, although not nearly by as much as in the previous query. It executes in 949 ms, which is just a little over 60% of Neo4j's 1.5 s. The fact that PostgreSQL doesn't make any use of its parallel queries is likely a great contributor to the drop in the difference between the two, although there are some other points of note.

We see in figure 6.2(d) that PostgreSQL actually executes about the same number of instructions as Neo4j. It's hard to tell exactly why this is, but what is clear is that PostgreSQL manages to execute a lot more instructions per cycle. Another point of note, is that PostgreSQL has a lot more last level cache misses, as we see in figure 6.2(i), and that this relation was the other way around for the previous query.

Unlike the last query, this query is linear for both approaches, with only the constant factors tilting the results in PostgreSQL's favor.

6.3 Level 3

Again in the third level, PostgreSQL is the fastest. It finishes the query in 7 seconds, while Neo4j uses 26 seconds, which is over three times as slow.

Neo4j seems to have a problem with imposing filters on nodes that are along a variable length path. As we noted in section 5.4.3, we were unable to construct a query that traversed the graph from a possession switch while actually stopping when it reached a new possession change. This means that it continues walking until the end of the period, and only filters the sub-graphs by `team_id` *after* they are all constructed.

This leads to heavier memory usage, as we can see when comparing the 22.3 billion memory store events in 5.3 with PostgreSQL's 5.7 billion in 4.7. It also means that it has to do a lot more work than necessary to filter the sub-graphs.

We also note the difference in context switches that we see in 6.3(e), which is also very prominent in the two other queries. This is likely also a reason for the difference in performance, as context switching is an expensive operation.

Neo4j's problems with enforcing a node filter while traversing the graph makes PostgreSQL the clearly best choice for this query.

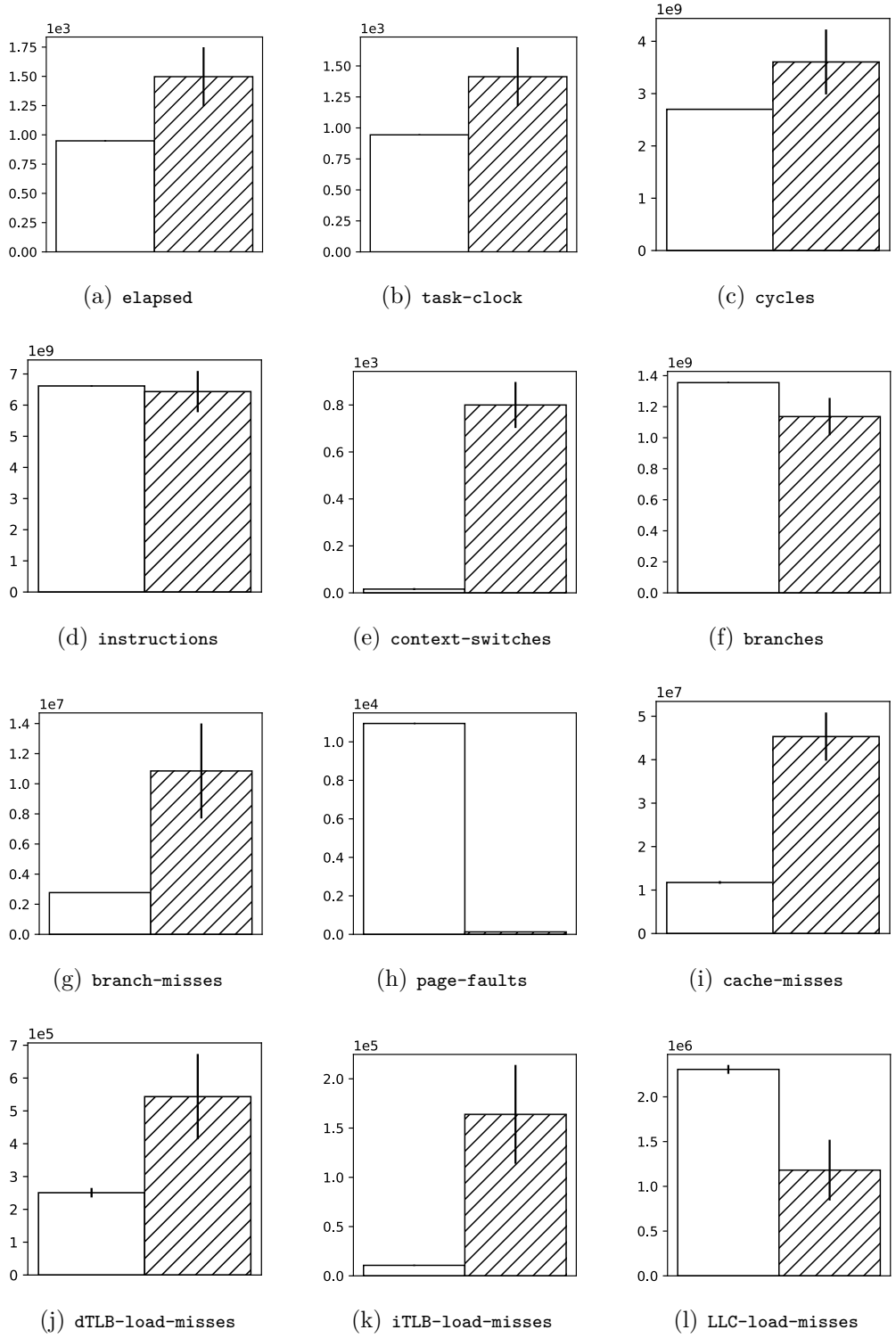
6.4 Summary

In this chapter, we took the best of our PostgreSQL schemas, and compared its benchmark results with those of Neo4j.

We found PostgreSQL to be be the more efficient choice in all three of our queries, although we admit that the poor Neo4j performance in the last level

is because we were unable to construct a query that filtered while traversing.

In conclusion, we believe that the Opta event data, and specifically the operation of finding patterns in the events, may not be a great match for modelling as a graph.



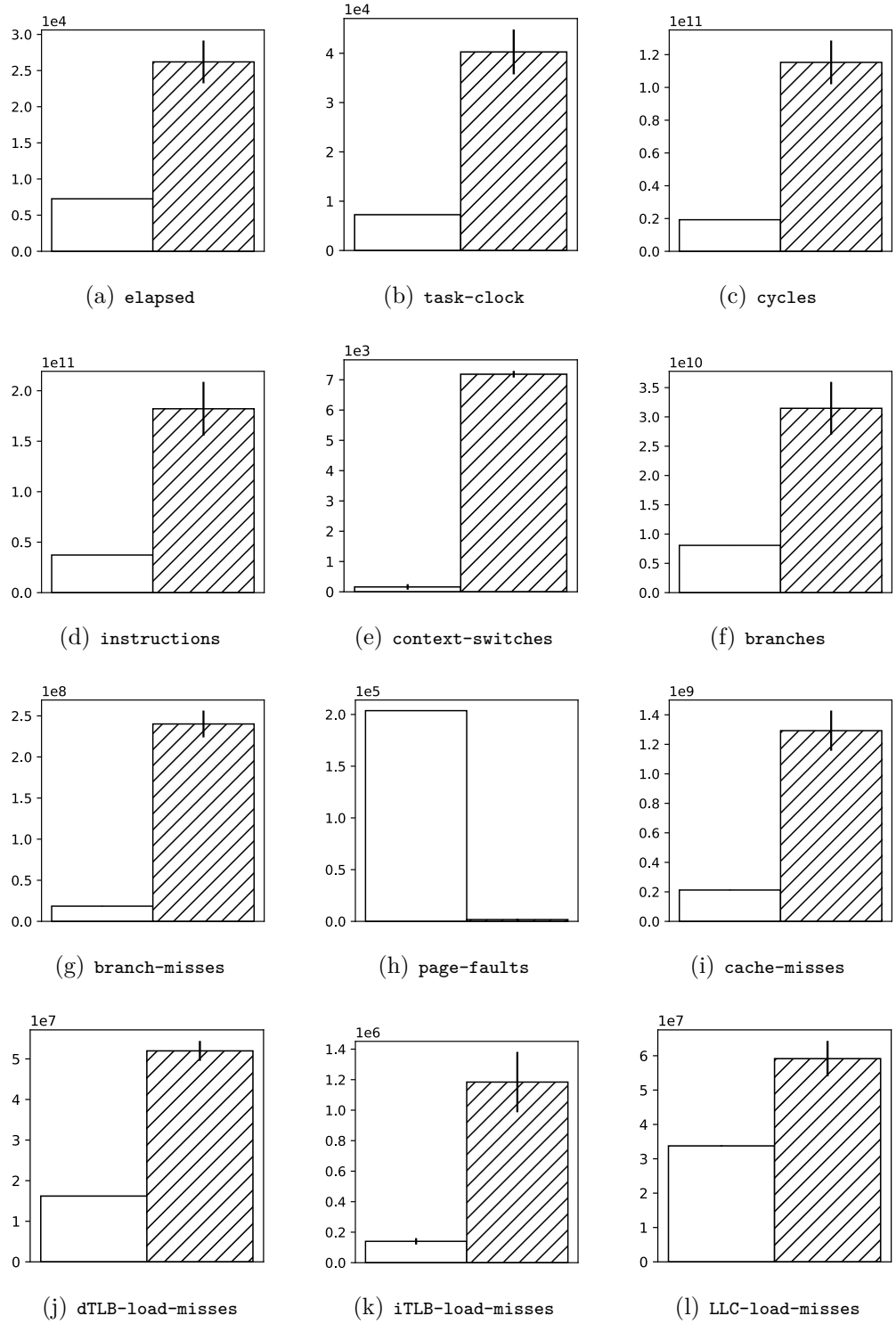
(a) `elapsed` (b) `task-clock` (c) `cycles`

(d) `instructions` (e) `context-switches` (f) `branches`

(g) `branch-misses` (h) `page-faults` (i) `cache-misses`

(j) `dTLB-load-misses` (k) `iTLB-load-misses` (l) `LLC-load-misses`

Figure 6.2: Comparison of mean event counts for Query 2 after 5 runs. White is PostgreSQL and striped is Neo4j. The error bars show the standard deviation.



(a) `elapsed` (b) `task-clock` (c) `cycles`

(d) `instructions` (e) `context-switches` (f) `branches`

(g) `branch-misses` (h) `page-faults` (i) `cache-misses`

(j) `dTLB-load-misses` (k) `iTLB-load-misses` (l) `LLC-load-misses`

Figure 6.3: Comparison of mean event counts for Query 3 after 5 runs. White is PostgreSQL and striped is Neo4j. The error bars show the standard deviation.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we have examined options for processing Opta's football event data efficiently, and whether it is possible to extract higher-level semantics by looking at patterns of simple events.

We first looked at the current state of football analytics, and thoroughly analyzed the structure of Opta's XML data. Then we looked at different types of database systems and their merits and ways of measuring their performance. Further, we examined some previous work in the field of mapping XML data to relational databases, and on benchmarking and comparing databases.

We then designed a benchmark for comparing databases for the specific use of modelling and analyzing Opta's event data, and gave an overview of its structure. We designed three queries of different types: A simple aggregating query, a query for identifying the compound event of a corner kick being cleared by a defender heading, and lastly a query for identifying the unbounded higher-level event of a counter-attack.

Next, we designed two different relational schemas in PostgreSQL, and one schema in the graph database Neo4j, we constructed queries for our three levels, and ran our benchmark on the different approaches.

Finally, we analyzed the results of the benchmark, and specifically compared the best of our relational approach to the graph approach.

7.2 Contributions

Through this thesis, we have answered the questions we posed in our problem definition. We have established that it is in fact possible to extract higher-level events from the simpler ones using either of a relational or graph based approach. Specifically, we have shown that we can identify the two compound event types "counter-attack" and "corners cleared by defender heading". We

have also shown that this is possible to do efficiently by using a relational approach and querying with recursive common table expressions.

Further, through evaluating the performance through our benchmark, we have shown PostgreSQL to be the most performant choice for storing and querying this type of data. In our first query, we found PostgreSQL to outperform Neo4j by a factor of 5 in execution time. In the second, Neo4j is about 60% slower, and in the third it was three times as slow.

7.3 Future Work

We have only benchmarked two databases in this thesis, although there are many more that might be interesting to benchmark for this type of data. Specifically, we think that Esper [43], a database built for event series and pattern matching, could be a good match for the problem of identifying higher-level events in the data.

Also, SQL:2016 means the SQL standard now includes "row pattern matching". This is currently only available in Oracle [44], but will likely make its way to other databases in the future.

Besides databases with pattern matching abilities, we think it would be interesting to compare our row based PostgreSQL results with a column store, possibly the PostgreSQL extension `cstore_fdw` [45].

Appendix A

Feed Examples

Listing A.1: The beginning of an F40 squad feed

```
<SoccerFeed timestamp="20160531T123211+0000">
  <SoccerDocument Type="SQUADS Latest" competition_code="
     EU_CL" competition_id="5" competition_name="Champions
     League" season_id="2015" season_name="Season 2015/2016">
    <Team city="London" country="England" country_id="1"
       country_iso="EN" postal_code="N5 1BU" region_id="17"
       region_name="Europe" short_club_name="Arsenal" street
       ="75 Drayton Park" uID="t3" web_address="http://www.
       arsenal.com">
      <Country>England</Country>
      <Founded>1886</Founded>
      <Name>Arsenal</Name>
      <Player uID="p48844">
        <Name>David Ospina</Name>
        <Position>Goalkeeper</Position>
        <Stat Type="first_name">David</Stat>
        <Stat Type="last_name">Ospina</Stat>
        <Stat Type="birth_date">1988-08-31</Stat>
        <Stat Type="birth_place">Medellín</Stat>
        <Stat Type="first_nationality">Colombia</Stat>
        <Stat Type="weight">80</Stat>
        <Stat Type="height">183</Stat>
        <Stat Type="jersey_num">13</Stat>
        <Stat Type="real_position">Goalkeeper</Stat>
        <Stat Type="real_position_side">Unknown</Stat>
        <Stat Type="join_date">2014-07-28</Stat>
        <Stat Type="country">Colombia</Stat>
      </Player>
      ...
```

Appendix B

Schemas

B.1 PostgreSQL

B.1.1 F9 Tables

Listing B.1: The F9 Tables

```
1  CREATE TABLE matches (
2          id                                  INTEGER PRIMARY KEY,
3          competition_id                      INTEGER REFERENCES
               competitions(id),
4          season_id                           INTEGER REFERENCES
               seasons(id),
5          country                             TEXT,
6          weather                             TEXT,
7          match_type                          TEXT,
8          result_type                         TEXT,
9          attendance                          INTEGER,
10         matchday                            INTEGER,
11         winner                              INTEGER REFERENCES
               teams(id),
12         kickoff                             TIMESTAMP WITH TIME
               ZONE,
13         first_half_extra_start              TEXT,
14         first_half_extra_stop               TEXT,
15         first_half_extra_time               INTEGER,
16         first_half_start                    TEXT,
17         first_half_stop                     TEXT,
18         first_half_time                     INTEGER,
19         match_time                          INTEGER,
20         second_half_extra_start             TEXT,
21         second_half_extra_stop              TEXT,
22         second_half_extra_time              INTEGER,
23         second_half_start                   TEXT,
24         second_half_stop                    TEXT,
25         second_half_time                    INTEGER
26 );
27
28 CREATE TABLE player_results (
```

```
29      id                              SERIAL PRIMARY KEY,
30      player_id                       INTEGER REFERENCES
           players(id),
31      match_id                        INTEGER REFERENCES
           matches(id),
32      team_id                         INTEGER REFERENCES
           teams(id),
33      shirt_number                    INTEGER,
34      position                        TEXT,
35      sub_position                    TEXT,
36      status                          TEXT,
37      side                            TEXT,
38      score                           INTEGER,
39      competition_id                  INTEGER REFERENCES
           competitions(id),
40      season_id                       INTEGER REFERENCES
           seasons(id),
41      country                         TEXT,
42      accurate_back_zone_pass         INTEGER,
43      accurate_chipped_pass           INTEGER,
44      accurate_corners_intobox        INTEGER,
45      accurate_cross                  INTEGER,
46      accurate_cross_nocorner         INTEGER,
47      accurate_flick_on               INTEGER,
48      accurate_freekick_cross         INTEGER,
49      accurate_fwd_zone_pass          INTEGER,
50      accurate_goal_kicks             INTEGER,
51      accurate_keeper_sweeper         INTEGER,
52      accurate_keeper_throws          INTEGER,
53      accurate_launches               INTEGER,
54      accurate_layoffs                INTEGER,
55      accurate_long_balls             INTEGER,
56      accurate_pass                   INTEGER,
57      accurate_pull_back              INTEGER,
58      accurate_through_ball           INTEGER,
59      accurate_throws                 INTEGER,
60      aerial_lost                     INTEGER,
61      aerial_won                      INTEGER,
62      assist_attempt_saved            INTEGER,
63      assist_blocked_shot             INTEGER,
64      assist_free_kick_won            INTEGER,
65      assist_handball_won             INTEGER,
66      assist_own_goal                 INTEGER,
67      assist_pass_lost                INTEGER,
68      assist_penalty_won              INTEGER,
69      assist_post                     INTEGER,
70      att_assist_openplay             INTEGER,
71      att_assist_setplay              INTEGER,
72      att_bx_centre                   INTEGER,
73      att_bx_left                     INTEGER,
74      att_bx_right                    INTEGER,
75      att_cmiss_high                  INTEGER,
76      att_cmiss_high_left             INTEGER,
77      att_cmiss_high_right            INTEGER,
```

```
78        att_cmiss_left                  INTEGER,
79        att_cmiss_right                 INTEGER,
80        att_fastbreak                   INTEGER,
81        att_freekick_goal               INTEGER,
82        att_freekick_miss               INTEGER,
83        att_freekick_post               INTEGER,
84        att_freekick_target             INTEGER,
85        att_freekick_total              INTEGER,
86        att_goal_high_centre            INTEGER,
87        att_goal_high_left              INTEGER,
88        att_goal_high_right             INTEGER,
89        att_goal_low_centre             INTEGER,
90        att_goal_low_left               INTEGER,
91        att_goal_low_right              INTEGER,
92        att_hd_goal                     INTEGER,
93        att_hd_miss                     INTEGER,
94        att_hd_post                     INTEGER,
95        att_hd_target                   INTEGER,
96        att_hd_total                    INTEGER,
97        att_ibox_blocked                INTEGER,
98        att_ibox_goal                   INTEGER,
99        att_ibox_miss                   INTEGER,
100       att_ibox_post                   INTEGER,
101       att_ibox_target                 INTEGER,
102       att_lf_goal                     INTEGER,
103       att_lf_target                   INTEGER,
104       att_lf_total                    INTEGER,
105       att_lg_centre                   INTEGER,
106       att_lg_left                     INTEGER,
107       att_lg_right                    INTEGER,
108       att_miss_high                   INTEGER,
109       att_miss_high_left              INTEGER,
110       att_miss_high_right             INTEGER,
111       att_miss_left                   INTEGER,
112       att_miss_right                  INTEGER,
113       att_obox_blocked                INTEGER,
114       att_obox_goal                   INTEGER,
115       att_obox_miss                   INTEGER,
116       att_obox_post                   INTEGER,
117       att_obox_target                 INTEGER,
118       att_obp_goal                    INTEGER,
119       att_obx_centre                  INTEGER,
120       att_obx_left                    INTEGER,
121       att_obx_right                   INTEGER,
122       att_obxd_left                   INTEGER,
123       att_obxd_right                  INTEGER,
124       att_one_on_one                  INTEGER,
125       att_openplay                    INTEGER,
126       att_pen_goal                    INTEGER,
127       att_pen_miss                    INTEGER,
128       att_pen_post                    INTEGER,
129       att_pen_target                  INTEGER,
130       att_post_high                   INTEGER,
131       att_post_left                   INTEGER,
```

```
    att_post_right                  INTEGER,
    att_rf_goal                     INTEGER,
    att_rf_target                   INTEGER,
    att_rf_total                    INTEGER,
    att_setpiece                    INTEGER,
    att_sv_high_centre              INTEGER,
    att_sv_high_left                INTEGER,
    att_sv_high_right               INTEGER,
    att_sv_low_centre               INTEGER,
    att_sv_low_left                 INTEGER,
    att_sv_low_right                INTEGER,
    attempted_tackle_foul           INTEGER,
    attempts_conceded_ibox          INTEGER,
    attempts_conceded_obox          INTEGER,
    back_pass                       INTEGER,
    backward_pass                   INTEGER,
    ball_recovery                   INTEGER,
    big_chance_created              INTEGER,
    big_chance_missed               INTEGER,
    big_chance_scored               INTEGER,
    blocked_cross                   INTEGER,
    blocked_pass                    INTEGER,
    blocked_scoring_att             INTEGER,
    challenge_lost                  INTEGER,
    clean_sheet                     INTEGER,
    clearance_off_line              INTEGER,
    corner_taken                    INTEGER,
    cross_not_claimed               INTEGER,
    crosses_18yard                  INTEGER,
    crosses_18yardplus              INTEGER,
    dangerous_play                  INTEGER,
    dispossessed                    INTEGER,
    dive_catch                      INTEGER,
    dive_save                       INTEGER,
    diving_save                     INTEGER,
    duel_lost                       INTEGER,
    duel_won                        INTEGER,
    effective_blocked_cross         INTEGER,
    effective_clearance             INTEGER,
    effective_head_clearance        INTEGER,
    error_lead_to_goal              INTEGER,
    error_lead_to_shot              INTEGER,
    final_third_entries             INTEGER,
    first_half_goals                INTEGER,
    formation_place                 INTEGER,
    formation_used                  INTEGER,
    foul_throw_in                   INTEGER,
    fouled_final_third              INTEGER,
    fouls                           INTEGER,
    freekick_cross                  INTEGER,
    fwd_pass                        INTEGER,
    game_started                    INTEGER,
    gk_smother                      INTEGER,
    goal_assist                     INTEGER,
```

```
    goal_assist_deadball            INTEGER,
    goal_assist_intentional         INTEGER,
    goal_assist_openplay            INTEGER,
    goal_assist_setplay             INTEGER,
    goal_fastbreak                  INTEGER,
    goal_kicks                      INTEGER,
    goals                           INTEGER,
    goals_conceded                  INTEGER,
    goals_conceded_ibox             INTEGER,
    goals_conceded_obox             INTEGER,
    goals_openplay                  INTEGER,
    good_high_claim                 INTEGER,
    hand_ball                       INTEGER,
    head_clearance                  INTEGER,
    head_pass                       INTEGER,
    hit_woodwork                    INTEGER,
    interception                    INTEGER,
    interception_won                INTEGER,
    interceptions_in_box            INTEGER,
    keeper_pick_up                  INTEGER,
    keeper_throws                   INTEGER,
    last_man_tackle                 INTEGER,
    leftside_pass                   INTEGER,
    long_pass_own_to_opp            INTEGER,
    long_pass_own_to_opp_success    INTEGER,
    lost_corners                    INTEGER,
    mins_played                     INTEGER,
    offside_provoked                INTEGER,
    offtarget_att_assist            INTEGER,
    ontarget_att_assist             INTEGER,
    ontarget_scoring_att            INTEGER,
    open_play_pass                  INTEGER,
    outfielder_block                INTEGER,
    overrun                         INTEGER,
    own_goals                       INTEGER,
    passes_left                     INTEGER,
    passes_right                    INTEGER,
    pen_area_entries                INTEGER,
    pen_goals_conceded              INTEGER,
    penalty_conceded                INTEGER,
    penalty_faced                   INTEGER,
    penalty_save                    INTEGER,
    penalty_won                     INTEGER,
    poss_lost_all                   INTEGER,
    poss_lost_ctrl                  INTEGER,
    poss_won_att_3rd                INTEGER,
    poss_won_def_3rd                INTEGER,
    poss_won_mid_3rd                INTEGER,
    post_scoring_att                INTEGER,
    pts_dropped_winning_pos         INTEGER,
    pts_gained_losing_pos           INTEGER,
    punches                         INTEGER,
    put_through                     INTEGER,
    red_card                        INTEGER,
```

```
        rightside_pass                  INTEGER,
        saved_ibox                      INTEGER,
        saved_obox                      INTEGER,
        saves                           INTEGER,
        second_goal_assist              INTEGER,
        second_yellow                   INTEGER,
        shield_ball_oop                 INTEGER,
        shot_fastbreak                  INTEGER,
        shot_off_target                 INTEGER,
        six_second_violation            INTEGER,
        six_yard_block                  INTEGER,
        stand_catch                     INTEGER,
        stand_save                      INTEGER,
        successful_final_third_passes   INTEGER,
        successful_open_play_pass       INTEGER,
        successful_put_through          INTEGER,
        total_att_assist                INTEGER,
        total_back_zone_pass            INTEGER,
        total_chipped_pass              INTEGER,
        total_clearance                 INTEGER,
        total_contest                   INTEGER,
        total_corners_intobox           INTEGER,
        total_cross                     INTEGER,
        total_cross_nocorner            INTEGER,
        total_fastbreak                 INTEGER,
        total_final_third_passes        INTEGER,
        total_flick_on                  INTEGER,
        total_fwd_zone_pass             INTEGER,
        total_high_claim                INTEGER,
        total_keeper_sweeper            INTEGER,
        total_launches                  INTEGER,
        total_layoffs                   INTEGER,
        total_long_balls                INTEGER,
        total_offside                   INTEGER,
        total_pass                      INTEGER,
        total_pull_back                 INTEGER,
        total_scoring_att               INTEGER,
        total_sub_off                   INTEGER,
        total_sub_on                    INTEGER,
        total_tackle                    INTEGER,
        total_through_ball              INTEGER,
        total_throws                    INTEGER,
        touches                         INTEGER,
        touches_in_opp_box              INTEGER,
        turnover                        INTEGER,
        unsuccessful_touch              INTEGER,
        was_fouled                      INTEGER,
        won_contest                     INTEGER,
        won_corners                     INTEGER,
        won_tackle                      INTEGER,
        yellow_card                     INTEGER
);

CREATE TABLE team_results (
```

```
294         id                                  SERIAL PRIMARY KEY,
295         team_id                             INTEGER REFERENCES
               teams(id),
296         side                                TEXT,
297         score                               INTEGER,
298         match_id                            INTEGER REFERENCES
               matches(id),
299         competition_id                      INTEGER REFERENCES
               competitions(id),
300         season_id                           INTEGER REFERENCES
               seasons(id),
301         accurate_back_zone_pass             INTEGER,
302         accurate_chipped_pass               INTEGER,
303         accurate_corners_intobox            INTEGER,
304         accurate_cross                      INTEGER,
305         accurate_cross_nocorner             INTEGER,
306         accurate_flick_on                   INTEGER,
307         accurate_freekick_cross             INTEGER,
308         accurate_fwd_zone_pass              INTEGER,
309         accurate_goal_kicks                 INTEGER,
310         accurate_keeper_sweeper             INTEGER,
311         accurate_keeper_throws              INTEGER,
312         accurate_launches                   INTEGER,
313         accurate_layoffs                    INTEGER,
314         accurate_long_balls                 INTEGER,
315         accurate_pass                       INTEGER,
316         accurate_pull_back                  INTEGER,
317         accurate_through_ball               INTEGER,
318         accurate_throws                     INTEGER,
319         aerial_lost                         INTEGER,
320         aerial_won                          INTEGER,
321         att_assist_openplay                 INTEGER,
322         att_assist_setplay                  INTEGER,
323         att_bx_centre                       INTEGER,
324         att_bx_left                         INTEGER,
325         att_bx_right                        INTEGER,
326         att_cmiss_high                      INTEGER,
327         att_cmiss_high_left                 INTEGER,
328         att_cmiss_high_right                INTEGER,
329         att_cmiss_left                      INTEGER,
330         att_cmiss_right                     INTEGER,
331         att_fastbreak                       INTEGER,
332         att_freekick_goal                   INTEGER,
333         att_freekick_miss                   INTEGER,
334         att_freekick_post                   INTEGER,
335         att_freekick_target                 INTEGER,
336         att_freekick_total                  INTEGER,
337         att_goal_high_centre                INTEGER,
338         att_goal_high_left                  INTEGER,
339         att_goal_high_right                 INTEGER,
340         att_goal_low_centre                 INTEGER,
341         att_goal_low_left                   INTEGER,
342         att_goal_low_right                  INTEGER,
343         att_hd_goal                         INTEGER,
```

```
344         att_hd_miss                     INTEGER,
345         att_hd_post                     INTEGER,
346         att_hd_target                   INTEGER,
347         att_hd_total                    INTEGER,
348         att_ibox_blocked                INTEGER,
349         att_ibox_goal                   INTEGER,
350         att_ibox_miss                   INTEGER,
351         att_ibox_own_goal               INTEGER,
352         att_ibox_post                   INTEGER,
353         att_ibox_target                 INTEGER,
354         att_lf_goal                     INTEGER,
355         att_lf_target                   INTEGER,
356         att_lf_total                    INTEGER,
357         att_lg_centre                   INTEGER,
358         att_lg_left                     INTEGER,
359         att_lg_right                    INTEGER,
360         att_miss_high                   INTEGER,
361         att_miss_high_left              INTEGER,
362         att_miss_high_right             INTEGER,
363         att_miss_left                   INTEGER,
364         att_miss_right                  INTEGER,
365         att_obox_blocked                INTEGER,
366         att_obox_goal                   INTEGER,
367         att_obox_miss                   INTEGER,
368         att_obox_own_goal               INTEGER,
369         att_obox_post                   INTEGER,
370         att_obox_target                 INTEGER,
371         att_obp_goal                    INTEGER,
372         att_obx_centre                  INTEGER,
373         att_obx_left                    INTEGER,
374         att_obx_right                   INTEGER,
375         att_obxd_left                   INTEGER,
376         att_obxd_right                  INTEGER,
377         att_one_on_one                  INTEGER,
378         att_openplay                    INTEGER,
379         att_pen_goal                    INTEGER,
380         att_pen_miss                    INTEGER,
381         att_pen_post                    INTEGER,
382         att_pen_target                  INTEGER,
383         att_post_high                   INTEGER,
384         att_post_left                   INTEGER,
385         att_post_right                  INTEGER,
386         att_rf_goal                     INTEGER,
387         att_rf_target                   INTEGER,
388         att_rf_total                    INTEGER,
389         att_setpiece                    INTEGER,
390         att_sv_high_centre              INTEGER,
391         att_sv_high_left                INTEGER,
392         att_sv_high_right               INTEGER,
393         att_sv_low_centre               INTEGER,
394         att_sv_low_left                 INTEGER,
395         att_sv_low_right                INTEGER,
396         attempted_tackle_foul           INTEGER,
397         attempts_conceded_ibox          INTEGER,
```

```
    attempts_conceded_obox          INTEGER,
    backward_pass                   INTEGER,
    ball_recovery                   INTEGER,
    big_chance_created              INTEGER,
    big_chance_missed               INTEGER,
    big_chance_scored               INTEGER,
    blocked_cross                   INTEGER,
    blocked_pass                    INTEGER,
    blocked_scoring_att             INTEGER,
    challenge_lost                  INTEGER,
    clearance_off_line              INTEGER,
    contentious_decision            INTEGER,
    corner_taken                    INTEGER,
    crosses_18yard                  INTEGER,
    crosses_18yardplus              INTEGER,
    defender_goals                  INTEGER,
    dispossessed                    INTEGER,
    diving_save                     INTEGER,
    duel_lost                       INTEGER,
    duel_won                        INTEGER,
    effective_blocked_cross         INTEGER,
    effective_clearance             INTEGER,
    effective_head_clearance        INTEGER,
    error_lead_to_goal              INTEGER,
    error_lead_to_shot              INTEGER,
    final_third_entries             INTEGER,
    fk_foul_lost                    INTEGER,
    fk_foul_won                     INTEGER,
    forward_goals                   INTEGER,
    foul_throw_in                   INTEGER,
    fouled_final_third              INTEGER,
    freekick_cross                  INTEGER,
    fwd_pass                        INTEGER,
    goal_assist                     INTEGER,
    goal_assist_deadball            INTEGER,
    goal_assist_intentional         INTEGER,
    goal_assist_openplay            INTEGER,
    goal_assist_setplay             INTEGER,
    goal_fastbreak                  INTEGER,
    goal_kicks                      INTEGER,
    goals                           INTEGER,
    goals_conceded                  INTEGER,
    goals_conceded_ibox             INTEGER,
    goals_conceded_obox             INTEGER,
    goals_openplay                  INTEGER,
    good_high_claim                 INTEGER,
    hand_ball                       INTEGER,
    head_clearance                  INTEGER,
    hit_woodwork                    INTEGER,
    interception                    INTEGER,
    interception_won                INTEGER,
    interceptions_in_box            INTEGER,
    keeper_throws                   INTEGER,
    last_man_tackle                 INTEGER,
```

```
452         leftside_pass                       INTEGER,
453         long_pass_own_to_opp                INTEGER,
454         long_pass_own_to_opp_success        INTEGER,
455         lost_corners                        INTEGER,
456         midfielder_goals                    INTEGER,
457         offtarget_att_assist                INTEGER,
458         ontarget_att_assist                 INTEGER,
459         ontarget_scoring_att                INTEGER,
460         open_play_pass                      INTEGER,
461         outfielder_block                    INTEGER,
462         overrun                             INTEGER,
463         own_goal_accrued                    INTEGER,
464         own_goals                           INTEGER,
465         passes_left                         INTEGER,
466         passes_right                        INTEGER,
467         pen_area_entries                    INTEGER,
468         pen_goals_conceded                  INTEGER,
469         penalty_conceded                    INTEGER,
470         penalty_faced                       INTEGER,
471         penalty_save                        INTEGER,
472         penalty_won                         INTEGER,
473         poss_lost_all                       INTEGER,
474         poss_lost_ctrl                      INTEGER,
475         poss_won_att_3rd                    INTEGER,
476         poss_won_def_3rd                    INTEGER,
477         poss_won_mid_3rd                    INTEGER,
478         possession_percentage               FLOAT,
479         post_scoring_att                    INTEGER,
480         punches                             INTEGER,
481         put_through                         INTEGER,
482         rightside_pass                      INTEGER,
483         saved_ibox                          INTEGER,
484         saved_obox                          INTEGER,
485         saves                               INTEGER,
486         second_yellow                       INTEGER,
487         shield_ball_oop                     INTEGER,
488         shot_fastbreak                      INTEGER,
489         shot_off_target                     INTEGER,
490         six_yard_block                      INTEGER,
491         subs_made                           INTEGER,
492         successful_final_third_passes       INTEGER,
493         successful_open_play_pass           INTEGER,
494         successful_put_through              INTEGER,
495         total_att_assist                    INTEGER,
496         total_back_zone_pass                INTEGER,
497         total_chipped_pass                  INTEGER,
498         total_clearance                     INTEGER,
499         total_contest                       INTEGER,
500         total_corners_intobox               INTEGER,
501         total_cross                         INTEGER,
502         total_cross_nocorner                INTEGER,
503         total_fastbreak                     INTEGER,
504         total_final_third_passes            INTEGER,
505         total_flick_on                      INTEGER,
```

```
        total_fwd_zone_pass             INTEGER,
        total_high_claim                INTEGER,
        total_keeper_sweeper            INTEGER,
        total_launches                  INTEGER,
        total_layoffs                   INTEGER,
        total_long_balls                INTEGER,
        total_offside                   INTEGER,
        total_pass                      INTEGER,
        total_pull_back                 INTEGER,
        total_red_card                  INTEGER,
        total_scoring_att               INTEGER,
        total_tackle                    INTEGER,
        total_through_ball              INTEGER,
        total_throws                    INTEGER,
        total_yel_card                  INTEGER,
        touches                         INTEGER,
        touches_in_opp_box              INTEGER,
        unsuccessful_touch              INTEGER,
        won_contest                     INTEGER,
        won_corners                     INTEGER,
        won_tackle                      INTEGER
);
```

B.1.2 F40 Tables

Listing B.2: The F40 Tables

```
CREATE TABLE teams (
        id                      INTEGER PRIMARY KEY,
        name                    TEXT NOT NULL,
        short_club_name         TEXT,
        official_club_name      TEXT,
        founded                 INTEGER,
        city                    TEXT,
        country                 TEXT,
        region                  TEXT
);

CREATE TABLE players (
        id                      INTEGER PRIMARY KEY,
        name                    TEXT NOT NULL,
        first_name              TEXT,
        middle_name             TEXT,
        last_name               TEXT,
        known_name              TEXT,
        birth_place             TEXT,
        birth_date              DATE,
        deceased                TEXT,
        weight                  INTEGER,
        height                  INTEGER,
        preferred_foot          TEXT,
        country                 TEXT,
```

```
26        first_nationality    TEXT
27  );
28
29  CREATE TABLE competitions (
30        id                   INTEGER PRIMARY KEY,
31        name                 TEXT NOT NULL,
32        symid                TEXT,
33        code                 TEXT NOT NULL
34  );
35
36  CREATE TABLE seasons (
37        id                   INTEGER PRIMARY KEY,
38        name                 TEXT NOT NULL
39  );
40
41  CREATE TABLE squad_entries (
42        id                   SERIAL PRIMARY KEY,
43        player_id            INTEGER REFERENCES players(id),
44        team_id              INTEGER REFERENCES teams(id),
45        season_id            INTEGER REFERENCES seasons(id),
46        competition_id       INTEGER REFERENCES competitions(
              id),
47        position             TEXT,
48        real_position        TEXT,
49        real_position_side   TEXT,
50        new_team             TEXT,
51        join_date            DATE,
52        leave_date           DATE,
53        jersey_num           INTEGER
54  );
```

Bibliography

[1] D. E. Comer et al. "Computing As a Discipline." In: *Commun. ACM* 32.1 (Jan. 1989). Ed. by Peter J. Denning, pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: http://doi.acm.org/10.1145/63238.63239.

[2] M.M. Lewis. *Moneyball: The Art of Winning an Unfair Game*. Norton paperback. W.W. Norton, 2003. ISBN: 9780393057652.

[3] MIT Sloan Sports Analytics Conference. *MIT Sloan Sports Analytics Conference*. URL: http://www.sloansportsconference.com/ (visited on 07/28/2017).

[4] Vumero Institute. *Sports Analytics World Conferences 2017/2018*. URL: https://www.analyticsinsport.com/ (visited on 07/28/2017).

[5] Dimitris Karlis and Ioannis Ntzoufras. "Analysis of sports data by using bivariate Poisson models." In: *Journal of the Royal Statistical Society: Series D (The Statistician)* 52.3 (2003), pp. 381–393.

[6] Neil Charles. *Find me a player like Andrés Iniesta*. URL: http://www.hilltop-analytics.com/football/find-me-a-player-like-andres-iniesta/ (visited on 07/28/2017).

[7] STATS LLC. *STATS SportVU Football Player Tracking*. URL: https://www.stats.com/sportvu-football/ (visited on 07/24/2017).

[8] Simen Sægrov et al. "BAGADUS: An Integrated System for Soccer Analysis (demo)." In: *Proc. of ICDSC*. Hong Kong, Oct. 2012.

[9] Pål Halvorsen et al. "BAGADUS: An Integrated System for Arena Sports Analytics – A Soccer Case Study." In: *Proc. of ACM MMSys*. Oslo, Norway, Mar. 2013, pp. 48–59.

[10] Håkon Kvale Stensland et al. "Bagadus: An Integrated Real-Time System for Soccer Analytics." In: *ACM TOMCCAP* 10.1s (Jan. 2014).

[11] ChyronHego Corporation. *ZXY Wearable Tracking*. URL: http://chyronhego.com/sports-data/zxy (visited on 07/24/2017).

[12] Tim Bray et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. URL: https://www.w3.org/TR/REC-xml/ (visited on 07/27/2017).

[13] Opta Sports. *Opta Documentation*. Requires login. URL: http://optasports.com/praxis/documentation/ (visited on 07/28/2017).

[14] Solid IT GmbH. *DB-Engines Ranking – Popularity Ranking of Database Management Systems*. URL: https://db-engines.com/en/ranking (visited on 07/28/2017).

[15] Brian McKenna. *Sports performance data analytics fuels Opta's business expansion*. June 6, 2013. URL: http://www.computerweekly.com/news/2240185389/Sports-performance-data-analytics-fuels-Optas-business-expansion (visited on 01/25/2017).

[16] Opta Sports. *Opta Job Vacancies: Software Architect*. 2015. URL: http://www.optasports.com/about/getting-in-touch/job-vacancies.aspx?JobID=25313 (visited on 01/11/2017).

[17] Priti Mishra and Margaret H. Eich. "Join Processing in Relational Databases." In: *ACM Comput. Surv.* 24.1 (Mar. 1992), pp. 63–113. ISSN: 0360-0300. DOI: 10.1145/128762.128764. URL: http://doi.acm.org/10.1145/128762.128764.

[18] Daniela Florescu and Donald Kossmann. "Storing and querying XML data using an RDMBS." In: *IEEE Data Engineering Bulletin, Special Issue on* 1060.22 (1999), p. 3.

[19] Renzo Angles et al. "Benchmarking Database Systems for Social Network Applications." In: *First International Workshop on Graph Data Management Experiences and Systems*. GRADES '13. New York, New York: ACM, 2013, 15:1–15:7. ISBN: 978-1-4503-2188-4. DOI: 10.1145/2484425.2484440. URL: http://doi.acm.org/10.1145/2484425.2484440.

[20] S. Jouili and V. Vansteenberghe. "An Empirical Comparison of Graph Databases." In: *2013 International Conference on Social Computing*. Sept. 2013, pp. 708–715. DOI: 10.1109/SocialCom.2013.106.

[21] Eigil Skjæveland. *thesis-experiments*. URL: https://github.com/eigilhs/thesis-experiments (visited on 07/31/2017).

[22] The PostgreSQL Global Development Group. *PostgreSQL: About*. URL: https://www.postgresql.org/about/ (visited on 01/11/2017).

[23] The PostgreSQL Global Development Group. *PostgreSQL: History*. URL: https://www.postgresql.org/about/history/ (visited on 07/28/2017).

[24] The PostgreSQL Global Development Group. *PostgreSQL Optimizer Source Code*. URL: https://github.com/postgres/postgres/tree/master/src/backend/optimizer (visited on 07/28/2017).

[25] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 10: Chapter 66. Database Physical Storage*. URL: https://www.postgresql.org/docs/10/static/storage.html (visited on 07/28/2017).

[26] Neo Technology. *Neo4j: The World's Leading Graph Database*. URL: https://neo4j.com/product/ (visited on 01/11/2017).

[27] Tobias Lindaaker. *An Overview of Neo4j Internals*. URL: https://www.slideshare.net/thobe/an-overview-of-neo4j-internals (visited on 07/28/2017).

[28] Neo Technology. *Neo4j Data Structures Source Code*. URL: https://github.com/neo4j/neo4j/tree/3.3/community/kernel/src/main/java/org/neo4j/kernel/impl/store/record (visited on 07/28/2017).

[29] Daniele Varrazzo. *Psycopg*. URL: http://initd.org/psycopg/ (visited on 07/24/2017).

[30] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 10: COPY*. URL: https://www.postgresql.org/docs/10/static/sql-copy.html (visited on 07/28/2017).

[31] Neo Technology. *cypher-shell: A command line shell where you can execute Cypher against an instance of Neo4j*. URL: https://github.com/neo4j/cypher-shell (visited on 07/29/2017).

[32] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 10: EXPLAIN*. URL: https://www.postgresql.org/docs/10/static/sql-explain.html (visited on 07/28/2017).

[33] Neo Technlology. *Neo4j Documentation: Profiling a query*. URL: https://neo4j.com/docs/developer-manual/current/cypher/query-tuning/how-do-i-profile-a-query/ (visited on 07/28/2017).

[34] Perf Wiki. *perf: Linux profiling with performance counters*. URL: https://perf.wiki.kernel.org/ (visited on 07/28/2017).

[35] Perf Wiki. *Tutorial*. URL: https://perf.wiki.kernel.org/index.php/Tutorial (visited on 07/28/2017).

[36] The PostgreSQL Global Development Group. *PostgreSQL Source Code: jsonb.h*. URL: https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob_plain;f=src/include/utils/jsonb.h;hb=42171e2cd23c8307bbe0ec64e901f58e297db1c3 (visited on 07/28/2017).

[37] The PostgreSQL Global Development Group. *PostgreSQL Source Code: jsonb_util.c*. URL: https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob_plain;f=src/backend/utils/adt/jsonb_util.c;hb=42171e2cd23c8307bbe0ec64e901f58e297db1c3 (visited on 07/28/2017).

[38] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 10: 7.8. WITH Queries (Common Table Expressions)*. URL: https://www.postgresql.org/docs/10/static/queries-with.html (visited on 07/28/2017).

[39] Neo Technlology. *Warm the Cache to Improve Performance from Cold Start*. URL: https://neo4j.com/developer/kb/warm-the-cache-to-improve-performance-from-cold-start/ (visited on 07/28/2017).

[40] Neo Technology. *Cypher Query Language*. URL: https://neo4j.com/developer/cypher/ (visited on 07/11/2017).

[41] Neo Technology. *What is openCypher?* URL: http://www.opencypher.org/ (visited on 07/11/2017).

[42] Greg Smith, Robert Treat, and Christopher Browne. *Tuning Your PostgreSQL Server*. URL: https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server (visited on 07/28/2017).

[43] EsperTech Inc. *Esper: Event Processing for Java*. URL: http://www.espertech.com/products/esper.php (visited on 07/28/2017).

[44] Oracle. *SQL for Pattern Matching*. URL: https://docs.oracle.com/database/121/DWHSG/pattern.htm#DWHSG8959 (visited on 07/28/2017).

[45] Citus Data Inc. *cstore_fdw: Columnar Store for Analytics with PostgreSQL*. URL: https://github.com/citusdata/cstore_fdw (visited on 07/28/2017).