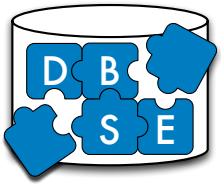
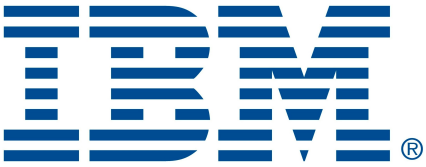
University of Magdeburg

School of Computer Science



Databases
and
Software
Engineering



Master's Thesis

Best Practices for Developing Graph Database Applications: A Case Study Using Apache Titan

Author:

Gabriel Campero Durand

6 January, 2017

Advisors:

Prof. Dr. Gunter Saake
David Broneske, MSc., Marcus Pinnecke, MSc.
University of Magdeburg

Thomas Spatzier, Dipl. Inf.
IBM Research and Development, Böblingen, Germany

Campero Durand, Gabriel:
Best Practices for Developing Graph Database Applications: A Case Study Using Apache Titan
Master's Thesis, University of Magdeburg, 2017.

Abstract

Networks and their interactions permeate every aspect of our lives, from our cells and the ecosphere to planetary systems. In recent years, computational network science has emerged as an active field, bringing forward an abundance of tools that help us gain insights into the interconnected systems that surround us (e.g. the personalized relevance of websites from within the whole Web). Graph databases constitute a modern class of non-relational database systems, tailored to support these network-oriented workloads. With this goal they offer a data model based on graph abstractions (i.e. nodes/vertices and edges), and can adopt different optimizations to speed up essential graph processing patterns, such as traversals.

In spite of their benefits, query optimization remains a significant challenge in these databases. As a result, in today's mainstream products a big part of the responsibility for performance can be shifted to the application developers.

In this research we provide some guidelines to developers for tackling performance with these databases. With this aim we propose simple microbenchmarks of alternative application-level query optimizatons, assessing them experimentally. We run our tests using TitanLab, an Apache Titan prototype, with Gremlin as a query language and Apache Tinkerpop as a graph-processing framework.

Specifically, we study basic selection queries using different global access primitives,further taking into account the effect of adding sorting, counting and pagination. We also take a close look at some potential factors affecting the efficiency of queries with composite indexes (i.e., Titan's native indexes): data cardinality, and Titan's runtime assignation of indexes to queries. We additionally evaluate some potential gains from optimization strategies such as denormalization (as an approach to improve traversals) and the leveraging of search engines for graph functionality. For these optimizations we report performance gains that change a two minutes response into a sub-second one, and a 70 minute function into a 28 seconds one (while using *Pokec*, a public dataset).

We conclude by summarizing best practices for developing graph database applications, as they pertain to our findings. We expect that this work could contribute to the understanding on how queries can be optimized in graph databases, and to the process of standardization of these technologies.

To my mom, Gladys. *O laborum dulce lenimen.*

In memory of my dad, Gustavo; my friends, Conchita and Darita.

And to my second dad, Francisco, who got me into computer science, John Coltrane and The Beatles' White Album.

Acknowledgements

This master thesis was made possible thanks to the sponsorship and support from IBM Research and Development Germany.

First, I would like thank my advisors: Thomas Spatzier is responsible for bringing this project into existence, working with me in evaluating different aspects of using Titan and putting me into contact with knowledgeable IBMers. His was a truly exemplary industrial mentorship. David Broneske and Marcus Pinnecke, my University advisors, offered me continuous guidance, valuable knowledge, healthy doses of realism and practical advice for getting things done. I couldn't have asked for a better team to undertake this research.

I'm greatly indebted to Matt Duggan, for his friendly guidance and valuable input, based on his extensive domain expertise. He also helped me with the synthetic datasets for our tests.

I'd like to thank IBMers Kilian Collender, Andrzej Wrobel, Jason C. Plurad, Kelvin Lawrence, Benjamin Anderson, Maurice Collins, Konrad Ohms, Stephane Millar, Stephen Uridge, Amardeep Kalsi, Jenny Li Kam Wa and Johanna Ang'ani for their generous and meaningful collaboration in different aspects of this research initiative.

In addition, I have to thank the IBM Netcool Network Management team and the IBM Bluemix Availability Monitoring team, for welcoming me into their daily scrums and development tasks during some months, helping me to better understand real-world use cases of graph technologies.

I'm thankful to Dr. Goetz Graefe and the organizers at Schloss Dagstuhl for the opportunity to participate in a week-long Data Management School in August 2016. Apart from the talks, conversations with participants, brainstorming, and even short visits to the local library, contributed to shape this research and hopefully more to come.

Finally, I'd like to express my gratitude to close friends (Vishnu, Shadi, Tuğçe, Rosi, Axel, Simon, Martin, Prachi, Guzmary), co-workers/friends (Oriana, Brian, Thish, Krish, Dan, Claudio, Somaya)[1] and to my awesome family for patiently tolerating my long thesis-related conversations, and for the valuable *espirit de corps* during the research period.

This work is clearly the product of a large graph of contributions. Thank you all!

[1] A special Thanks to Vishnu who, when needed, helped me reach writing speedups of 1000x.

Bertrand Russell, the noted English mathematician and philosopher, once stated that the theory of relativity demanded a change in our imaginative picture of the world. Comparable changes are required in our imaginative picture of the information system world... Copernicus laid the foundation for the science of celestial mechanics more than 400 years ago. It is this science which now makes possible the minimum energy solutions we use in navigating our way to the moon and the other planets. A similar science must be developed which will yield corresponding minimum energy solutions to database access. This subject is doubly interesting because it includes the problems of traversing an existing database, the problems of how to build one in the first place and how to restructure it later to best fit the changing access patterns. *Can you imagine restructuring our solar system to minimize the travel time between the planets?*

Charles Bachman
The programmer as a navigator
Turing Award Lecture: 1973

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Graph databases are a novel class of non-relational database system, tuned to support network-oriented queries and storage. For this they offer graph abstractions as building blocks to their systems (i.e., vertices/nodes and edges), and they have a design that aims to improve common graph processing patterns, such as traversals. In spite of their specific design characteristics, the query optimization strategies in these databases can fail to reach optimal performances, casting doubt about the role of these databases as the best tool for graph processing tasks [WRW+13].

The limitations for query optimizers in graph databases can be explained by some observations: 1) the standards for implementing and querying these databases are still evolving, thus there are few industry-adopted principles to frame performance tuning approaches; 2) the execution time of graph algorithms may depend on hard-to-predict topological characteristics, demanding novel models from query optimizers; 3) the complexity of some graph query languages (in particular traversal-based languages) exceeds by far that of traditional query languages (e.g. Gremlin is actually Turing-complete[AAB+16]), this complexity could arguably make queries harder to optimize automatically; and 4) graph databases can be integrated with other storage systems, acting as a multi-store, in this context there are scarce studies on how these neighboring storage systems could be leveraged for helping query optimization.

As a result from the limitations of query optimizers, developers are burdened with a bigger responsibility to guarantee the performance of their graph database applications. In absence of a community-accepted set of best practices, the application-level optimization of queries in a graph database can be a hard, uncertain task for developers.

Clearly, this state of affairs restricts graph databases from living up to their potential performance. In turn, this can be detrimental to the adoption of the technologies and it could prevent their evolution. On the other hand, any contribution to improve the performance that developers perceive (either by assisting developers with a set of reliable best practices for good performance, or by improving the query optimizer of

the database) can be a specially valuable and impactful contribution, as it could help in shaping a technology that might be in its defining stages.

Rather than addressing the challenges of database-level query optimizations, we propose in this study the instructive approach of focusing on application-level optimizations that an existing graph database might offer to its developers. Specifically, we compare some alternative optimizations, by evaluating experimentally the performance tradeoffs of each in some carefully designed microbenchmarks. In pursuing this study we aim to derive a collection of documented and reliable best practices for developers to write application-level query optimizations for a given graph database. With this work we hope to contribute to the ongoing process of standardization, improving the understanding on the optimization alternatives for general queries in a mainstream graph database.

Summary of Contributions:

We outline the specific contributions of our work in the following way:

1. **Background:** We touch briefly upon the theoretical background on the field, introducing fundamental concepts from network analysis, naming some frameworks that the database community has proposed to support network analysis tasks and existing graph query languages. We also include a summarized presentation on Apache Titan (Chapter 2).

2. **Evaluation Questions:** We define a set of core evaluation questions about the impact of alternative developer choices on the performance of their graph database applications. These questions act as a focal point to structure our experiments (Section 3.1).

3. **TitanLab:** We introduce TitanLab (Section 3.2), our protoype for microbenchmarking application-level graph database query optimizations. TitanLab is based on Titan, Gremlin and the TinerkPop framework.

4. **Microbenchmarks:** With TitanLab we implement and evaluate a set of 7 carefully designed microbenchmarks to examine our core questions. We start by considering global access to items in Titan through IDs vs UIDs (Section 4.3.1) and with different filter queries (Section 4.3.2). Next we examine the performance characteristics of Titan's native indexes in regards to cardinality Section 5.2.1 and the ordering of subqueries Section 5.2.2 when querying with conjunctive predicates. We additionally provide a simple evaluation on the sensitivity of traversals in Titan to variations in topologies (Section 6.2.1). In further microbenchmarks we illustrate experimentally the potentials of some optimization strategies, such as denormalization (Section 6.2.2) and the use of search engines for graph functionality in Titan (Section 7.2).

5. **Summary of Best Practices:** We conclude by summarizing our findings in a set of best practices (Section 8.1), disclosing some possible threats to the validity of our evaluations (Section 8.2), and by proposing future work (Section 8.4).

2. Background

In this chapter we present the necessary theoretical background and context for our study. We organize this presentation as follows:

- **Context for the Development of Graph Technologies** We begin by concisely presenting the context in which graph technologies are being developed (Section 2.1).
- **Network Analysis** We briefly introduce the field of network analysis (Section 2.2), relevant related concepts (Section 2.2.1) and examples of network analysis tasks which have motivated the development of technologies for graph storage and processing (Section 2.2.2).
- **Frameworks Supporting Network Analysis** Discussion on frameworks (Section 2.3) that the database community has proposed to support network analysis tasks. We divide our survey of these systems in three main categories: *systems with relational support for graph processing*, *graph databases* and *graph processing engines*.
- **Graph Query Languages** A brief introduction to the languages that the community has developed to query graph databases (Section 2.4).
- **Query Optimization in Graph Databases** Based on the previous sections where we introduce graph query languages and graph databases, we can present an overview of relevant work in query optimization for graph databases (Section 2.5).
- **Apache Titan** We conclude this chapter with a review of the database that we selected to carry out our evaluation: Apache Titan (Section 2.6).

2.1 Context for the Development of Graph Technologies

Since its early days the computer industry has been continuously devoted to creating tools for the essential tasks of storing, sharing and processing varied data, helping end-users to extract insights and useful knowledge from data sources.

Database systems have played an important role in this history, growing into a major sector of the software industry, and one of the principal corporate software products.

Filling a gap between the OS and the applications, database systems abstract away the complexities of data management from software development. In this position they act as guarantors of data integrity and efficient access to stored data, allowing programmers to focus on building applications.

As a concept, databases came into being in the 1960s, maturing from early file management systems, with the proposals for network[1] and hierarchical models. In the first case (e.g. the Codasyl DBTG standard), records were organized as types and relationships; in the later, records followed tree hierarchies (e.g. IBM's IMS). Although these models gained traction during a decade, several drawbacks limited their mainstream adoption, among them the fact that, in most commercial implementations, databases under these models required programmers to be aware of the physical data arrangement and manage pointers.

Relational databases were proposed as an alternative model by Edgar F. Codd in the 1970s, embodying an approach towards data independence, offering developers the abstraction of tables and a relational set-based algebra. This specific algebra opened the door to the development of high-level declarative data access languages -with SQL being the main representative-. Such languages eased the demands on application developers working with the database, and they offered opportunities for runtime adaptive performance optimizations when mapping them to low-level data processing primitives, making additional use of performance-tuned physical data structures and statistical information. The complete range of these optimizations has been called by some authors "one of the crown jewels of the database systems research community"[HFC+00]. In fact, the combination of a high-level query language with low-level vendor-specific solutions for query optimization led to the creation of competitive solutions that contributed to the mainstream adoption of relational databases during the early 1980s, a trend sustained for decades[2], with relational databases commonly marketed as a one-size-fits-all solution.

Changes in the last decades have brought into question the tenets of relational systems as a one-size-fits-all solution. The foremost of these changes is the datafication of

[1] The network model is, to an extent, a precursor to graph databases, championing concepts such as traversing a data store following links.

[2] For a historical overview of the early database industry, we refer the reader to the work of Martin Campbell-Kelly[CK04]

society, a process broader than simple digitization, consisting of *"taking all aspects of life and turning them into data"*[CMS13]. The scale of datafication is evidenced in our daily production of 2.5 quintillion bytes of data[IBM][3].

Datafication has turned *data science* into a fundamental discipline -encompassing mathematics, statistics, machine learning, computer skills and domain knowledge- for companies to be able to integrate large amounts of rapidly generated data in diverse forms and from diverse sources (social networks, heterogeneous sensors[4], etc.), making valuable data-driven decisions and avoiding information overload.

The emergence of data science brings with it novel workflows for application architects, developers and business analysts, consisting of a variety of tasks, workloads and models for which relational stores are not typically designed. Network analysis, a domain with significant business value, is one of such cases, spuring the creation of different graph technologies.

In the next section we will introduce some concepts and tasks from network analysis. We hope that these tasks can help to illustrate some use cases of graph technologies.

2.2 Network Analysis

2.2.1 Concepts from Graph Theory

A *network* can be defined, following the Merriam-Webster dicitonary [MW04], as *"an interconnected chain, group, or system"*. This abstract concept of a network generalizes everyday systems that we percieve as networks, such as radio networks, computer networks, social networks.

Networks and their interactions permeate every aspect of our lives, from our cells and the ecosphere to planetary systems. In recent years, computational network science has emerged as an active field, bringing forward an abundance of tools for network analysis, helping us gain insights into the interconnected systems that surround us (e.g. the proportion of weak interpersonal ties needed to spread information faster in social groups, or the personalized relevance of websites from within the whole Web).

Graphs are the main model to represent networks for analysis. Following mathematical conventions, a *graph* G can be defined as a set of *vertices* (also called nodes) V and *edges* (also called links) E, such that each edge connects a pair of vertices. More precisely, we can express that each edge has one vertex in V as a *source vertex* and a vertex in V as a *target vertex*. Thus, a graph is defined as $G = (V, E)$.

[3]It is estimated that 90% of the data in today's world has been created within the last two years[IBM]. The IDC projects a sustained increase of 236% per year in our data production rate, leading to an expected volume of more than 16 zettabytes (16 Trillion GB) of useful data by the year 2020[TGRM14].

[4]Researchers at HP Labs estimate that by the year 2030 there will be 1 trillion sensors in the world (roughly 150 sensors per person)[HPL12], continuously producing information.

Another way of presenting a definition for a *graph* could be to consider them simply *dots* (vertices) and *lines* (edges), which can be organized in varied and complex ways ([RN10]). While the resulting definition is the same, by looking at graphs as primitive dots and lines, the particular nature of graphs, as tools for modeling and representing real-world data, is emphasized.

Building on both of these definitions, graphs can be further characterized as *directed* or *undirected*, where a *directed graph* can be described as a graph in which for each edge there is a semantic difference concering which vertex is the source vertex and which one is target (also in directed graphs edges can be conveniently drawn with an arrow-tip at their end). Conversely in an *undirected graph* there need not be any distinction about the origin or end vertexes of the edges.

Another relevant categorization considers the number of parallel edges (i.e., edges that have a same source vertex and a same target vertex) that a graph can have: *simple graphs* allow for no parallel edges, *multigraphs* allow for parallel edges. Furthermore, graphs can allow for *loops* (i.e., edges that start and end in the same vertex) or not; they can present *weights* on edges (i.e., a numerical value given to an edge, which can express several aspects of the relation that the edge models) or they can be unweighted graphs (note, the category is unweighted, not weightless).

Several data structures can represent graphs for analysis purposes. Among them, adjacency matrixes/lists and incidence matrixes/lists.

For storage, a common and useful model used is that of a *property graph*. This model basically extends directed multigraphs with the following considerations:

- Vertexes can be grouped into *vertex types* and edges can be grouped into *edge types*.
- Through the different edge types, a single graph can now model multiple types of relations, thus becoming a *mult-relational graph*. In contrast, a *single-relational* graph contains only a single type of edges, thus it expresses a single type of relation.
- Each vertex and edge has a *label* that defines its type, hence the corresponding graph is called a *labeled graph*.
- With each type, there is the option to define a *schema* (i.e., a blueprint of fields that the entity can contain). In this way, vertices and edges can store attributes or properties, according to their type. Correspondingly, the generated graph can be deemed an *attributed graph*.

As a result of these considerations, a *property graph* can be defined as a directed, attributed, labeled, multigraph, which can moreover express multiple relations ([RN10]). Some authors add to this definition the fact that both edges and vertices can be uniquely addressable by an identifier ([Wie15]).

Due to its characteristics, *property graphs* can be useful to developers because they can easily use such type of graphs to represent other graphs by adding or removing some of the characteristics (e.g. making the graph undirected or making it express a single relation).

Aside from the *property graphs*, there are other models used to store graphs, among them the RDF model. This is a less expressive model than *property graphs*. RDF is standardized by the World Wide Web Consortium (W3C), for representing semantic information about the Web. In this context they present a model in which data is stored as triples.

2.2.2 Examples of Network Analysis Tasks

[AW+10] and [CF12] are some of the most important textbooks for introducing network analysis tasks. Among those tasks, they present the following.

- **Graph metrics:** Calculation of graph metrics, such as diameters, centrality of vertices, among others.
- **Global algorithms:** Community detection, influence propagation, page-rank.
- **Path-based functionality:** Shortest-paths, etc.

2.3 Frameworks Supporting Network Analysis

Among the frameworks supporting network analysis we can outline at least three types: those based on relational database, those proposing a dedicated graph database, and those that don't concern themselves with graph storage but instead focus on large-scale processing.

2.3.1 Relational Support for Graph Processing

Numerous authors have studied how to use a relational model for network analysis tasks. Authors have considered approaches that use DSLs to hide away the complexity of graph queries from developers. Some example cases of studies covering the use of relational systems for graph processing include: [JRW+14] and [WRW+13].

2.3.2 Graph Databases

[AG08] offer a comprehensive survey where the models used for graph databases are discussed at length. The authors start by comparing graph databases to other similar systems, such as XML representations, object-oriented databases, semi-structured, among others.

Graph databases constitute a modern class of non-relational database systems proposed by the database community. It is tailored to support the network-oriented workloads of

network analysis. With this goal they offer a data model based on graph abstractions (i.e. nodes/vertices and edges), and can adopt different optimizations to speed up essential graph processing patterns, such as traversals.

Some examples include, DEX ([MBGVEC11]), Neo4j ([Web12]), OrientDB ([Tes13]) and Apache Titan[5].

It is to be noted that OrientDB can also be used as a document store, a key/value store and general object store.

2.3.3 Graph Processing Engines

Graph processing engines are not intended for storage, but for distributing large-scale processing over graph structures, following a directed-acyclic graph assignation of tasks. Pregel ([MAB+10]) is a common predecessor for most systems. In Pregel, authors propose a vertex-centric approach to graph processing. Thus, every compute instance can represent a vertex in a large distributed graph.

Other examples of graph processing engines include [XGFS13], [SWL13] and [MMI+13].

2.4 Graph Query Languages

There's still no consensus on the most efficient way to query graph-structured data. To date a vast amount of solutions have been proposed. [AAB+16] presents a survey on the characterstics of the main representatives (i.e., SPARQL for semantic queries overt triple data (RDF), Cypher for pattern-matching over general graph data and Gremlin for navigation-based queries).

Given that we have selected Gremlin for our study, we will briefly discuss its main characteristics in more detail than those of its alternatives. Gremlin is a iterator based language, offering a syntax similar to that of functional programming languages, and of modern system such as Spark. In Gremlin every statement is formed as a sequence of steps. Gremlin is a language that relies on the concept of lazy evaluation, thus it postpones the evaluation of the queries expressed until the invocation of a concrete step. We refer readers to the documentation of this language [6].

In order to be used by graph processing systems Gremlin is paired with the Apache TinkerPop framework. This framework offers graph processing abstractions, and a library for evaluation of functional components. Accordingly, TinkerPop constitutes a bridge between high-level queries written in Gremlin and low-level domain-specific implementations. Among the domain implementations, TinkerPop can connect to Neo4j, OrientDB, Titan and distributed processing frameworks like Hadoop.

[5] http://titan.thinkaurelius.com/
[6] http://tinkerpop.apache.org/docs/current/reference/

2.5 Query Optimization in Graph Databases

In graph databases the possible optimizations are closely related to the query language itself. In this sense, navigation-based languages could enable a different framework for optimization than pattern-based ones. [Gub15] reviews some optimization methods, such as operator ordering and cardinality estimation, in the context of pattern-matching based graph query languages.

Although scarcely researched, the commonality of the TinkerPop could help in the standardizaton of graph query optimizations for different graph databases. We hope that our work will contribute to this task, even though we focus on a very specific database product.

Authors have also researched on database implementation considerations to improve graph processing tasks [Pin16].

2.6 Apache Titan

Apache Titan is a graph database that presents a property graph model. It relies on non-native storage (i.e., supporting systems like Apache Cassandra, HBase and BerkeleyDB). Titan distributes vertices in key-column value stores, by assigning a row to each vertex with vertex properties and edges being saved as different column values of that row. We refer readers to Titan's data model[7].

Titan offers Gremlin as its native query language.

In addition Titan integrates with search engines such as Elasticsearch and Solr, through different add-ons that propagate all the changes in the core storage towards the search engine storage. Search engines enable full-text querying, geo querying and numeric range queries. In our study we will evaluate this integration.

2.7 Summary

In this background we introduced fundamental concepts on property graphs, as they pertain to our research on an existing graph database that offers the property graph model. Next we briefly listed some network analysis tasks, overviewing (from a great height) some systems that the database community has proposed to support network analysis tasks. We roughly outlined some different graph query languges, and we refered the reader to work on query optimization, and we concluded by pointing the reader to information about the database that we used in our study. A large amount of meaningful and impactful work was left out of our discussion in this chapter. We strongly encourage the reader to complement our discussion with the cited research.

In the next section we embark on the core of our study, presenting TitanLab, a prototype for microbenchmarking application-level query optimizations for graph databases and the essential evaluation questions which motivate our study.

[7] http://s3.thinkaurelius.com/docs/titan/current/data-model.html

3. Microbenchmarking Application-Level Query Optimizations in Titan

In this chapter we establish the core evaluation questions that we tackle in our study. Subsequently, we describe the key components that support our experimental evaluations: the prototype for microbenchmarking, the testing environment, the datasets and the measurement methodology.

For our examination of application-level query optimizations in Titan, we consider particularly the usage of alternative indexes (Chapter 4), the characteristics of composite indexes (Chapter 5), local traversals (Section 6.2.1), denormalization of properties to speed up filter steps in traversals (Section 6.2.2) and the utilization of full-text search engines to support graph queries (Chapter 7). The evaluations and the specifics of the microbenchmarks are the subject of the next chapters.

The outline for this chapter is as follows:

- **Evaluation Questions**. We begin by providing the complete set of evaluation questions that we aim to address in our study (Section 3.1).
- **TitanLab**. We present TitanLab, a prototype for microbenchmarking query optimization opportunities provided to end users by Titan (Section 3.2).
- **Evaluation Environment**. A quick listing of the defining characteristics from the execution environment of our tests (Section 3.3).
- **Datasets**. Description of the datasets used (Section 3.4).
- **Measurement Process**. We conclude the chapter with an account of our measurement methodology (Section 3.5).

3.1 Evaluation Questions

Good developers and engineers need to know well their toolbox, so they can select the right tool for the task at hand. In this study we want to contribute to this needed knowledge by evaluating some different tools available for the development of graph database applications with Titan.

Graph databases, specially those relying on the Gremlin language and the TinkerPop framework, differ from traditional relational databases in how queries are optimized. Whereas relational databases have developed an infrastructure that enables the database system itself to optimize query plans and other related aspects during run time, in graph databases the query languages and processing engines are still evolving, thus, developers have a bigger responsibility in query performance optimization. In fact, developers must decide exactly how to use the graph database so as to achieve the best possible performance for their queries. In this context knowing the right tool to is an imperative matter.

Concretely, we propose to answer the following set of research questions, covering some of Titan's basic application-level query optimization alternatives:

1. At a basic level, considering *access by ID*, should a developer keep track of the specific Titan ID of a frequently accessed item, or would it be the same to access it by an application-level unique identifier supported by a native Titan index? (Chapter 4)

2. What benefits can be expected of using an index to speed up *global filter queries* over not using any? (Chapter 4)

3. Is there a best indexing solution, among Titan's alternatives, to support single-predicate *global filter queries* in the context of ad-hoc exploratory graph analysis? (Chapter 4)

4. Practicioners report that extreme cardinality data might be detrimental to the performance of common indexing structures in mainstream databases, such as B-trees[1], and Cassandra's secondary indexes[2]. Considering that Cassandra-based composite indexes are the basic indexing approach of Titan when using Cassandra, it would be important to determine: How do composite indexes perform under extreme cardinality (very low or very high) cases?(Chapter 5)

[1]This has been studied by IBM's DB2 community: https://www.ibm.com/developerworks/data/library/techarticle/dm-1309cardinal/

[2]As an example, [Mis14] specifically claims that: *"(an index over) a column with high cardinality may result in performance overhead under high data load and large Cassandra clusters."* Similar observations are given by DataStax, the main vendors of Cassandra's enterprise versions: https://docs.datastax.com/en/cql/3.1/cql/ddl/ddl_when_use_index_c.html

5. Queries with conjunctive predicates can be answered either by an index over all query fields (which requires database developers to create the indexes with the conjunctive predicate query in mind), or by combining different indexes (which allows for more generic indexing). As a developer supporting ad-hoc queries, it would be valuable to know: How does Titan currently handle multi-index conjunctive predicate queries with its native indexes, and is there any performance trade-off that should be kept in mind regarding this? (Chapter 5)

6. When traversing over a given number of nodes, how sensitive is the performance of basic traversals to variations in the topology? (Chapter 6)

7. What are the potential gains from the strategy of denormalizing vertex data (by duplicating it as properties of incident edges) to speed up traversals? (Chapter 6)

8. What amount of performance improvement can be achieved by leveraging search engine functionality in traditional graph algorithms and what are the tradeoffs involved? (Chapter 7)

These questions are far from exhaustive in their exploration of the tools available for graph application developers when using Apache Titan. We selected them because they cover essential everyday functionality (global access, traversals) and some potentially valuable optimization opportunities (denormalization, leveraging search engines) that Titan offers.

3.2 TitanLab

To support the evaluation of competing application-level query optimizations, such as denormalization or using the graph as an index, and to address our core evaluation questions (Section 3.1) we implemented TitanLab.

TitanLab handles the execution of different microbenchmarks representing diverse aspects of an Apache Titan application. For this TitanLab implements the creation of a Titan schema, the loading of data and other required functionality with Titan embeded into the application. Figure 3.1 provides an architectural overview of TitanLab.

TitanLab uses Apache Titan 1.1.0 as a graph database. Titan in turn supports graph queries through the TinkerPop[3] graph processing framework. Specifically we use TinkerPop 3.1.1. As a storage backend Titan currently supports Cassandra, HBase and BerkeleyDB. We selected Cassandra 2.1.11 as a storage backend for TitanLab. In addition Titan supports full-text indexing and queries with mainstream search engines such as Solr, Lucene and Elasticsearch. In TitanLab we use Elasticsearch 1.5.2 as a full-text indexing backend.

[3]http://tinkerpop.apache.org/

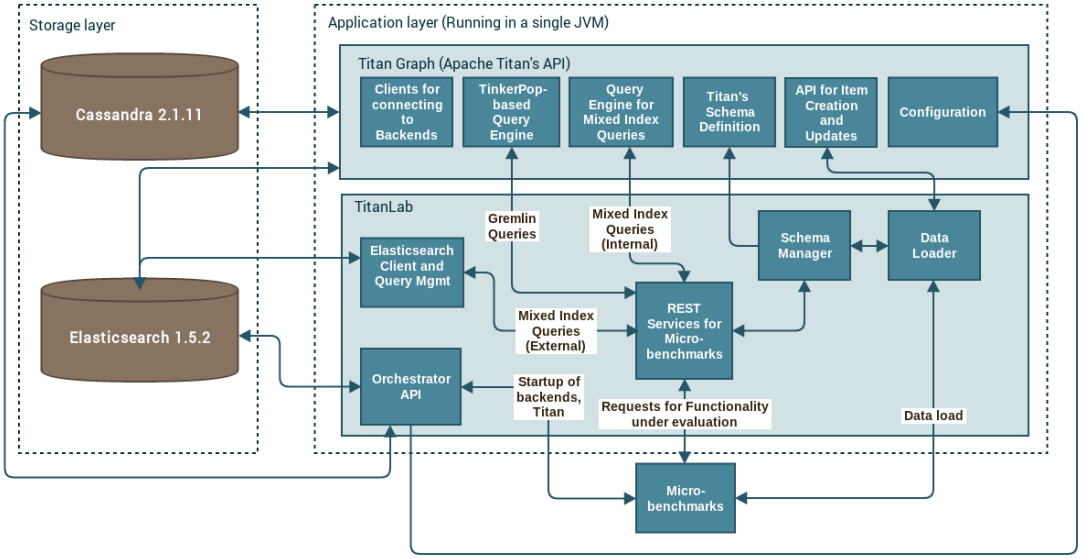


Figure 3.1: Architectural overview of TitanLab

To help understand the different Titan components, we propose a high-level structure for them in Figure 3.1. The "TitanGraph" section of the figure can be explained as follows:Titan offers developers the abstraction of a "TitanGraph" as an embedded API for the graph functionality. This API presents in the first place the option to configure how Titan connects and uses the backends. The connections themselves are hidden from the API users. Next, TitanGraph offers components for defining the database schema (including indexes, properties, labels, etc.) and for creating items using transactional scopes. Finally, for the purposes of our presentation, TitanGraph exposes 2 alternatives for querying graph data, one is the native API, utilizing Gremlin and the TinkerPop framework. This API supports all graph functionality from Gremlin. The second alternative is an API for querying full-text indexes. We refer to the queries that are addressed to this API as mixed internal queries. These type of queries will be presented better in the upcoming chapters.

Concretely, TitanLab offers the following functionality, based on its use of TitanGraph:

- **Configurable Service Orchestration:** An orchestrator application to start, delete, and gracefully terminate Cassandra, Elasticsearch and our Titan-based application according to user-provided server configurations.

- **Schema Definition:** The definition of the Titan schema on data-load time, as configurable options, either by environmental variables or by given configuration files. Specifically, this concerns properties, property keys, cardinality of properties, labels and indexes. We approach indexing in TitanLab by using both of Titan's index types (composite and mixed) as follows:

  - For composite indexes (i.e., Titan's native indexes, supported by Cassandra), we create an index over each single property of vertices or edges, according to the needs of the evaluation. This is clearly stated for each corresponding test.
  - For mixed indexes (i.e., full-text indexes, supported by Elasticsearch), we adopt the approach of creating a single index for vertices and a single index for edges, over any selection of properties. This maps each Titan entity type to a single Elasticsearch document type under a common Elasticsearch index.

- **RESTful Functionality for Microbenchmarks:** TitanLab offers a central Titan-based application offering (as RESTful services) the functionality tested in our microbenchmarks. The different microbenchmarks ared defined in the following chapters.
- **Data Load and Generation:** A dedicated RESTful service for data loading, with the option to add properties and randomized values with a given selectivity.
- **Stand-alone Elasticsearch Queries:** A stand-alone Elasticsearch project for querying the full-text backend in a manner independent from Titan, using a fixed pool of Elasticsearch clients to reduce connection overheads.
- **Documentation:** Documentation for using and extending this prototype.
- **Microbenchmark Applications:** A set of microbenchmark applications that make requests to selected functionality, with a given number of repetitions, recording the time measurements returned with the responses.

TitanLab is based on a locally built Titan 1.1.0 artifact, cloned from the public Titan repository. The code for the artifact is included with TitanLab, to help in evaluating the effect of changes to Titan's codebase on TitanLab performance tests.

Given its characteristics, a solution like TitanLab can be employed as a tool for Titan developers to assess the impact of new functionality on the performance of end user optimizations. To the best of our knowledge, Apache Titan's native performance tests do not consider comprehensively this kind of optimizations; and there is no tool with the characteristics of TitanLab in the Apache Titan repository.

Before concluding this section we would like to mention a design decision in TitanLab concerning the external queries to Elasticsearch: The Java API of Elasticsearch offers several types of queries, with different performances. For generality we decided to utilize a SimpleQueryStringQuery request in all cases, save for the degree centrality microbenchmark, where we use a more fitting TermQuery. We consider that the decision to use a SimpleQueryStringQuery corresponds to a naive usage of Elasticsearch. More specialized Elasticsearch queries could indeed lead to better performance for the mixed external case in our tests. However, contrasting Elasticsearch query solutions was beyond the scope of this study.

3.3 Evaluation Environment

Our experiments were conducted on a commodity multi-core machine, running Ubuntu 14.04 and Java SDK 8u111-linux-x64, with an Intel® Core™ i7-2760QM CPU @ 2.40 GHz processor (8 cores in total) and 7.7 GiB of memory.

Our decison to use commodity hardware stems from the observation that given the limited hardware requirements of Titan/Cassandra/Elasticsearch, small business companies are likely to use these products on commodity hardware or cloud-based offerings. Given that benchmarking cloud applications involves a more complex methodology, which perhaps departed from our concerns, we decided to focus on commodity hardware. Researchers have also adopted the practice of testing graph database applications using commodity hardware [BPK15].

3.3.1 Backend Configuration

Regarding the configuration of the different storage solutions, we decided the following:

- Titan's configurations were left with their default values. This included late materialization (i.e., query.fast-property = false). Readers are referred to Titan's documentation[4] to be informed on the defaults. We ran TitanLab with a maximum heap of 1GB. This was only changed for the test involving the *Pokec* dataset, where it was raised to 2GBs, in consideration of the extra-needs of the test.
- Cassandra ran with the default properties for the 2.1.11 version[5]. Among these properties, Cassandra defines a maximum heap usage of 1/4 of the available memory.
- Elasticsearch was configured with the default options[6], save for the enablement of dynamic scripting (i.e., script.disable-dynamic= false), which Titan needs to process batch updates. This setting should not affect the validity of our results. We assigned a maximum heap of 1GB. This was changed for the test involving he *Pokec* dataset, where it was raised to 2Gbs, in consideration of the extra-needs of the test.

3.4 Datasets

As our *basic dataset* we adopt a synthetic multi-graph composed of 3.1 million nodes and 2 million edges, divided in 2 relations. This graph was generated by the IBM Netcool Operations Insight team, as a representation of observed network monitoring data. It presents 2 node properties and 1-2 edge properties. For reference, the degree distribution of the different relations in the graph is presented in Chapter 7.

[4]http://s3.thinkaurelius.com/docs/titan/1.0.0/titan-config-ref.html

[5]https://docs.datastax.com/en/cassandra/2.1/cassandra/configuration/configCassandra_yaml_r.html

[6]https://www.elastic.co/guide/en/elasticsearch/reference/1.5/setup-configuration.html

Our selection of this dataset stems from two evaluation design aims. First, the interest of having real textual properties of varying size, representing better real-world use cases. Second, the goal of this project to present experimental evaluations that could directly inform the development of TITAN applications in the domain of Communication Service provisioning.

For global access (Chapter 4) and composite index tests (Chapter 5) we loaded the graph with an addition of 20 dummy textual and integer properties with different indexing and selectivity characteristics. For the selectivity, we assigned the values randomly. Given that we experienced several OOM errors while running these tests over the complete dataset (in particular when using the graph as an index option), we limited our observations for these tests to a subset of 450k nodes, which gave us a stable behavior. The same dataset was used for our evaluation on the potentials of denormalization of properties to speed up filter steps (Section 6.2.2). We'll refer to this dataset as the *selectivity dataset*.

For local traversals (Section 6.2.1) we loaded the complete dataset with an addition of 3 small subgraphs, each formed with 509 new edges, capturing different local topologies for the traversals. Fittingly, we call this the *traversal dataset*.

In consideration of the specificity of the aforementioned tests, noting that the synthetic basic dataset only played a role as an extra load for the tests, we assess that there is little threat to the validity of our results related to our choice of dataset.

For our evaluation on the potential of full-text integration to provide performance gains (Chapter 7), we used the basic dataset. Given that the functionality under test (the calculation of the average degree centrality of vertices) is broader than the functionality of other tests, we complemented our results by additionally using 2 public datasets, the *Openflights* [Ops11] and *Pokec* [TZ12] datasets, from the Tore Opsahl and the Stanford SNAP repositories.

Openflights portrays routes between US airports. It consists of 8056 nodes and 30500 edges, thus it represents a very small dataset. *Pokec* collects friendship connections in a Slovakian social network, with 1632803 nodes and 30622564 edges, thus it represents a very large dataset. Both are directed and unweighted graphs.

We selected *Openflights* and *Pokec* based on the amount of data per node. The first represents a topology with few node properties (i.e., 10 fields, 6 textual and 3 numerical) and no edge properties. The latter is the opposite case, with nodes that can have up to 59 properties of various data types. Numerous edge properties would've been another interesting feature from a dataset, but it was not considered in our evaluation.

A summary of the datasets is provided in Table 3.1.

3.5 Measurement Process

In our evaluation we focus on the response time of the core retrieval functionality for each request (i.e., we do not consider rendering times). The measurements were made by instrumenting the central application code with lightweight Java System timers (i.e., `System.nanoTime()`), and returning the measurements next to the REST responses.

We selected to use the Java System timers because of their simplicity and their sufficient precision for the time ranges of our experiments. Although these timers have been shown by practitioners to be adequate for differential time measurements, with accurracy and precision commonly in the microsecond range[7], we are aware of some of their limitations in contrast to more developed benchmarking solutions (i.e., their measurements might still be succeptible to errors due to dynamic optimization, resource reclamation and others). In future work we might consider double-checking our results by using more advanced benchmarking solutions.

In our evaluations measuring response time we carried out one type of test at a time, without concurrent requests. For each test we ran a sequence of "warm-up" queries (to rule out cold cache effects) before launching the recorded tests. We increased the number of repetitions from 5, 10, 100, 1000, according to the duration of the test and the value of a paired two-tailed t-test. This test was used to making sure that when reporting an important difference in averages (1.5x speedups or more), it was done with observations that had a probability lower than 0.05 of belonging to the same population (i.e., the value of the t-test over both ranges of observations had to be lower than 0.05).

Next to the response time, we considered the metrics of Gremlin steps, by using the Profile functionality. Given the monitoring overhead we used these measures solely to gauge the typical distribution of time between the Gremlin steps, aiming to understand better the query execution.

For insights into the factors affecting the response time for each test, we complemented our studies with system performance metrics. In consideration of the monitoring overhead, these measurements were made in separate runs of each individual test. The following tools were used to measure system performance:

- First, we monitored TitanLab with JConsole, noting its specific individual heap usage and other essential runtime characteristics. The observations on the heap usage of TitanLab are important because they capture the runtime memory footprint of the embedded Titan, its cached data and transaction intermediate results, all of which live in the JVM heap allocated to the application.
- Titan, Cassandra and Elasticsearch are able to provide numerous metrics about their performance. Titan monitors offered us information about aspects such as the uses of iterators, the histograms built on backend slices and the transaction times. Cassandra provided us with a good amount of details on its

[7] https://www.ibm.com/developerworks/library/j-benchmark1/

overall state (e.g., thread counts, key caches), plus specific information on the column families and keyspaces that it managed. Most importantly, Cassandra monitors informed us on runtime characteristics of the Titan keyspace, such as the MemTable heap sizes, estimated row counts and row cache hits. From Elasticsearch we focused on metrics for query latency and the field data cache. All these metrics were collected with Graphite, using different off-the-shelf monitoring agents from Maven.

- As an add-on to the Graphite monitoring, we used a Graphite-Carbon solution to include system characteristics (CPU, memory usage, cache queues, etc.).
- JConsole and Graphite-web/Graphite Browser were used to visualize the metrics.

While the response times are included in our discussion for all tests, the measurements from system performance are only included when they help to understand better our results (i.e., in Section 4.3.1.2, and Section 4.3.2.2). Otherwise, for conciseness, they are ommited.

3.6 Summary

In this chapter we presented the key features from the implementation of Titan-Lab, a prototype for microbenchmarking application-level optimizations when using the Apache Titan graph database with Elasticsearch as an additional search engine backend. We provided the key characteristics of our evaluation environment, the datasets selected and the measurement methodology in our evaluations. At last, we presented the concrete evaluation questions that we aim to address in this study.

In the next chapters we present our evaluations dealing with the usage of alternative indexes (Chapter 4), composite indexes (Chapter 5), local traversals (Section 6.2.1), denormalization (Section 6.2.2) and support for graph queries with integrated search engines (Chapter 7).

Dataset	Nodes	Edges	Origin	Description
Basic	3.1M	2M	Private	• Synthetic network infrastructure. • 2-relations multi-graph. • 2 textual node properties, 1-2 edge properties. • Unweighted, directed.
Selectivity	450k	450k	Private	• Synthetic network infrastructure graph. • A subset of 450k nodes from the basic dataset, with a new relation for the graph as an index option. • 22 textual node properties, 12 edge properties. • Unweighted, directed.
Traversal	3.1M	2M	Private	• Synthetic network infrastructure. • Extension of the basic dataset with added 509x3 edges forming specific local topologies. • 3-relations multi-graph. • 22 textual node properties, 11-12 edge properties. • Unweighted, directed.
Openflights	8056	30.5k	[Ops11]	• Transport routes between US airports. • 10 node properties, no edge properties. • Unweighted, directed.
Pokec	1.6M	30M	[TZ12]	• Online social network from Slovakia. • 59 node properties, no edge properties. • Unweighted, directed.

Table 3.1: Summary of datasets used in our evaluations

4. Case Study: Global Access

In this chapter we present our evaluation on the performance impact of selecting different global access primitives. We start by looking at the basic trade-off between retrieving vertices by their ID or through a unique identifier supported by an index. Next we propose a targeted microbenchmark that studies exclusively the retrieval of vertices given a simple equality predicate on a property. In this study we compare 5 access options: a) full-table scan, b) composite indexes, c) the option of using the graph as an index, d) full-text indexes accessed via Gremlin, and e) full-text indexes accessed via a Java API. To provide a broader understanding on the access alternatives for the task, we considerer variations of our microbenchmark, such as retrieving with pagination, counting, sorting.

This chapter is outlined in the following way:

- **Global Access Primitives in Titan**. A presentation of the access primitives that Titan offers to developers (Section 4.1).
- **Evaluation Questions**. The precise questions that we aim to answer with the evaluation described in this chapter (Section 4.2).
- **Microbenchmark**. Design and implementation of the microbenchmark for our evaluation (Section 4.3) , with a run-down of the results from our tests, and discussion (Section 4.3.1, Section 4.3.2).
- **Best Practices**. A bullet-point summary on the best practices that can be inferred from our evaluation (Section 4.4).

4.1 Global Access Primitives in Titan

4.1.1 Access Path Types

As a developer working with Titan, the global access to graph elements can be accomplished in three basic ways: a full-table query, an access by ID and a global filter query. In Titan they can be used as follows:

- **Full-table query**:The general way would be to access all the vertices or edges of a given graph, using a full-table scan. In Gremlin syntax this would be: `g.V()` or `g.E()`.
- **Access by ID**: The most selective way would be to access elements given their IDs: `g.V(id1, id2...)` or `g.E(id1, id2...)`. In this case, for vertices, Titan uses the basic ID access of the backends, with a lightweight mapping from the Titan ID to the backend ID. For edges, given that they are stored locally with the vertices, Titan needs to access one of the adjacent vertices (i.e., the one with the less number of edges), and then Titan does a full-scan on the edges searching for a matching ID.
- **Global filter query**: Another option could be to select elements according to the values of some distinguishing properties or labels: `g.V().has("propertyA", "valueA")` or `g.E().hasLabel("labelA")`. Within the context of our study, we will refer to this as a *global filter query*.
  - **Full-table scan**: When there is no index covering at least one of the selected properties, Titan answers global filter queries via a full-table scan.
  - **Composite index usage**: When a pre-defined composite index (i.e., Titan's native indexes, backed by Cassandra in TitanLab) covers at least one of the selected properties, then this query becomes automatically answered by such an index.
  - **Mixed index usage**: When Titan is backed by a search engine, there is the alternative of using a mixed index (i.e. full-text indexes, backed by Elasticsearch in TitanLab) to answer this query.
    - *Internal usage*: Queries to mixed indexes can be written using the Titan functionality and internal clients. In the syntax of Titan's Java API this would be: `g.indexQuery("docTypeForVertices", "v.propertyA:valueA").vertices()` or, for edge properties `g.indexQuery("docTypeForEdges", "e.propertyA:valueA").edges()`.
    - *External usage*: Queries to mixed indexes can also be sent directly to the search engine backend, using the functionality and clients of the search engine API.

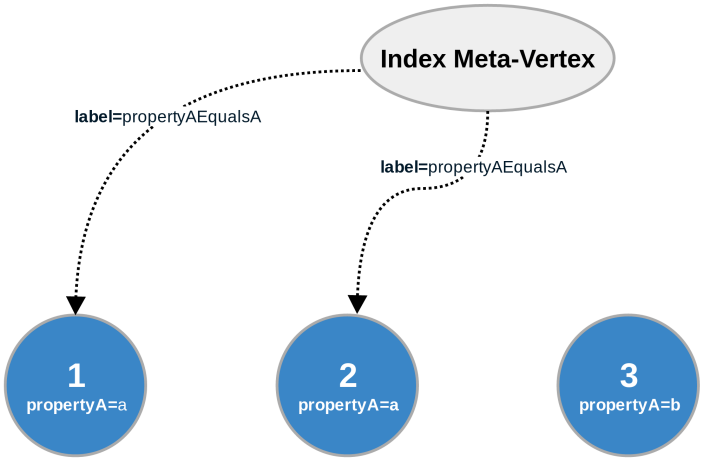


Figure 4.1: An example of using the graph as an index

– **Graph as an index**: Finally, any given vertex can be used as an indexing entrypoint, by having it directly connected to the vertices that the query should return. To accomplish this, data load and property updates should be reflected in the connections of such indexing vertices. An example is given in Figure 4.1, with the *index meta-vertex* pointing to vertices 1 and 2, where propertyA=a. A Gremlin query for this case could look as follows: `g.V(indexVertexId).outE().inV()`. Indexing edges is also possible in this approach but, in the absence of edges able to point to edges, this would require a workaround that introduces yet another level of indirection.

4.1.2 Disclaimer on the Usability of Mixed Indexes

Unlike the other approaches discussed for the *global filter query*, the *external usage of mixed indexes* has a limited access to the items in the context of Titan. Namely, the indexed information of items can be efficiently retrieved, but subsequent traversal steps from the items are not immediately possible. Subsequent traversal steps can only be accomplished by accessing the item once again through its retrieved ID.

This situation also happens for *internal use of mixed indexes*. Such a limitation is the product of the current implementation of Titan, where mixed indexes are not included in the same query optimization framework as other indexes. We sketch this distinction in Figure 3.1.

4.2 Evaluation Questions

Considering the access primitives listed in Section 4.1, we can now define the questions that motivate the evaluation that is the subject of this chapter:

1. At a basic level, considering *access by ID*, should a developer keep track of the specific Titan ID of a frequently accessed item, or would it be the same to access it by an application-level unique identifier supported by a native Titan index (i.e. a global filter query mediated by a composite index)?

2. What benefits can be expected of using an index to speed up *global filter queries* over not using any?

3. Is there a best indexing solution, among Titan's alternatives, to support single-predicate *global filter queries* in the context of ad-hoc exploratory graph analysis?

These are the first 3 questions in Section 3.1.

4.3 Microbenchmark

We designed 2 different groups of tests to answer the evaluation questions. The first set is for *IDs vs. Unique Identifiers*, the second for the *Comparison of Access Primitives for Global Filter Queries*. The tests are presented next.

4.3.1 IDs vs. Unique Identifiers

Addressing our first question, we designed 2 small tests to compare the retrieval by ID vs. by unique identifiers (UIDs). The core difference between the two access methods is that unique identifiers are supported by composite indexes, and Titan might not be aware of the uniqueness of the element. On the other hand Titan IDs map one-to-one to the IDs of elements in the backends, thus Titan can rely on the database indexing of elements, and can add optimizations related to this.

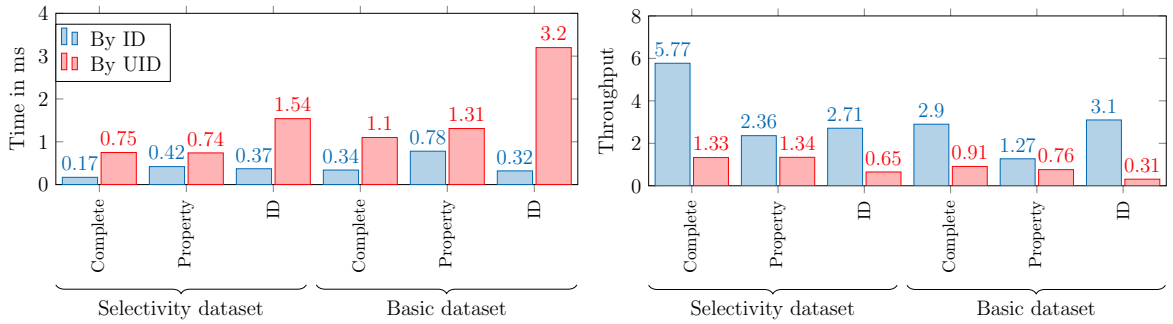
Given that Titan is configured for late materialization, in our tests we evaluated the response time of a) retrieving the complete vertex, b) retrieving only a given property of a vertex, c) retrieving only the ID of the matching vertex. For these tests we used the *basic dataset* and *selectivity dataset*, as defined in Section 3.4.

Considering that retrieving an edge by ID in Titan is a $\mathcal{O}(\log k)$ operation, with k being the number of incident edges on the adjacent vertices[1], and that evaluating this properly would've required additional considerations regarding topologies and local indexes, we decided to not include edge retrieval in our test.

Specifically, we ran the following tests:

1. The retrieval of the same vertex by ID or UID, following a number of 2000 repetitions.

2. The retrieval of random vertices by ID or UID, carried out sequentially per approach.



((a)) Response time ((b)) Throughput

Figure 4.2: Response time and throughput of accessing a vertex by its Titan ID or a unique identifier

4.3.1.1 Response Time and Throughput

Figure 4.2 shows the results of our first test, Figure 4.3 displays the recorded speedups from accessing an item via its ID rather than via its unique identifier.

Our initial observation is that there is generally no visible difference between the response times of the *basic* and the *selectivity* datasets, despite the significant divergence in the number of items (450k vs. 3.01M). This suggests that these individual retrieval operations scale successfully in Titan.

For both datasets, retrieving a complete vertex by ID was slightly faster when explicitly projecting a property or an ID. Roughly, selecting the whole vertex under these conditions, instead of applying a projection, could give a speedup of approximately 2x for the ID case. For the UID the performance difference was closer, with speedups of 0.98 and 1.19x.

We expected that in both approaches projecting an ID would be visibly more efficient than projecting a property, however the small amount of data accessed and proper cache usage seem to hide this.

All our results show a consistently favorable behavior of retrieving by the ID instead of the UID, with speedups from 3-4x when retrieving the complete vertex, and from 1.5-10x when projecting over the ID or the properties.

For this scenario, where the resulting vertex is cached, the performance differences between retrieving this single item by ID vs. by a unique identifier, do not seem to reach an order of magnitude.

The results of our second test are collected in Figure 4.4 and Figure 4.5. This test considers random access to vertices. Unlike the first test, here we evaluate the sensitivity to cache misses of the single-vertex retrieval alternatives.

[1]For more information on this, see Section 14.1.3 of the Titan documentation: http://s3.thinkaurelius.com/docs/titan/1.0.0/limitations.html

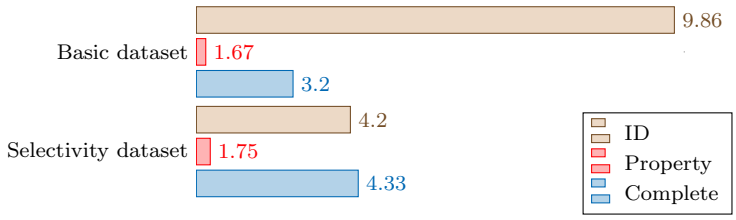
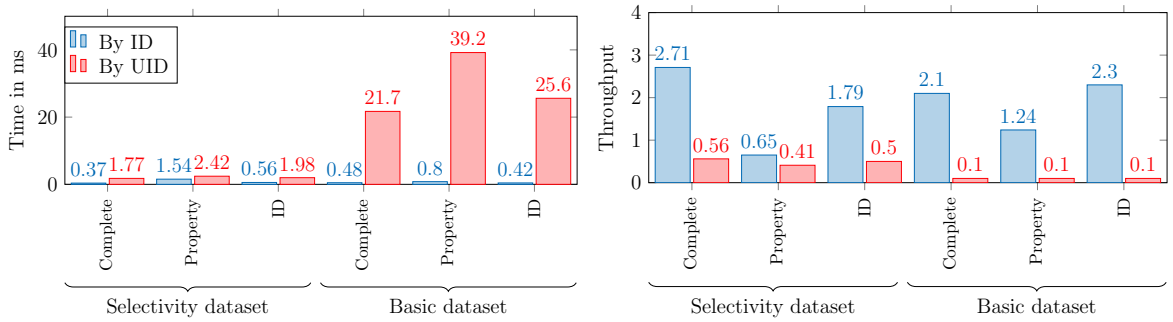


Figure 4.3: Speedups from retrieving a vertex by its Titan ID rather than a unique identifier



((a)) Response time ((b)) Throughput

Figure 4.4: Response time and throughput of accessing random vertices by their Titan IDs or unique identifiers

Our results show that the response time for UID access deteriorates significantly when compared to the first test. While this cannot be quite perceived in the *selectivity dataset*, it is markedly evident in the *basic dataset*. Here we see a performance deterioration that is roughly 40 times slower than access by ID. From these observations we can infer that access to native property indexes (i.e., to the UID composite index) does not scale as well as the access to the native Titan ID index.

An explanation for this can be found by studying Titan's codebase. There, we see that IDs are the natural way to query for specifc items: ID queries are supported by a special LRU cache for items. If an item is not found in this cache, a slice query for edges is sent to the backend with the identifier, being further tuned thanks to the expectation of retrieving a single item. On the other hand, querying for items by a property stored in a composite index translates into a GraphCentricQuery, which in turn implies the selection of an index in the first place. Once selected, queries to a given composite index are improved with dedicated caches (different from the LRU caches mentioned for items) and to conclude, if there is no cache hit, the query is sent to the backend. In this case, the backend query cannot be improved with the knowledge that the query will return a single item.

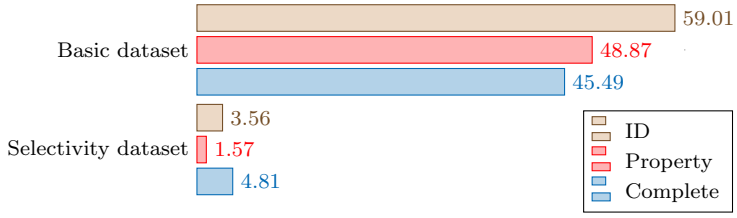


Figure 4.5: Speedups from retrieving random vertices by their Titan IDs rather than unique identifiers

In the second test, whole vertex retrieval continues to be more efficient than projected retrieval. No clear difference appears between the response time of alternative projections (ID. vs. property).

As a result from our tests, we conclude that for the scenario of retrieving a single item, the selection of using the ID over a unique identifier represents a tuning alternative with limited impact (we observed speedups of 1.6-10x). Nevertheless, it might still be a pertinent solution for mission-critical real-time systems.

Conversely, for the broader case, where random items are frequently accessed, and the dataset is large, we can make a strong argument for using direct ID access rather than UID access. In our tests, this application design alternative lead to performance gains from 45 to 59x.

4.3.1.2 System Performance Observations

For the task of retrieving a complete vertex by its ID, the Gremlin profile step shows that the query spends all its time in a single graph access step. When the retrieval happens with the unique identifier, the profile takes 34% of its time in an optimization step (selecting the composite index to use) and a remaining 11% in a backend query. The situation is exactly the same for retrieving only IDs, except that the distributions for the unique identifier approach are: 78% in the optimization step and 17.4% in the backend-query.

The projection over a property changes the profile of the query. For the ID-based method, the regular graph step is followed by 2 steps, a costly TitanPropertiesStep (taking 50% of the query time), in charge of the projection, and a PropertyValueStep (taking roughly 10% of the time). In the UID-based method, more time is spent in the original TitanGraphStep (approximately 30%), with the TitanPropertiesStep taking comparably less time (14%) and the PropertyValueStep taking 34%.

The query profiles matched in both tests, but the time percentages spent for the substeps could not be properly measured for the random test. tes All in all, these profiles show the expected patterns of more complex queries when projecting over a property, and small optimization overheads when related to composite index usage.

The heap usage of TitanLab during both tests was observed with JConsole, with the application itself having a footprint of 120-150 Mbs during the tests for both datasets.

Cassandra monitors provided little information for the *selectivity dataset*. On a run of 500 repetitions of the *basic dataset* Cassandra monitors reported interesting contrasts between the first and second tests. These factors show the effect of random access on the overall system:

- Read latency increased for the different keyspaces that Titan uses in Cassandra to store graph data. For the edgestore keyspace the mean read latency went from less than 10 to more than 25 ms. For the graph index the rise was from 20k to 40k ms.
- Hit rate fell for different keyspaces. In the edgestore keyspace this decreased from 0.57 to 0.52. In the graph index the drop was from 0.54 to 0.46. In the KeyCache of Cassandra the hit rate decreased from 0.53 to 0.49.
- KeyCache entries went from almost 0 to 8k.
- The value of Carbon cache queries went from almost 0 in the first test, to a peak of 625 in the second test.

4.3.2 Global Filter Queries

Global filter queries are the graph database equivalent to the SELECT FROM WHERE clauses in SQL. Similarly, they cover an essential everyday functionality for interacting with the database.

The two last evaluation questions that we defined in Section 4.2 address how developers should support global filter queries. To answer these questions we designed 2 types of tests:

1. **Multi-selectivity tests:** In these tests we compare query alternatives at different selectivities[2] over the *selectivity dataset*. To portray better the alternatives we complement the functionality under test with sorting and counting over the results.
2. **Pagination tests:** Here we compare the query alternatives for the global filter query with pagination over the results. In addition we evaluate the combination of pagination with counting the whole number of results. For these tests, we use the *selectivity dataset* and we query on a property-value combination with a fixed selectivity of 0.5.

[2]Please note that in our work we use the term selectivity as a short for *predicate selectivity*. However, to match the concept of being "selective", we define selectivity as the complement on the proportion of items (from the whole set) that are expected to be returned as a result of the query with that predicate. Thus in our notation, a selectivity of 1 means that no items are returned, and a selectivity of 0 means that all items are returned.

Contrary to previous tests from Section 4.3.1, where the retrieval of a complete vertex proved faster than a projection over its properties, here we considered that when retrieving a large number of vertices it might be better to only retrieve the IDs. This decision reduces the memory footprint of our test, enabling us to focus on the core functionality: index access. Furthermore we decided on testing with a simple equality predicate over a single type of data– strings. More complex predicates, queries over edges[3], projections and additional functionality (such as aggregations and grouping) could require more specific test designs.

The size of 450k vertices for the *selectivity dataset* was actually a design decision that took place while implementing the multi-selectivity tests in this section: Larger numbers of vertices caused relatively frequent Out of Memory errors for the heap-intensive graph as an index case, which jeopardized our testing abilities.

In order to properly represent the support for sorting in TITAN, we created a special sorted composite index for the property that we used for sorting.

Before presenting the results we would like to add a comment: For the implementation of the *Graph as in index* concept used in the tests in this chapter, we decided on an approach exactly as outlined in Figure 4.1. Namely, we used a single index vertex as a source for pointing to vertices with specific property values (e.g., propertyA=a or propertyB=b). By limiting our *Graph as an index* model to edges that point to specific values, we could not model adequately with this approach queries with selectivity= 1, therefore we did not include this alternative among the results for this selectivity.

4.3.2.1 Response Time

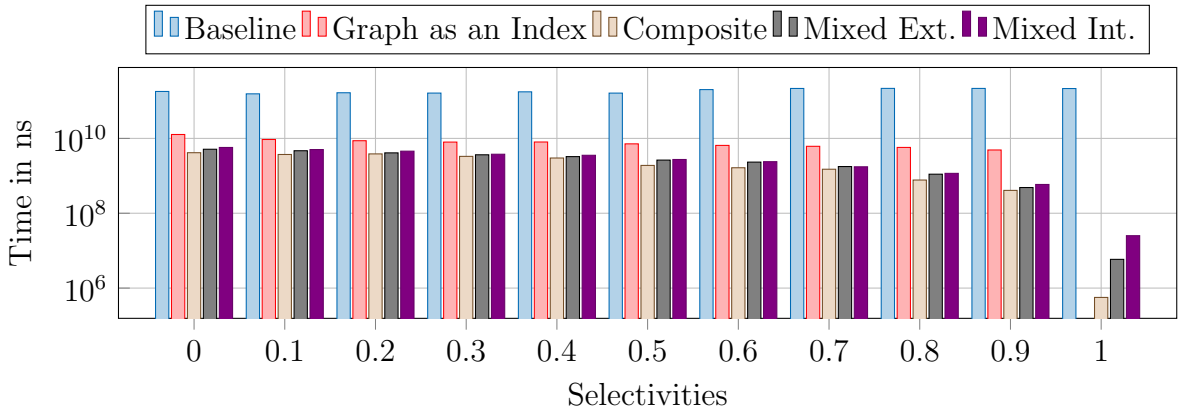


Figure 4.6: Response time of access alternatives for a global filter query with a simple predicate

[3]We already discussed in Section 4.3.1, the fact that edge retrievals have a different performance than vertex retrievals, thus requiring different types of tests.

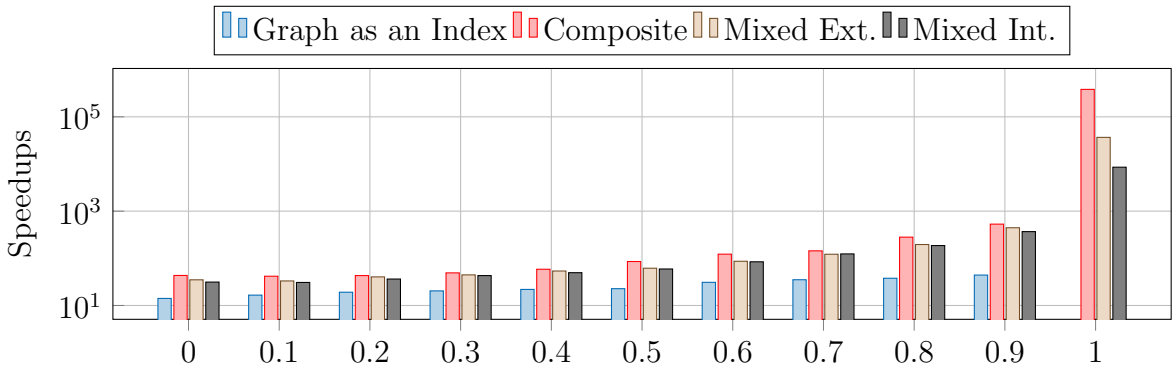


Figure 4.7: Speedups from baseline of access alternatives for a global filter query with a simple predicate

Multi-selectivity tests:

Figure 4.6 presents the response times of the different access alternatives for the basic global filter query in our multi-selectivity tests. Figure 4.7 presents the speedups observed w.r.t the baseline (i.e., a full-table scan).

As expected, full-table scans perform poorly across all selectivities. Furthermore, they perform exactly the same no matter the selectivity. These observations confirm our expectations that full-table scans would be insensitive to selectivity. In contrast, all the other alternatives seemed to scale linearly with selectivity. One unexpected finding in selectivity =0, was that the full-table scan performed at least 14 times slower than the index-based alternatives. We conjecture that these results might be due to a more costly projection over ID in the table than in the index (where the IDs are stored in advance).

Index usage (either mixed or composite) led to the best performance observed with speedups from 30 to 530x on selectivities lesser than 1. For the special case where there is nothing to retrieve (i.e., selectivity=1), composite indexes were notably more efficient than the alternatives, followed by the mixed internal index strategy.

Composite indexes achieved the best performance. The gap between the composite index and the mixed indexes was in the range of 200-1000 milliseconds for selectivities lesser than 0.6 in the external case, and in the range of 400-1500 milliseconds for the internal case; thus it was a significant gap for low selectivities, but it was (arguably) less significant (300 milliseconds or less) for high selectivities.

The graph as an index approach did not perform as well as the index alternatives, but still it produced speedups of 14-44x over the full-table scan.

Figure 4.8 and Figure 4.9 show our evaluation of the same functionality, but retrieving only the number of items matching the query instead of the items themselves. This query would be equivalent to SQL's SELECT COUNT() FROM WHERE clause.

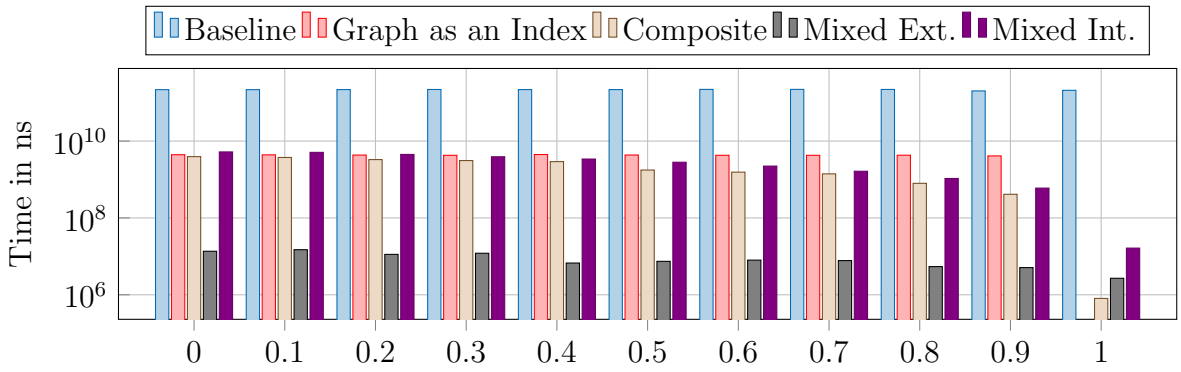


Figure 4.8: Response time of access alternatives for a global filter query with a simple predicate, only counting the results

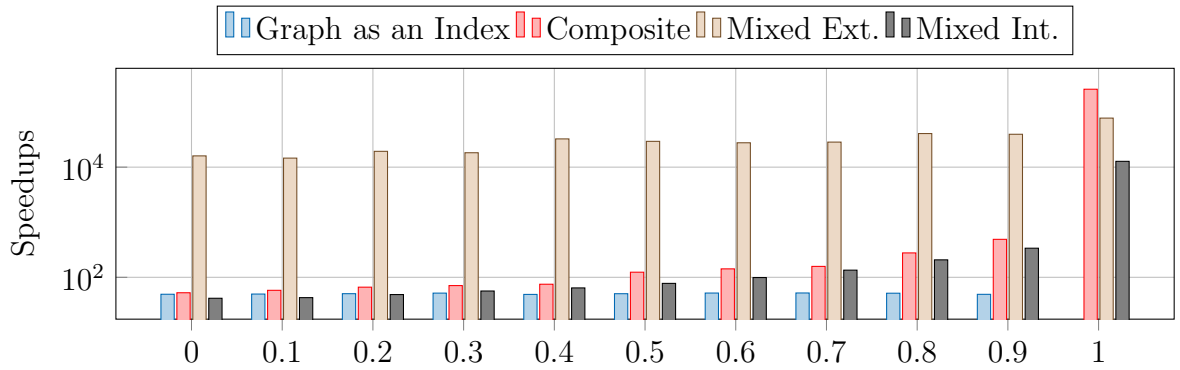


Figure 4.9: Speedups from baseline of access alternatives for a global filter query with a simple predicate, only counting the results

Our results show that the mixed external index performs outstandingly better than the alternatives, with speedups of at least 14000x over the baseline for all selectivities except 1.

Arguably the mixed internal index could perform as well as the external version, but the current Gremlin/Titan support forced us to implement this functionality with a normal IndexQuery, followed by an iterator-based count of the results in Java. In a stark contrast, all the counting happens on the server-side in the external version, and the application only receives the result.

The key reason for the notable efficiency of mixed indexes in counting has to do with optimizations that Elasticsearch brings from Lucene, such as support for faceting (i.e., search by categories) and counting via sampling and complements.

For the full-table scan, the composite index and the mixed internal alternatives we observed no difference between the response times for the global filter query and the global filter count query. Across all selectivities, the graph as an index case was from 1.2 to 3 times faster than its counterpart filter query.

On its part, the mixed external index was from 100 to almost 500 times faster for the global filter count query in contrast to its corresponding case in the global filter query.

Both the insensitivity to selectivity of the full-table scan, and the linear scaling w.r.t selectivity of the other alternatives, were observed again for the global filter count query.

For the edge case where selectivity=1, the composite index continued to perform better than the other alternatives, being 3 to 20 times more efficient than the mixed index cases. We would like to note that in spite of this difference being statistically significant, it has perhaps little practical significance given that the gap between mixed indexes and composite indexes is on on the order of magnitude of a few milliseconds.

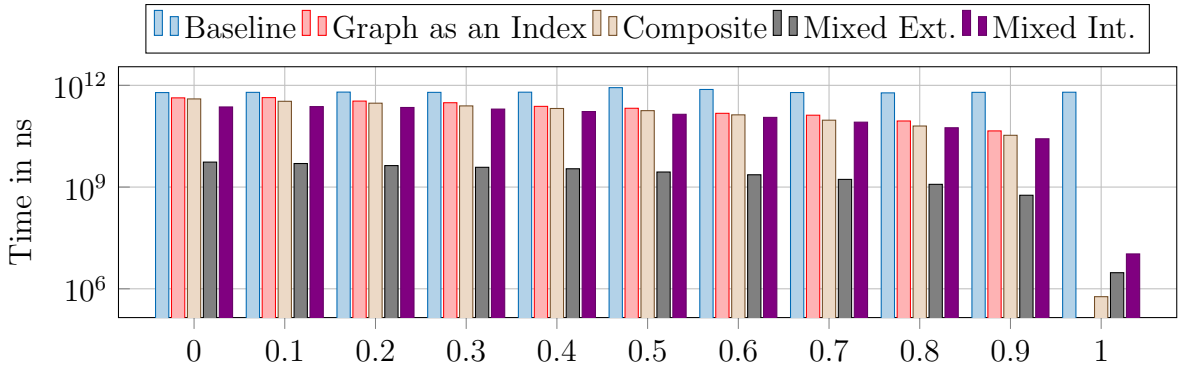


Figure 4.10: Response time of access alternatives for a global filter query with a simple predicate, sorting the results

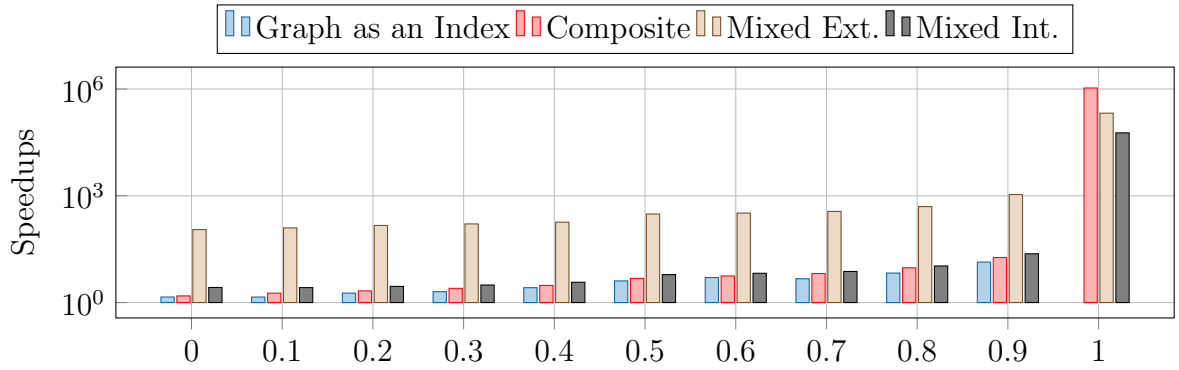


Figure 4.11: Speedups from baseline of access alternatives for a global filter query with a simple predicate, sorting the results

Apart from counting the results, the sorting of these results over a specific field constitutes another common use-case for global access queries. Figure 4.10 and Figure 4.11 collect the response time and speedups for this task at different selectivities.

Sorting was 3-5x slower than the simple global filter query for the full-table scan.

At a first impression the slow-down of sorting could seem like a small difference. However, it is a significant performance deterioration, representing changes in response times

that went from 3 minutes to 10.1 minutes. Response times also increased in almost all the other access alternatives, with slow-downs of 9-46x for the graph as an index alternative, 62-95x for the composite indexes (save for selectivity=1), and 40-53x for the mixed internal index.

Only the mixed external provided a response time unchanged by the sorting. Thus, the mixed external approach displayed speedups higher than all counterparts, reaching values of 112-1082x w.r.t the baseline throughout the different selectivities.

Once again, the mixed internal functionality could have achieved a performance similar to the external approach, however the current version of Gremlin/Titan does not allow to request sorted results. Consequently, sorting has to be carried out on the client-side, with a performance penalty that is presented in our evaluation. In spite of this, the mixed internal variant managed a consistently better performance than the composite alternative.

One reason for the better sorting with mixed indexes might have to do with the special per-shard caching of sort field values in Elasticsearch. Such specialized treatment does not appear to be taken into consideration in the current Titan/Cassandra implementation.

For the edge case where selectivity=1, the composite index continued to be the top performer among all the other alternatives. Similarly to the other cases, the difference with the mixed indexes is the order of a few milliseconds.

Through our multi-selectivity tests we could establish that full-table scans are a costly solution to support the global filter query. These scans should be avoided when possible, in favor of index-mediated global access.

Composite indexes performed the best for the global filter query, followed by the mixed approaches across all selectivities. The gap between the composite indexes and the mixed ones was significant in low selectivities, but insignificant for selectivities higher than 0.6.

In further tests, composite indexes proved to be inflexible to small changes in the functionality (adding sorting and counting). In a stark contrast, the mixed external approach emerged as a significantly better approach across all selectivities for the sort and count functionality.

Our results also show the limitations of the current implementation of the Gremlin/Titan API for querying using mixed indexes. There is no reason why the mixed internal approach should perform notably worse than the mixed external for any functionality. Thus, our results argue strongly for the need to improve the integration between Titan and full-text indexing backends.

Following our study of the multi-selectivity tests, we will consider pagination in the following subsection.

Pagination tests:

When the dataset is signficantly large users might not be interested in retrieving the complete set of vertices in a single request. Instead, users might prefer a workflow where they can paginate through the vertices.

In this section we consider the performance of the different global access alternatives for the same task of performing a global filter query on a simple predicate, but paginating through the results rather than retrieving them all at once. We use the term *pagination* because it is a common term to describe the task. It should not be confused in any way with memory pages. In our case, the page size refers to the number of single items (i.e., vertices) retrieved.

To study this adequately we decided to focus on a selectivity of 0.5, and to evaluate 2 cases: retrieving from page 0, and retrieving from a page further down the sequence. Moreover, given that pagination is usually enabled by having a count of all the results, we considered the combination of pagination and count. A special observation should be made regarding the mixed external index: given its original usage scenarios from common search engine functionality, this alternative returns always the count of all results with each page.

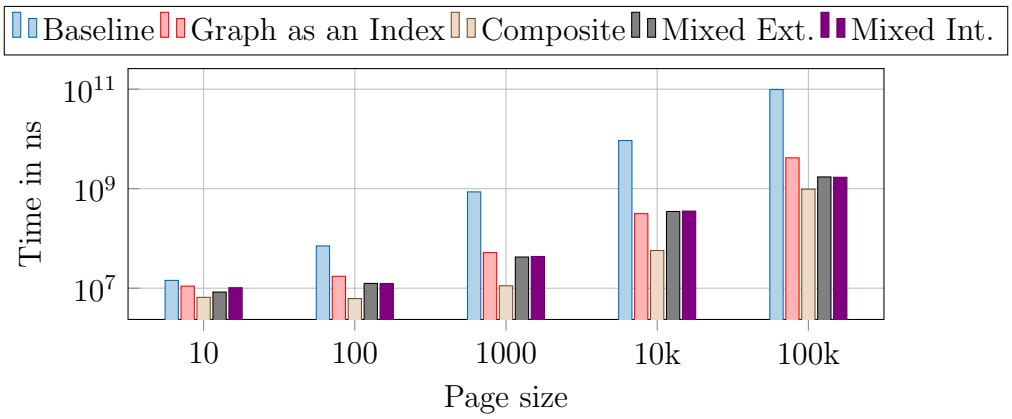


Figure 4.12: Response time of access alternatives for a global filter query with a simple predicate, retrieving a single page, starting from page 0, selectivity=0.5

Figure 4.12 and Figure 4.13 show our results for retrieving a single page starting from 0. In all alternatives the response time scales sub-linearly w.r.t the size of the page retrieved. Similar to our observations for the complete global filter query (Figure 4.6), the composite index performed the best, with performance gains from 2-100x. As expected (based on our observations in Figure 4.6 for large selectivities), the difference between composite indexes and mixed indexes was insignificant for all cases, with the alternatives achieving comparable speedups, but the gap grew to a significant number of 700 milliseconds when retrieving pages of size 100k in both the internal and external uses.

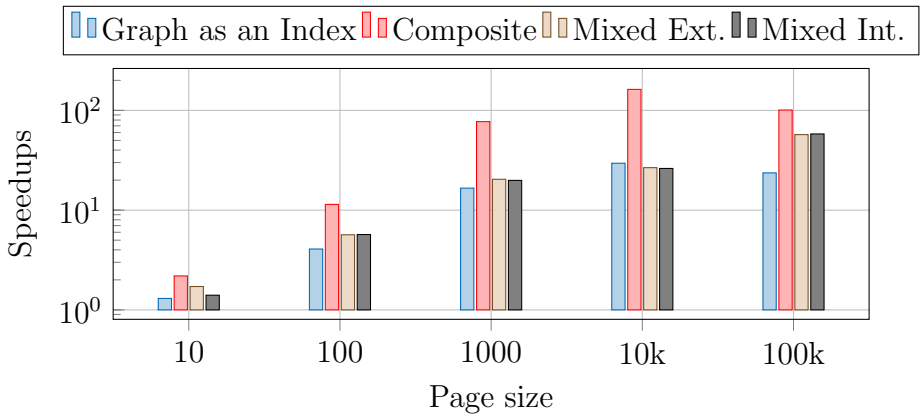


Figure 4.13: Speedups from baseline of access alternatives for a global filter query with a simple predicate, retrieving a single page, starting from page 0, selectivity=0.5

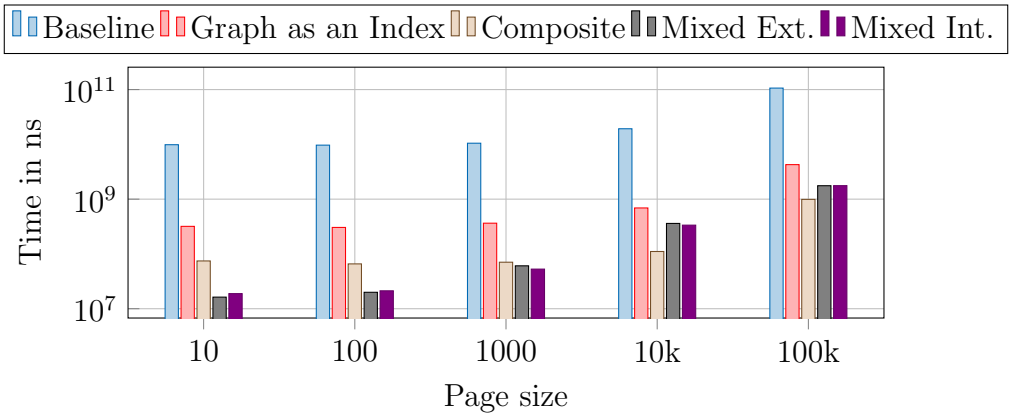


Figure 4.14: Response time of access alternatives for a global filter query with a simple predicate, retrieving a single page, starting from page 10k, selectivity=0.5

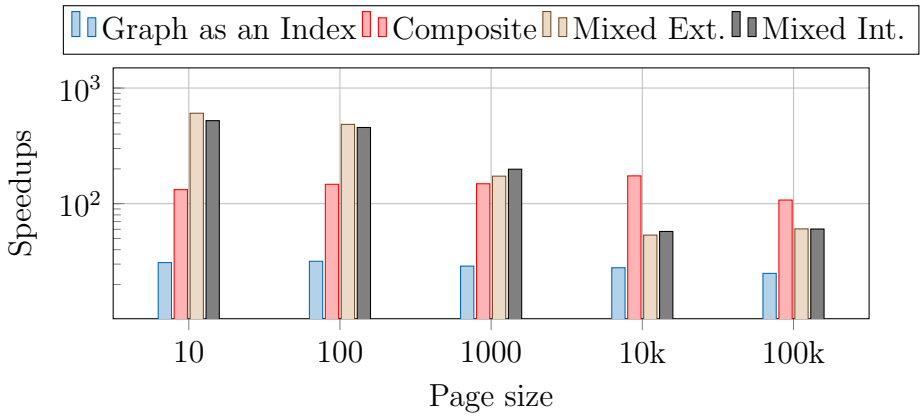


Figure 4.15: Speedups from baseline of access alternatives for a global filter query with a simple predicate, retrieving a single page, starting from page 10k, selectivity=0.5

Figure 4.14 presents our results for the global filter query over a simple predicate retrieving a single page, starting from the page number 10000. Naturally we see some performance deterioration. When compared to their counterparts that start from page 0, each alternative is significantly slower for retrieving pages with a small size, but this evens out as the page size increases, with little difference between starting from 0 and starting from 10k. For small page sizes (10-1000) mixed indexes perform better than composite indexes.

Sub-linear scaling also appears in the response time for the different alternatives.

Figure 4.15 presents the speedups for our tests. These results match with our expectations of significant speedups, because, unlike index-mediated access, full-table scans need to scan the table up till 10k + number of hits according to page size. The smallest speedup observed was for the graph as an index alternative, with approximately 30x performance gains. To put this into perspective, this was a difference from 19 seconds (baseline) to 0.6 seconds (graph as an index).

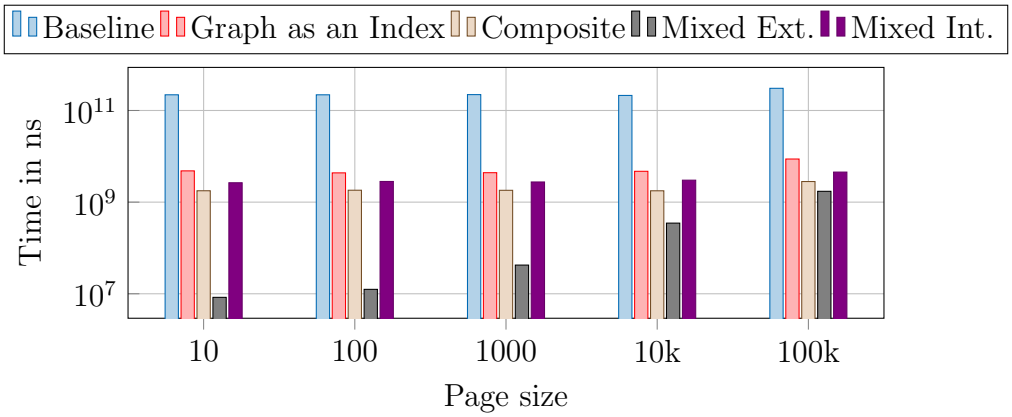


Figure 4.16: Response time of access alternatives for a global filter query with a simple predicate, retrieving a single page and counting the total results, starting from page 0, selectivity=0.5

Pagination is usually made possible by first having the number of overall results. Thus we designed a test that returns a given page next to the total amount of results.

Figure 4.16 and Figure 4.17 display the results of the test, starting from page 0. In this case, linear or sub-linear scaling disappears for all cases except for the mixed external alternative. This specific alternative outperforms significantly all the other approaches, reaching speedups of 26000x w.r.t the baseline.

These results are to be expected, because while counting is offered next to mixed external results, it is only available as an extra call in the other alternatives. Thus, the response time of the alternatives is the response time of the count call plus the pagination call.

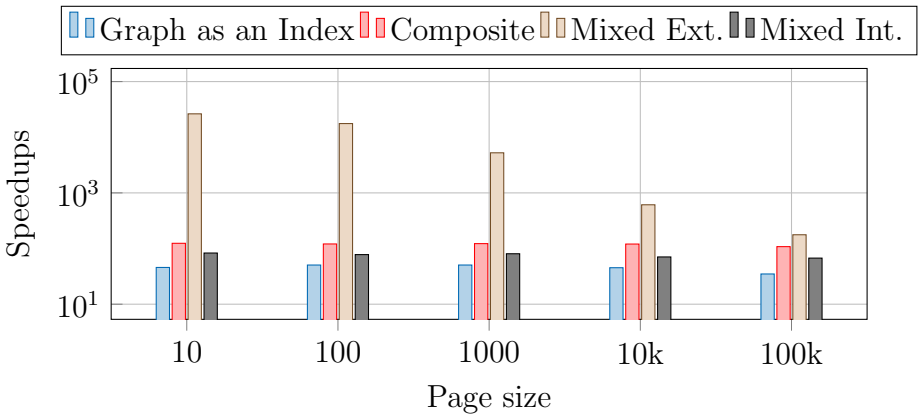


Figure 4.17: Speedups from baseline of access alternatives for a global filter query with a simple predicate, retrieving a single page and counting the total results, starting from page 0, selectivity=0.5

Once again we find that there is no reason why mixed internal indexes should be less performant than the external usage, thus there is room for improvement in the Gremlin/Titan usage of mixed indexes.

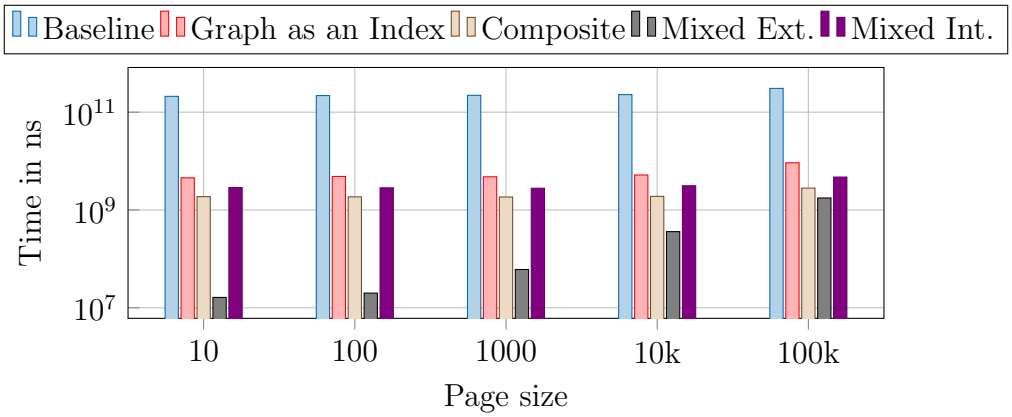


Figure 4.18: Response time of access alternatives for a global filter query with a simple predicate, retrieving a single page and counting the total results, starting from page 10k, selectivity=0.5

Figure 4.18 and Figure 4.19 show our observations for pagination when starting from the 10000th page.

These results confirm our expectations that mixed external indexes would outperform the alternatives, with speedups of 174-13000x w.r.t the baseline. No other alternative achieves comparable performance gains.

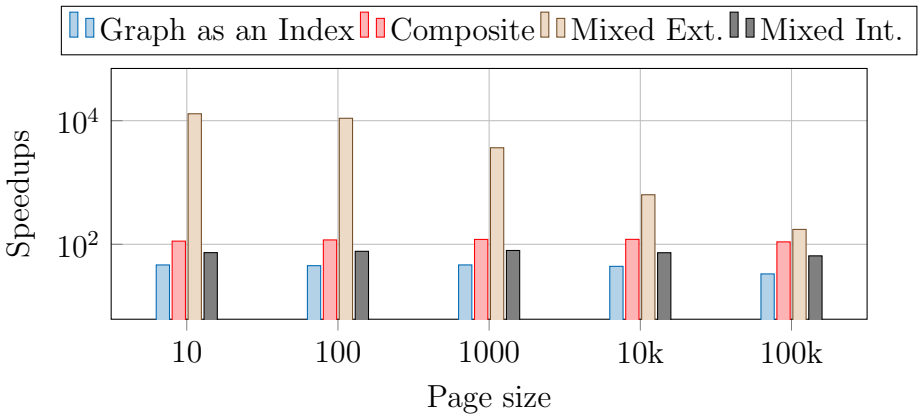


Figure 4.19: Speedups from baseline of access alternatives for a global filter query with a simple predicate, retrieving a single page and counting the total results, starting from page 10k, selectivity=0.5

4.3.2.2 System Performance Observations

Gremlin profiles of the full-table scan for the global filter query reveal that in the basic case and for counting, all time is spent on a single TitanGraphStep, which is further subdivided in an optimization phase (taking from 8-25% of the time) and a scan phase. Sorting introduces drastic changes into the profile of this query, by appending an OrderGlobalStep, which takes approximately half of the query time. Similar observations occur for the composite index case, with the exception that the scan phase is exchanged for a backend-query phase which takes about 40% of the time assigned to the TitanGraphStep. These profiles concur with our understanding of Titan: indexed and unindexed queries are handled in the same way, but can be distinguished after the optimization phase.

Consistent with our expectations, the graph as an index approach has a more complex profile for the basic global filter query, consisting of the following steps and percentages: GraphStep (20%), TitanVertexStep (1%), TraversalFilterStep (less than 1%) and to end an EdgeVertexStep (16%). Counting changes the basic profile by adding replacing the EdgeVertexStep with a CountGlobalStep (75%). Sorting changes the basic profile by adding an OrderGlobalStep (30%).

Pagination added a range step to all traversals that took 20% of their time at most.

Mixed index usage could not be profiled in the same manner as the alternatives, because, as we mentioned before (see Section 4.1.2), Titan does not include these indexes into the same query optimization framework of composite indexes.

The heap usage of TitanLab was monitored with JConsole for individual runs of the different alternatives in the whole microbenchmark (global filter query, count, sorting and pagination cases). Through this we established that the graph as an index approach led to the highest heap usage, with 500 Mbs (around 50% of the heap available) and 30% of CPU usage. This was followed by the composite index approach, which utilized

400 Mbs, and next the mixed indexes, which used from 250 to 300 Mbs in both usages. The composite and mixed indexes employed the CPU on a 25% basis. The baseline case, although significantly slower than its counterparts, had a heap footprint that never exceeded the 90 Mbs, and CPU usage was on average around 5%.

Although Cassandra monitors and system metrics provided copious data about the microbenchmarks, we did not find any special information that could give us insights about the distinctive performance of the alternatives. Elasticsearch monitors showed that both mixed index approaches had a similar usage of the heap (for the service), with a 59 to 70% footprint.

4.4 Best Practices

Following the tests in this chapter we are prepared to define some best practices regarding global access. These best practices are a direct answer to the questions that spawned our evaluations for this chapter (Section 4.2).

1. Titan IDs are the best choice for retrieving items, in contrast to unique identifiers. The argument for this access method comes from its notably better handling of random access, not from a particularly important speedup in individual access.

2. Indexes are essential to support global filter queries and small alternatives such as counting and sorting. Indexes can lead to significant speedups of at least 14, 50 and 1.4x for the basic case, counting and sorting, respectively. Full-table scans should be avoided altogether for global filter queries, given their extreme inefficiency in contrast to index-mediated access.

3. Composite indexes are the best tool to use for global filter queries, but their role as the best tool does not extend for global filter queries with pagination (when retrieving pages smaller than 10k), counting and sorting. For these cases mixed indexes used externally are by far the best tool.

4. Currently, mixed indexes are not supported to their fullest by Gremlin/Titan, thus, for more functionality we suggest the possibility of using Titan's mixed indexes via an external Elasticsearch API. Titan's evolution might benefit by devoting some work to better integrating mixed indexes into the query optimization framework that nowadays can only use composite indexes.

5. Before using mixed indexes for a given functionality, developers should consider the key limitation of this approach (see Section 4.1.2). Namely, that after retrieving the items from a mixed index, they cannot be used immediately for traversals. Instead, a new traversal has to be created, starting by querying items through their retrieved IDs. Thus, even though mixed indexes can lead to performance gains in the global filter query with pagination, sorting and counting, they might not be the most suitable choice when this query is only the entrypoint to more complex graph traversals.

6. For simple predicates, as the ones considered in our tests, other index alternatives should be chosen over the graph-as-an-index approach. Further evaluation has to be carried out to see if this holds for complex predicates. Theoretically, it's possible that the graph as an index could outperform composite indexes on single properties for complex multi-property predicates.

7. The index alternatives that we considered might differ in their stability to random access, but such a study was beyond our scope. Before choosing an index, we advice developers to test how this index responds to the access patterns in the use cases of their deployed applications.

4.5 Summary

In this chapter we evaluated alternatives for global access. First we outlined what these alternatives were: full-table scan, access by ID, composite indexes, mixed-indexes (with external and internal requests) and using the graph as an index. Next we described the microbenchmarks that we designed to compare the alternatives for the tasks of retrieving a single item, or for the global filter query. We also discussed our results. Stemming from these we were able to propose some best practices for application developers to support global access with Titan.

In the following chapter we continue with our task of answering the research questions defined in Section 3.1. Specifically we focus on some key aspects that might affect the performance of composite indexes: cardinality, and how Titan currently supports index selection.

5. Case Study: Composite Indexes

In this chapter we discuss our study on the performance impact of two very specific aspects of using Titan's native global indexes (i.e. composite indexes): the effect of index cardinality on retrieval times, and the role that Titan's index selection can have on performance of global filter queries with conjunctive predicates.

This chapter is outlined as follows:

- **Evaluation Questions** We begin by presenting the research questions that motivate our evaluation (Section 5.1).
- **Microbenchmarks** Discussion of the microbenchmarks of our evaluation (Section 5.2) and the the results of them. We divide this section in two parts, corresponding to the specific tests (Section 5.2.1, Section 5.2.2).
- **Best Practices** A list of the best practices that can be inferred from the evaluation in this chapter (Section 5.3).

5.1 Evaluation Questions

As we saw in the previous chapter (Chapter 4), composite indexes represent a good solution for answering global filter queries on simple predicates. As a developer, there might be an interest on evaluating how these indexes perform for more complex predicates.

Additionally, bearing in mind that data cardinality (i.e. the number of different values for any given set of data) is often a factor that needs to be considered before creating an index on a given field/property, developers might be interested in knowing how composite indexes fare under different data cardinalities.

The specific concern with data cardinality stems from the fact that most indexes split the indexed data into subsets according to the values of the indexing field. When this indexing field has only few distinct values (i.e., there is low cardinality) the subsets can become very large and the efficiency of the index to support fast searches might be deteriorated. In addition, when the indexing field presents a large number of distinct values (i.e., it has high cardinality) the indexing structure can become very large, affecting the performance of the index. These concerns motivate developers and database designers to assess the effect of cardinality on the indexes that they use.

Based on these information needs we can formulate our evaluation questions for this chapter:

4. Practicioners report that extreme cardinality data might be detrimental to the performance of common indexing structures in mainstream databases, such as B-trees[1], and Cassandra's secondary indexes[2]. Considering that Cassandra-based composite indexes are the basic indexing approach of Titan when using Cassandra, it would be important to determine: How do composite indexes perform under extreme cardinality (very low or very high) cases?

5. Queries with conjunctive predicates can be answered either by an index over all query fields (which requires database developers to create the indexes with the conjunctive predicate query in mind), or by combining different indexes (which allows for more generic indexing). As a developer supporting ad-hoc queries, it would be valuable to know: How does Titan handle multi-index conjunctive predicate queries with its native indexes, and is there any performance trade-off that should be kept in mind regarding this?

These are the evaluation questions 4 and 5 of the list presented in Section 3.1.

5.2 Microbenchmark

To answer these questions we proposed two different tests based on the basic global filter query, as defined in Section 4.1, and supported by composite indexes. We present these tests next.

[1]This has been studied by IBM's DB2 community: https://www.ibm.com/developerworks/data/library/techarticle/dm-1309cardinal/

[2]As an example, [Mis14] specifically claims that: *"(an index over) a column with high cardinality may result in performance overhead under high data load and large* Cassandra *clusters."* Similar observations are given by DataStax, the main vendors of Cassandra's enterprise versions: https://docs.datastax.com/en/cql/3.1/cql/ddl/ddl_when_use_index_c.html

5.2.1 Composite Index Cardinality

For evaluating the effect of data cardinality on composite indexes we decided to add new properties to the *selectivity dataset* and the *basic dataset* as defined in Section 3.4. Specifically we added 3 properties with a 0.5 selectivity (i.e., from 450k vertices, half of them have a given value for these properties). In the first case we made the other 0.5 of data have the same value (low selectivity case), for the second we made the other 0.5 have a random assignation of up to 30 different values (normal selectivity case), for the third we made the remaining 0.5 of the data have unique values (high cardinality case). Next, we simply tested the response time of the basic global filter query on the modified datasets.

Figure 5.1 records the response times for the global filter query on composite indexes of varying cardinality. High cardinality indexes ranked consistently as the slowest for supporting this query. This agrees with the expectation that high cardinality would affect the performance of composite indexes. By contrast, low cardinality indexes performed slightly better than normal cardinality indexes in our evaluations. These observations does not agree with the expectations that low cardinality indexes might affect read performance negatively.

Low cardinality indexes translate into wide rows in Cassandra. This in turn might deteriorate common housekeeping tasks, such as compaction and repairs. On very large datasets, this could become an issue. We consider that the small gains that we observed for this case might be exclusive to the size of datasets that we evaluated with. We consider that on larger datasets low cardinality might lead to a different behavior.

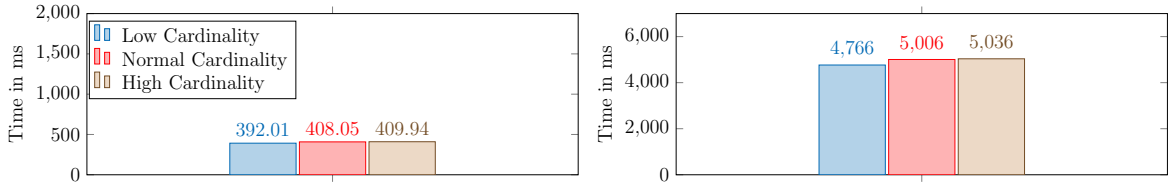
The highest difference observed between the best and worse responses was of 0.3 seconds, and the speedups when comparing best to worse cases were never beyond 1.1x. Because they are not separated by at least one order of magnitude, we conclude that the small divergences observed for the different cardinalities have limited difference.

As a conclusion from our tests we can establish that on the restricted domain of small datasets (up to 3.1 M nodes), cardinality does not have a notable effect on the response time of reads using composite indexes. Further studies are needed to determine the extent to which cardinality might affect these indexes in larger datasets.

5.2.2 Index Selection in Titan

Apart from retrieving items by their IDs, or by matching a simple equality predicate over an indexed property, conjunctive predicate queries are another way to retrieve items in Titan. They represent, in fact, a variation of the single-predicate global filter query that we have considered, with the difference that several equality predicates are asked over different properties.

One example of a conjunctive predicate query would be asking: `g.V().has("propertyA", "valueA").has("propertyB", "valueB")`. To answer such a query with composite indexes, Titan must first determine which indexes to use. Naturally, if an index covers



((a)) Selectivity Dataset ((b)) Basic Dataset

Figure 5.1: Response time for a global filter query with a simple predicate, selectivity=0.5, using a composite index with different cardinalities, on the selectivity and basic datasets

all properties, TITAN should employ such an index. However, in the case that several non-overlapping indexes cover different properties, theoretically it would be possible to select the order in which indexes are accessed, considering their selectivity, for improving the performance.

We studied TITAN's codebase[3] to determine how this index selection and ordering of access takes place for composite indexes. We established that the core of this process is implemented in the com.thinkaurelius.titan.graphdb.query.graph.GraphCentricQueryBuilder class, in the constructQueryWithoutProfile function. Here we see that indeed indexes that cover more properties are selected usually. Next, we found that a jointQuery is assembled, constituting an ordering over the subqueries. The results of the JOINTQUERY are finally processed with iterators, starting from the first subquery.

As a small test on how this ordering might affect the query performance, we designed a simple evaluation with one conjunctive predicate query made of 2 subqueries (simple equality predicates over indexed properties) with different selectivities. Specifically, we made one of those subqueries to have the lowest possible selectivity (i.e., 0), while we made the other subquery take selectivities from 0.1 to 1. Each subquery was only answerable by a given index. In our evaluation we assessed how TITAN handled the query, taking the response times and also checking which subqueries were placed first in the JOINTQUERY. At last we changed TITAN's codebase to force it into: a) choosing continuously the most selective subquery as the first one in the JOINTQUERY, b) choosing always the least selective first.

To put our results into perspective we also ran this evaluation with the other index alternatives discussed in Chapter 4 and the addition of a composite index covering both properties.

These tests were carried out using the *selectivity dataset*.

Figure 5.2 collects the results of our evaluation. We found that TITAN only placed the optimal subquery (i.e., the most selective one) first on selectivities 0.4 to 0.7; on all the other cases TITAN made the wrong decision and selected the worse-case ordering (i.e., placing the less selective subquery first).

[3]Avaliable here: https://github.com/thinkaurelius/titan

Figure 5.3 shows the possible speedups of selecting the optimal ordering (i.e., starting always with the most selective subquery). On selectivities over 0.8, the possible gains are from 10-15x. Overall, the selection of the optimal ordering in our tests led to response times that were from 4 to 96 seconds faster than TITAN's selection.

Figure 5.2: Response time of a conjunctive predicate filter query over 2 composite indexes. Effect of different ordering of subqueries.

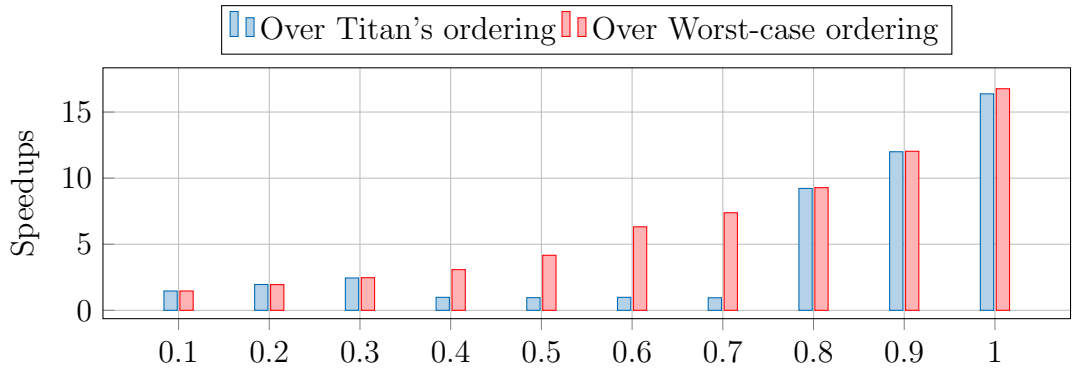


Figure 5.3: Speedups of choosing the Optimal subquery ordering, compared to TITAN's ordering and the Worst-case ordering of a conjunctive predicate filter query over 2 composite indexes

As a first conclusion from this evaluation, we can strongly advice TITAN developers to improve the current subquery ordering for composite index usage in the GraphCentricQueryBuilder class. We consider that this is an important issue because query performance in TITAN can differ significantly, with a gap of at least one order of magnitude, from the optimal. Runtime selectivity estimators and statistics could be useful mechanisms for establishing the most beneficial ordering.

Figure 5.4 collects our evaluation on how the response time for the different subquery orderings compares to the response time of processing the query with other global access alternatives.

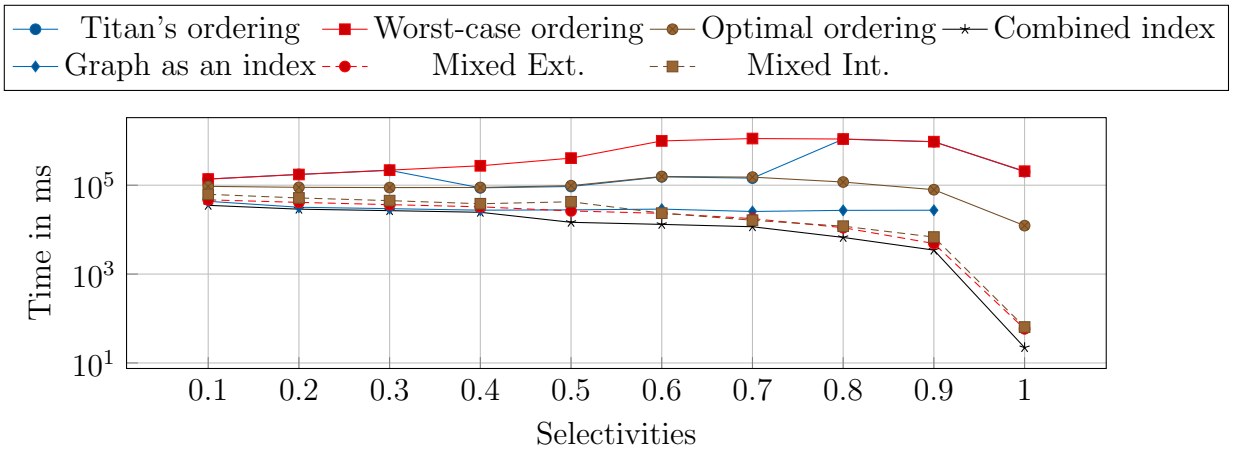


Figure 5.4: Response time of a conjunctive predicate filter query over 2 composite indexes. Results considering different ordering of subqueries and global access alternatives.

A special disclaimer should be made, that the graph as an index approach is not properly evaluated in our tests, as due to relatively frequent memory errors we decided to limit these results to a maximum of 150k which provided stable measurements.

This evaluation shows that the approach of using a combined index over both properties performs consistently better than the alternatives, followed closely by the mixed indexes.

The limitations in our testing of the graph as an index case only allow us to observe this to perform less efficiently than the mixed approaches.

The optimal ordering solution was comparable to the mixed index internal case for low selectivities, but the gap between both became larger as selectivities grew. On the other hand, the mixed external and the composite combined were from 2.6 to 22 times faster than the best ordering of using multiple composite indexes.

As a conclusion from this evaluation we can establish that the management of subquery ordering for conjunctive predicate queries currently limits Titan's performance when it must combine results from several composite indexes. Given this, we can strongly recommend the use of mixed indexes for ad-hoc queries, when possible. Composite indexes covering all properties in the query would be the best solution, but this requires some extra planning to create the indexes in the first place. In our evaluations we could not determine if the graph as an index solution is a viable alternative to improve the performance of ad-hoc conjunctive predicate queries.

5.3 Best Practices

Following our evaluations, we can propose a list of best practices regarding the use of composite indexes. These are a direct answer to the evaluation questions in this chapter (Section 5.1).

1. On small datasets (of less than 3.1 million vertices), cardinality does not seem to be an issue that affects in a notable way (i.e., with slowdowns of at least one order of magnitude) the performance of Titan's composite indexes for global filter queries. However, as a developer working with larger datasets, it would be important to assess if there is any impact of cardinality on these indexes. Developers should keep in mind that the Cassandra community warns against using secondary indexes for extreme cardinality values. Given that Titan can rely on Cassandra for composite indexes, the advice of the Cassandra community should be taken into account.

2. For the simple case of a conjunctive predicate query, the use of 2 composite indexes (one over each property queried against) did not appear to be the best solution when compared to mixed indexes or to a composte index covering both properties. Furthermore, we can advice against using several composite indexes to answer a conjunctive predicate query, in consideration that there are issues on how Titan orders the subqueries, which can lead to sub-optimal performance. In light of the limitations of mixed indexes (see Section 4.1.2), the graph as an index seems like a promising alternative for ad-hoc queries. However, since our evaluation did not cover this entirely, it is up to developers to assess if the graph as an index approach could help to improve ad-hoc conjunctive predicate queries over multi-index support with composite indexes in their use case.

3. The alternative of answering a conjunctive predicate query by a single composite index (that covers all the properties queried against) might be the best solution to the limitations of multi-index performance for the task. This alternative, however, might not be suitable for ad-hoc analysis, because creating such unique indexes requires special planning.

5.4 Summary

In this section we evaluated two specific aspects that shed more light on the capabilities of composite indexes: their sensitivity to extreme data cardinality, and how Titan uses these indexes to answer conjunctive predicate queries. As a product of our tests we were able to propose a couple of best practices related to these indexes.

In the upcoming chapter we will continue our evaluation on basic best practices for developing Titan applications, following our core research questions (Section 3.1). Concretely, we will tackle possible best practices for local traversals, including the consideration of using denormalization as a read optimization.

6. Case Study: Local Traversals and Denormalization

This chapter moves away the focus of our evaluations from global access to local functionality such as simple hop traversals. In this way we can broaden our understanding on best practices for using Titan over a wider set of use cases.

We begin our evaluations by looking into the sensitivity of traversal response times to different topologies with the same number of vertices. Then we investigate the potentials of denormalization to improve traversals.

This chapter is organized in the following manner:

- **Evaluation Questions** The definition of the exact questions that we aim to answer with our evaluation (Section 6.1).
- **Microbenchmark** Tests and results of microbenchmarks (Section 6.2), measuring sensitivity of traversal time to topology variations (Section 6.2.1) and offering a practical study on the possible impact of denormalization to speed up traversals (Section 6.2.2).
- **Best Practices** We wrap-up this chapter by recapping the best practices that arise from our studies (Section 6.3).

6.1 Evaluation Questions

Traversals constitute a different query pattern than simple global queries. To traverse is to move from one given vertex to another, following edge paths, with different logics for moving through these paths. Traversals are a way of accessing local elements rather than global ones.

Traversals of different complexity are at the center of graph analysis tasks. Accordingly, developers could benefit by knowing how to properly design and use traversals.

On a very specific level, this involves figuring out the best Gremlin steps and traversal strategies for the task at hand. On a more general level, this implies resolving how to best represent the graph, within the possibilities of the database, to speed up certain traversals. In this chapter we consider these more general issues about traversals.

To be precise, we intend to examine the following questions:

6. When traversing over a given number of nodes, how sensitive is the performance of basic traversals to variations in the topology?

7. What are the potential gains from the strategy of denormalizing vertex data (by duplicating it as properties of incident edges) to speed up traversals?

These correspond to questions 6 and 7 from the core research questions in Section 3.1.

6.2 Microbenchmark

We designed two separate tests to address our evaluation questions. For the first test, discussed in Section 6.2.1, we used the *traversal dataset* as introduced in Section 3.4. This dataset is a variation on the *basic dataset*, with an addition of 3 small subgraphs that represent small local topologies for our testing. This will be explained in the corresponding section.

For the second test, included in Section 6.2.2 we used the *selectivity dataset*.

6.2.1 Local Traversals

Simple traversals were implemented in TitanLab for this test. Additionally, for this test we introduced 3 small subgraphs into the *basic dataset*. Each subgraph constitutes a different topology connecting 509 vertices within 9 hops of a selected seed vertex. These are the topologies that we devised:

- *Path topology* corresponds to a layout where there is a single path of 8 edges from the seed vertex to a given "small supernode" vertex fom which there are 500 edges to the remaining 500 nodes of the topology.

- *Balanced topology* forms a hierarchical layout with the seed vertex as root and the remaining vertices arranged in such a way that each vertex has the double of outgoing edges as incoming ones, starting from 2 outgoing edges from the seed vertex, until covering 509 vertices.

- Finally, *Binary tree* stands for the case where each item has only 2 outgoing edges.

These topologies do not aim to represent in any way real-world cases, instead they are synthetic topologies intended only to evaluate the effect on traversal times of changes in the number of vertices needed to access in order to retrieve all edges and subsequent vertices. The *path topology* represents a case with a very little number of vertices needed to access (9), the *balanced topology* increases this number, and the *binary tree* brings it to a higher value.

We designed a simple test consisting of elementary 9 hop traversals retrieving all items in these varied topologies, and measuring the response time.

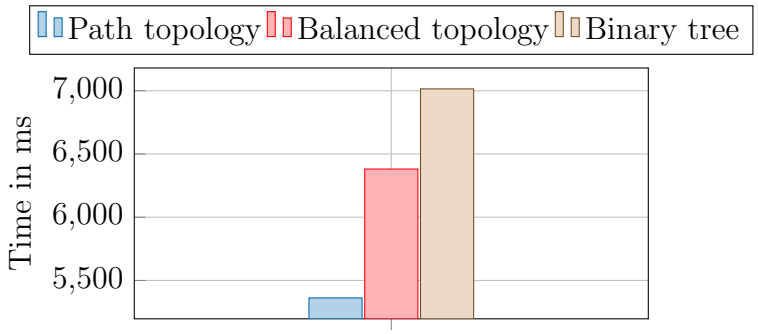


Figure 6.1: Response time of 9 hop traversals from a seed vertex, retrieving all items in different topologies of 509 vertices.

Figure 6.1 shows the average response times of our tests. The path topology achieved consistently the best response times, which corresponded to 1.2-1.3x gains over its counterparts. While the gains might seem paltry, they represent from 1 to almost 2 seconds of a faster response time for the path topology. Thus, we consider that these are important differences, with visible implications to end-user response times.

As a conclusion from this small test we can state that Titan does indeed seem to be sensitive to variations in the graph topology for basic traversals in small datasets. From this we can extrapolate that processing might display a greater sensitivity to topology variations in larger datasets.

From our findings we can suggest developers to consider TraversalStrategies and options as denormalization of data between elements, to compensate for these differences. Future work in the development of Titan might also bring forward improvements in traversing.

In the next section we consider one of the known alternatives to improve traversals: denormalization.

6.2.2 Denormalization as an Approach to Improve Traversals

Denormalization refers to the strategy of making redundant copies of stored data, with the goal of accelerating analysis tasks. In the case of graphs, this can be achieved by copying vertex properties into their connected edges, or edge properties into their connected vertices. On runtime these copies can be used to avoid extra lookups in filter statements.

In order to give a practical idea on the possible gains of this strategy, we designed a basic test using the *selectivity dataset*. This test consists of a 2 hop traversal from a super-node with 450k outgoing edges. All of these edges pointed towards vertices where a given property value had a selectivity of 0.5.

Concretely, in the test we tried two approaches:

- *Basic traversal*, a first hop through all outgoing edges of our seed vertex, followed by a second hop passing through all vertices matching the property value with selectivity 0.5.
- *Traversal optimized with denormalization*. For this case we copied the property value used for filtering into the connected edges. This denormalization changed the traversal into the following case: a first hop through all outgoing edges of our seed vertex, filtering on the given property value, followed by a second hop over all vertices found.

Both alternatives retrieve the same results.

Figure 6.2 presents the response time of the *Basic traversal* and the *Traversal optimized with denormalization* traversal. Within the limited dataset of our test case, denormalization produced a speedup of 34x, which translated into a notable difference in performance of 1.8 minutes.

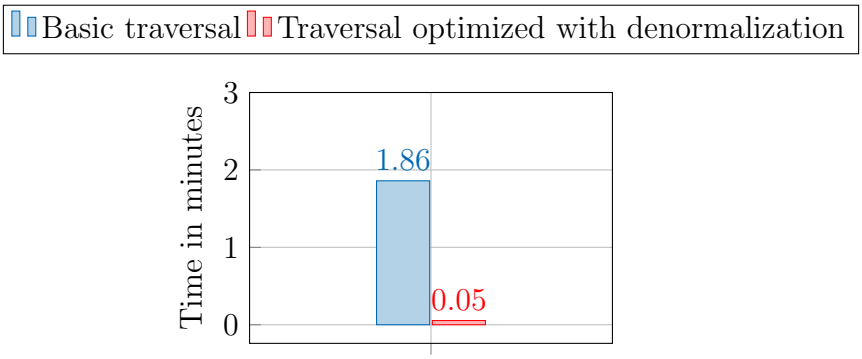


Figure 6.2: Response time for 2 hop traversal over 450k edges in a basic traversal or in a traversal optimized with denormalization

Our results clearly validate that denormalization is an important mechanism to improve traversal response times.

For the limited dataset used, JConsole showed a decrease in heap usage for repeated runs of our test in the denormalization method. Heap usage reached 170 Mbs for the basic case and 70 Mbs for the denormalized version. These observations emphasize the reduction in access to vertices achieved by denormalization.

6.3 Best Practices

Building on the experimental findings presented in this chapter, we can now define some general best practices related to traversals. These practices are a direct answer to the evaluation questions that motivated our research for this chapter (Section 4.2).

1. Much like in general graph algorithms, the performance of traversals in Titan is affected by the topology of the data. This might be true for all other graph databases. Even in small datasets we observed this to produce performance differences in the range of seconds. A corollary from this realiztion is that both the database processor and the developer are tasked with improving traversals according to topology specifics. In circumstances it might be possible that the graph model (i.e., the design of what to model as vertices, edges and properties) could be tuned to create topologies that improve traversals.

2. A general way for improving traversals is to denormalize vertex properties towards the edges, thus reducing access to elements and some of the influence of the topology. On small datasets this can lead to performance gains of 34x, amounting to making a 2 minutes function return in 3 seconds. Regarding denormalization we can make some more fine-grained comments:

    - In order to understand why denormalization works so well in Titan, we propose developers to stop thinking about graph elements, and instead concieve the graph as a set of rows, where each row corresponds to a vertex and its connected edges. Through this model it is easier to grasp that when vertex filters are applied, the denormalization of vertex data to edges reduces row accesses, making traversals faster.
    - Prior to adopting denormalization as a strategy, developers should take into consideration the tradeoffs from data denormalization: increases in memory footprint, reductions in write throughput, the higher potential for data inconsistency, among others. In fact, it is likely that on large datasets, there could be cases where the increased memory use of denormalization could lead to worse performance than the basic alternative.
    - Following our experiments we can recommend a limited use of denormalization as an optimization strategy for small datasets, if developers accept the tradeoffs and if denormalization can be indeed exploited in the targetted traversal algorithms.
    - For larger datasets we urge developers to assess carefully the increased memory usage of denormalization before adopting it as an optimization solution.

- To conclude, we can recommend the exploration of traversal strategies, given that they might be a more sustainable solution than denormalization for optimizing traversals.

6.4 Summary

This chapter presented our evaluation on the performance of general traversals in Titan. Instead of addressing specific concerns in tuning traversals for a given graph algorithm, this chapter aimed to provide experimental insights on general traversal issues. These issues were: the sensitivity of Titan to topology variations when traversing a determined number of items; and the role of denormalization as a tool to accelerate traversals.

The next chapter is concerned with an optimization strategy that, similar to denormalization, might be applicable to a broad set of graph analysis tasks: exploiting the multi-store integration of Titan with search engines. The following chapter will conclude our evaluations in this study.

7. Case Study: Opportunities from Search Engine Integration

In this chapter we survey some opportunities that arise from the integration between graph databases and search engines. In previous chapters we've appraised some benefits and limitations of mixed indexes in supporting global access. Here, we expand the functionality under consideration, by illustrating how mixed indexes used externally can provide novel tools to improve general graph algorithms.

We outline this chapter as follows:

- **Evaluation Questions** We establish the evaluation questions that motivate this chapter (Section 7.1).
- **Microbenchmark** Experimental analysis and results to answer the evaluation questions (Section 7.2). Particularly we test the calculation of the average degree centrality of vertices in the graph. In our tests we considered basic Gremlin implementations, tuned with composite indexes, and an Elasticsearch-based implementation that indexes all edges and copies on them the IDs of their connected vertices (Section 7.2.1).
- **Best Practices** To conclude, we summarize the findings of this chapter in a list of best practices (Section 7.3).

7.1 Evaluation Questions

Mainstream graph databases such as Apache Titan and Neo4j offer developers the alternative to integrate the database with a search engine, such as Solr or Elasticsearch, for full-text indexing and queries. This integration provides a set of search

engine functionality which is supported by the graph database as an *internal* alternative (e.g., indexQueries in Titan). Equally important, the search engine itself offers further functionality that, although not supported directly by the graph database, can still be exploited by the application developer as an *external* alternative (an example of the latter was given in our use of mixed indexes with the external Elasticsearch API).

This state of affairs tasks developers with the question on how to leverage effectively together search engines and graph databases, via internal or external calls, for a given task. Such a question is indeed task-specific in nature, demanding careful design, testing and prototyping to reach an answer.

In order to provide some guiding insights for developers facing this question, we address in this section a more general concern about the extents of performance contributions possible from an improved search engine integration:

8. What amount of performance improvement can be expected by leveraging search engine functionality in traditional graph algorithms and what are the tradeoffs involved?

This is the last of our core evaluation questions, as presented in Section 3.1.

7.2 Microbenchmark

For our evaluation we developed a single microbenchmark consisting on the task of calculating the average degree centrality (in-degree and out-degree) of all vertices in a graph.

As datasets we used the synthetic *basic dataset*, and two real-world datasets, *Openflights* and *Pokec*. Our decision to adopt real-world datasets for this microbenchmark arose from the consideration that tests in this section had less requirements from synthetic data characterstics (i.e., special selectivities, cardinalities, topologies or supernodes) than other tests in our study. The datasets used are described in Section 3.4.

7.2.1 Search Engine Integration for Calculating the Average Vertex Degree Centrality

The degree centrality of vertices is one of the many metrics to characterize and understand the actors of a network.

The degree centrality of vertices can be defined as the number of edges (either incoming or outgoing) connected to them. The in-degree centrality of a vertex indicates how *prominent* this vertex is within the network. The out-degree centrality indicates how *influential* a vertex is. The average vertex degree centrality for a graph is the average of the degree centrality of its vertices.

Our tests evaluated three alternative implementations for calculating the average vertex degree centrality:

1. **Basic implementation** A basic GREMLIN implementation, based on recommendations from the TINKERPOP community[1]. The key traversal for calculating the in-degree centrality of all vertices was implemented as follows: `g.V().in("label").inV().group().by(__.ID()).by(__.inE(label).count())`. Note that the traversing over incoming edges of a specific label (i.e., `g.V().in("label")`...) was added to represent the case for multi-graphs (i.e., in our microbenchmarks only the *basic dataset*). In general it can be avoided, so the traversal can be written as: `g.V().inV().group().by(__.ID()).by(__.inE(label).count())`

2. **Composite index implementation** Observing that the *basic implementation* requires a full-table scan on the vertices before selecting those that match a given relation, and having proven experimentally the inefficiency of this approach (Chapter 4), we decided to improve the traversal for multi-relation graphs by using composite indexes. With this in mind we created a new boolean property on vertices that would act as a flag for them having an edge with a given label. We indexed this property with a composite index. Accordingly, our resulting traversal was re-written in this way: `g.V().has("hasLabel", Boolean.TRUE).group().by(__.ID()).by(__.inE(label).count())`. The selection of indexing a boolean property for this case could be a wrong design choice given the low cardinality of the property. However, we have previously shown that cardinality has little effect on index usage for this dataset (Section 5.2.1).

3. **Implementation using a search engine** We indexed the edges of the graph in ELASTICSEARCH by defining a mixed index in TITAN. To increase the value of the indexed data, we included with each edge the TITAN ID of the connected vertices[2]. Finally, we constructed the average degree centrality calculation as a single ELASTICSEARCH term query that searches for a given value over all edge labels, and aggregates the count of results based on the IDs of the vertices (either the source or the target vertex). The result returns an ordered list of vertices and the number of edges of which they are either the source or the target. This was implemented using the Java API of ELASTICSEARCH[3]. A prototypical implementation for in-degree calculation, similar to the one used in TITANLAB, is presented in Listing 7.1.

[1]Available here: http://tinkerpop.apache.org/docs/3.2.1-SNAPSHOT/recipes/#degree-centrality

[2]This design decision is, in fact, another use of denormalization. Albeit, a limited one and with less chances for inconsistency, as the IDs of connected vertices do not change

[3]Available here: https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/index.html

Listing 7.1: Prototypical degree centrality calculation for all vertices, using the Java API of Elasticsearch

```
TermsBuilder termsBuilder= AggregationBuilders
            .terms(aggregationName)
            .field("idOfTargetVertex")
            .size(numberOfVertices);
XContentBuilder contentBuilder;

contentBuilder = JsonXContent.contentBuilder().startObject();
termsBuilder.toXContent(contentBuilder, ToXContent.EMPTY_PARAMS);
contentBuilder.endObject();

SearchRequestBuilder searchRequest= esClient
             .prepareSearch("titanESIndex")
             .setTypes("edges"));
             .setQuery(QueryBuilders.termQuery("edgeLabel","labelValue"))
             .setAggregations(contentBuilder);
response= searchRequest.execute().actionGet();
```

Note that in our implementation we perform a termQuery over the label of the edges. This would not be needed for single-relation graphs.

Please note additionally that in our study only the *basic dataset* was a multi-relational graph, thus the composite index implementation could only be evaluated there.

For documentation purposes we include in Chapter A the distribution of edges (in-degree and out-degree) of the two relations in our synthetic basic dataset. We also document the degree distribution of the *Openflights dataset*.

The edge distribution of the *Pokec* dataset has already been shown to follow a power law distribution [TZ12], in tune with the expectations of a social network to constitute a scale-free network. Roughly, this distribution goes from few of the 1.5M profiles having 9k friendship connections, to a big part having less than 10 friendship connections.

Figure 7.1 shows the observed response time for the alternatives on the *Openflights dataset*. For the in-degree centrality, the speedup of the search engine method over the basic implementation was of 132x. The speedup for the out-degree relation was of 92x. In both cases these speedups translate into answering in less than one second a query that otherwise would take more than three seconds.

Figure 7.2 collects the observed response time in our experiments using the *basic dataset*. Across both relations in this dataset, the basic implementation lasted on average 37.1 minutes to give the results. In a notable improvement, the implementation that relied on composite indexes took on average 4.1 minutes to answer the query. On the other hand, with even larger performance gains, the implementation based on search engine functions took on average 2 seconds to answer the query. In the worst case observed,

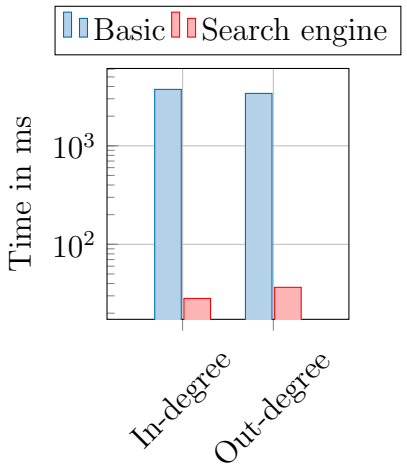


Figure 7.1: Response time of alternative implementations for calculating the average degree centrality on the Openflights dataset.

the search engine based implementation took 7 seconds to answer the query. Overall, the recorded speedups of the search engine implementation were from 300 to 5700x over the basic implementation, and from 50-500x over the composite index implementation. Composite indexes, on their part were from 6.7-13.5x faster than the basic implementation.

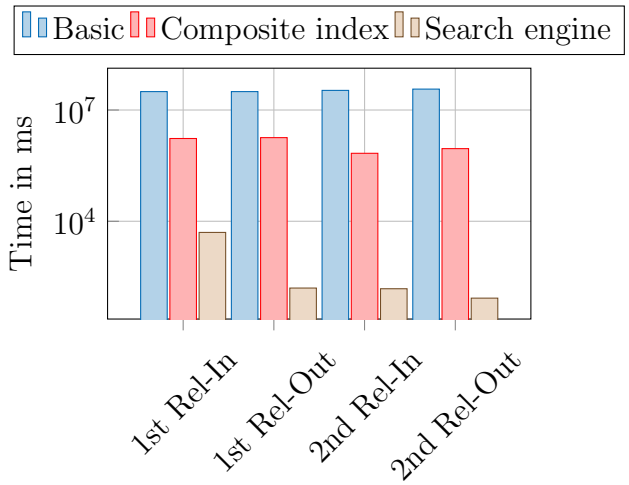


Figure 7.2: Response time of alternative implementations for calculating the average degree centrality over different relations on the basic dataset.

Figure 7.3 displays the response times we observed for the *Pokec* dataset. This was the largest dataset from our study, and the longest to load into Apache Titan (about 2 days for loading the 30M edges)[4]. Futhermore, this proved to be a heap intensive

[4]However, a note should be made that in the current version of TitanLab the loading of data was implemented in a naive single-threaded way. Surely this could be improved upon.

case for ELASTICSEARCH, with the program giving several heap errors and timeouts for nodes during the execution of the test. In spite of this, the test returned successfully the results on all occasions. The basic implementation took about 70 minutes to calculate the average degree centrality for all vertices, the search-engine version took about 28 seconds for the same tasks. These results amount to speedups of 150x from using search engine functionality.

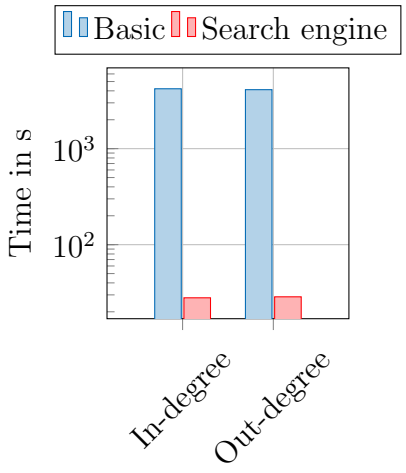


Figure 7.3: Response time of alternative implementations for calculating the average degree centrality on the Pokec dataset.

Our results clearly show that search engine integration can provide important gains to graph analysis tasks. In our study using the *basic dataset* this alternative reached notable speedups of at least 50x when compared to the best implementation using composite indexes. We observed 150x gains when using the *Pokec* dataset, compared to a naive implementation based on full-table scans.

The fundamental reason for the good performance of our search engine implementation is the fact that it runs an entirely different algorithm from the GREMLIN traversal used in other versions. The GREMLIN traversal begins by querying a given set of vertices (either using a composite index or a full-table scan), for each vertex it counts the number of incoming edges with a specific label, at last it groups the results by the IDs of the vertices and returns these results. On the other hand, the ELASTICSEARCH implementation relies on a simpler representation of the same data: a set of edge "documents". These documents are further simplified for querying, by the use of LUCENE's inverted indexes[5] which help term-based search, such as our *entry query*, which asks for all edge "documents" having a given label. Inverted indexes speed up this type of query because they point from a term value to the document numbers having this value. Another available inverted index, covering the ID of the source vertex, can also be used by "joining" it with the document numbers from our *entry query*. At last, the

[5]More information on LUCENE's index structure available here:https://lucene.apache.org/core/5_1_0/core/org/apache/lucene/codecs/lucene50/package-summary.html

resulting collection of terms and document numbers can be aggregated by counting for each term the number of resulting edge "documents" on which this term appears as the ID of the source vertex. In this way, by using 2 inverted indexes over a collection with a single document type, ELASTICSEARCH speeds up the calculation of the average degree centrality for all vertices in a graph.

In spite of the observed benefits from pairing TITAN with a search engine, there are still significant limits to the search engine usability. The essential issue is the restricted integration with graph databases, as we have discussed in other sections (Section 4.1.2). Nowadays developers cannot fully use a graph query language over a search engine representation of graph data. Instead, to leverage search engines, developers must adapt search engine APIs (such as we did in Listing 7.1) to their needs. These APIs have concepts and tuning characteristics that differ from traditional graph processing tasks. Using these APIs for network analysis can lead to: a) *impedance mismatch*, b) inadequate performance optimization in the search engine, and c) overheads from application-level mapping of items between the different storages. Developers should consider these limitations before adopting search engines as an alternative to improve performance of graph processing tasks.

Search engines also have some tradeoffs that developers should consider. Among them: a) search engines could be an additional burden to the operational side of applications, constituting an extra storage system that requires monitoring, provisioning and maintenance; b) search engines have different consistency guarantees when compared to storage engines tuned for OLTP workloads (such as CASSANDRA). Relying on search engines in environments with fastly changing data could lead to frequent inconsistencies.

7.3 Best Practices

Following our experimental findings in this chapter, we are ready to propose some best practices related to TITAN's integration with search engines. Similar to other chapters, our suggested best practices are a direct answer to the evaluation question that guided this chapter (Section 7.1).

1. Consistently throughout our experiments, we have observed that the pairing of TITAN with a search engine offers valuable chances for improving the performance of graph applications. Nevertheless, we also found that the current integration between the TITAN programming interface and the search engines is currently lacking in support for elementary search engine queries (e.g., counting the number of results), thus we encountered the need to study the external usage of the search engine. For functions as basic as counting on the global filter step we found that by querying the search engine externally we could achieve speedups of at least 80x when compared with composite indexes on selectivities lesser than 1. For more complex functions, such as the calculation of the average degree centrality of vertices in a graph, we found speedups of 50-500x when compared with an alternative implementation relying on composite indexes. As a result,

our observations strongly suggest that there are important performance gains by utilizing the search engine externally from Titan. Thus, we advice developers to consider this as a possibly valuable option in their graph processing toolbox.

2. Our observations have also implications for future Titan development, firmly building a case for work in advancing the search engine integration.

3. In our experiments we did not evaluate the tradeoffs from search engine external usage. In spite of this oversight, we would like to point out some issues that developers should consider before commiting to employ an external search engine API in graph applications: the use of a separate API could lead to *impedance mismatch*, inefficient performance in the search engine, and overheads in the application-level mapping of items between the storages. Further drawbacks of pairing search engines with graph databases include the increased operational costs and the possible loss of consistency guarantees.

7.4 Summary

In this chapter we document the experimental results of a simple microbenchmark to evaluate some potential gains of using search engines for graph processing tasks. Unlike other chapters in this study where we have focused on elementary graph processing (e.g., global access or one hop traversals), here we considered a more general graph algorithm– the calculation of the average vertex degree centrality.

With this chapter we conclude our evaluation on some core issues that programmers must tackle for developing graph applications with Titan. In the next chapter we wrap-up our study by summarizing the best practices derived from our findings in Chapter 4, Chapter 5, Chapter 6 and Chapter 7; we also propose future work to advance the research initiative that we have undertaken in this project.

8. Conclusion

In this study we aimed to evaluate a selection of application design choices offered to developers of graph database applications. With the goal of providing immediate practical applicability of our results, we focused on a specific open-source graph database: Apache Titan, in its current version. To carry out our evaluations we developed TitanLab, a prototypical application that uses Titan as an embedded database. Among the application design choices we mainly studied global access alternatives, further considering issues such as sensitivity of composite indexes to cardinality, or how the ordering of subqueries in Titan might affect the performance of multi-index conjunctive predicate queries. Complementing these evaluations, we used TitanLab to gain experimental confirmation on the sensitivity of small traversals to differences in topology. Finally, we used TitanLab to illustrate, through practical examples, some potential gains of simple application optimization strategies: denormalization and leveraging search engine functionality for graph algorithms.

In this chapter we conclude our study, as follows:

- **Summary: Best practices for graph database applications** We summarize the best practices that we infer from our experimental studies (Section 8.1).
- **Threats to validity** We disclose some possible threats to the validity of our evaluations (Section 8.2).
- **Concluding Remarks** Brief notes wrapping-up this study (Section 8.3).
- **Future work** We complete this project by proposing future work that could expand on the experiences of our study (Section 8.4).

8.1 Summary: Best practices for graph database applications with Apache Titan

1. Titan *IDs are the best choice for retrieving items*, in contrast to accessing them by application-assigned unique identifiers. While this choice makes a small difference (i.e., achieving speedups of 1.6-10x) when comparing access for a single item, there are notable benefits when considering random access to multiple items: retrieving items by IDs reduces database-level read latencies and cache misses. In our evaluation this translated to a 50x faster performance for retrieving random items.

2. *Indexes are essential to support global filter queries and should be used always over full-table scans.* For this use case, indexes are a valuable solution to prevent costly full-table scans. The benefits of indexes grow with increasing selectivities. The same holds for small variations on global filter queries, such as only counting the results or sorting them. In our evaluation we observed indexes leading to significant speedups of at least 14, 50 and 1.4x for the basic case, counting and sorting, respectively. While the 1.4x speedup might seem small, this represented a function that returned 3 minutes faster from a long 10 minutes full-table scan with sorting.

3. *Each type of global index has its strengths and weaknesses.*

   - Composite indexes are the native indexing approach in Titan, thus they have the benefit of a special integration with Titan's main query engine. Nonetheless, their usability for conjunctive predicate queries has certain limitations.
   - Mixed indexes are Titan's solution to enable full-text queries. They have the additional benefit of supporting well ad-hoc conjunctive predicate queries. Even though the current integration of Titan with these indexes offers limited functionality (most notably, missing counting, aggregations and sorting), developers can use these indexes with external APIs, reaping further benefits from them. The main limitation of these indexes is that they are not integrated with Titan's main query engine, thus query results cannot be immediately used for traversals. Furthermore, search engines can have additional operational costs and weaker consistency guarantees, which might deter their adoption.
   - There is an additional global indexing alternative available to developers, which is the usage of the graph as an index. This can be accomplised with special meta vertices pointing to a given set of vertices, thus speeding access to them. This approach might have the benefit of enabling users to express very complex and selective relations, which might be less efficient to retrieve by other methods. However, in our studies we found no significant gain from such approach for simple predicates and basic conjunctive predicate global

filter queries. We also observed a notable increase in heap usage by this alternative.

- The global indexing alternatives might differ in their stability to random access. Before selecting an index, we advice developers to test how this index responds to the access patterns in the use cases of their deployed applications.

4. *Among all the other alternatives, composite indexes are the best solution to support basic single-predicate global filter queries.* This is also the case when the query is combined with pagination using large pages (i.e, 10k vertices/page, in our observations).

5. *Although cardinality does not have a notable effect on the response time of composite indexes for basic single-predicate global filter queries in small datasets, it remains to be checked its impact on large datasets.* Our experiments show that in small datasets (of less than 3.1 million vertices), cardinality doesn't affect notably the response time of global filter queries. When using Titan with Cassandra, developers should be aware that the Cassandra community warns against using secondary indexes for extreme cardinality values. Considering that Titan relies on Cassandra for the composite indexes, the advice from the community might be relevant to Titan application developers.

6. *Ad-hoc conjunctive predicate global filter queries require special consideration.* Our results show that the best solution for conjunctive predicate global filter queries is a composite index over all properties. However, this might not be a feasible solution for ad-hoc queries when the specific combination of properties is not known in advance and indexes covering all properties cannot be guaranteed to exist. In consideration, other approaches have to be selected, such as using multiple composite indexes, using mixed indexes or the graph as an index. We advice developers to study the tradeoffs of the alternatives according to the requirements of their application.

    - *The naive option of supporting conjunctive predicate queries with multiple composite indexes is not efficient.* In the current version of Titan, the performance of this option might be further detriorated by mistakes in the query processors ordering of subqueries. Our observations show that other alternatives can be from 2 to 22 times faster if there are no mistakes in the subquery ordering, and from 4 to 275 times faster when the query processor makes mistakes in the subquery ordering.
    - *Mixed indexes are a good solution but developers should consider their limitations.* In our evaluations mixed indexes displayed performance close to that of combined composite indexes. There is a possibility that finely tuned queries to mixed indexes use could lead to better performance than that of a combined composite index.

- *The graph as an index alternative might be useful for ad-hoc conjunctive predicate queries when there is no combined composite index available, but it was not covered by our research.*

7. *Small functional variations on the basic single-predicate global filter query (i.e., pagination, counting, sorting and more complex predicates) might be better supported with indexing solutions other than composite indexes.* In our experiments, mixed indexes, specially when used externally, showed to be a good solution across these cases. The graph as an index approach did not prove to be better than the composite alternative in our studies.

8. *Traversal times are sensitive to the topology.* In our studies we validated the expectation that the topology of the data affects the performance of the traversal in Titan. It should be noted though, that our experiments were done on a very small dataset, with synthetic data that did not represent any specific real-world network. As a corollary from this observation, we suggest developers to study if their schema modelling (i.e., the design of what to model as vertices, edges and properties) could be tuned to create topologies that improve traversals. We further suggest developers to consider the use of strategies such as denormalization. Although not studied here, the use of Gremlin's traversal strategies seems to be a nother good way to improve traversals.

9. *Denormalization could be a useful strategy for improving traversals.* In our study we illustrate a potential gain from denormalization on a synthetic super node from a small dataset (450k vertices), observing a performance gain of 34x, amounting to making a function that took 2 minutes return in 3 seconds. This method also includes tradeoffs not evaluated here, such as increases in memory footprint, reductions in write throughput, the higher potential for data inconsistency, among others. In fact, it is likely that on large datasets, there could be cases where the increased memory use of denormalization could lead to worse performance than the basic alternative. Thus, we urge developers to evaluate the tradeoffs for denormalization in their use cases before adopting it.

10. *Leveraging search engine functionality seems to be a promising approach to improving network analysis tasks with graph databases.* In our studies we illustrate this case with the simple task of calculating the average degree centrality for all vertices in a graph. Leveraging search engines leads to significant performance gains of 100x on the Openflights dataset, 48-505x on a slightly larger synthetic dataset and on the Pokec dataset. In the best scenario, search engines contributed to make 4 minute calls return in less than 2 seconds. Nonetheless, there are important limitations to the leveraging of search engine functionality. Apart from the same issues with mixed index integration to Titan, we can mention *impedance mismatch*, inefficient performance in the search engine (considering that its optimizations might not adapt well to graph access patterns), and overheads in the application-level mapping of items between the storages.

This list is not an exhaustive one by any means. Important aspects of using TITAN, such as its integration with HADOOP and other distributed processing frameworks, traversal strategies, support for temporal features, multi-tenancy, improved graph algorithms, special configurations (e.g., adding transactionally-bounded caches to long-running queries) and operational concerns, were left out of our study.

8.2 Threats to validity

In this section we disclose some items which we consider could be threats to the validity of our evaluations.

- *Possible design bias the microbenchmarks presented in Section 6.2.2 and Section 7.2*: We would like to point out that in the design of our microbenchmarks, there is a different treatment for the evaluation questions 7-8. Namely, the corresponding microbenchmarks do not examine the optimization strategies for a neutral function, instead for both questions a specific use case is given in the microbenchmark where the optimization strategies might be applicable. For this reason these are more illustrative examples than proper evaluations. More impartially designed tests could be valuable to understand the disadvantages of the considered optimization strategies when they do not fit the use case. Furthermore, the deterimental aspects of these strategies need to be studied more comprehensively.

- *Concerns about ID projection in global filter query*: Our decision to project over the ID of items when testing the global filter query, might fail to represent the actual performance when retrieving the whole vertex. This second case might display a lower gap between the performances of the full table scan and the indexing alternatives.

- *Missing OLAP engines*: The developers of TITAN recommend the use of OLAP engines, with distributed processing, to speed long-running queries. By not considering this, our evaluations might not portray to the fullest the real-world performance that TITAN users percieve.

- *Limitations in product versions*: The use of the current version of TITAN might not be representative of the database as a whole, given that some functionality (e.g. the integration with search engines) might still be improved upon by the community. Furthermore, the use of versions 1.5.2 of ELASTICSEARCH and 2.1.11 of CASSANDRA (the most recent verstions officially supported by TITAN), fails to portray improvements that these storage solutions have included in newer versions.

- *Misrepresentation of* ELASTICSEARCH *query potentials*: Our decision to use SimpleQueryStringQueries in ELASTICSEARCH, instead of more specialized query types (such as a Term Query), does not display to the fullest the performance gains available with ELASTICSEARCH. This is pertinent to the performance of the mixed

index external alternative in the global filter query microbenchmark (Section 4.3.2 and the index selection case (Section 5.2.2). In a short test we estimate that by using a TermQuery the performance could've been from 10 to 100x better for the alternative than those reported in (Section 5.2.2). However, deeming that this might give an overspecialization to one alternative over another, we limited our study to a SimpleQueryStringQuery.

8.3 Concluding Remarks

In this project we aimed to consider application-level choices available to developers for improving the performance of graph database applications. Instead of comparing the alternatives for different databases, we focused on a specific graph database: Apache Titan.

To examine these alternatives we proposed a set of research questions considering index support for global access to items. While this might be a highly limited focus, global access can be the entrypoint to traversals and thus we consider that improving this can have a big impact on the performance of applications.

We complemented our inquiry with questions about the sensitivity of traversals to topology, and the potential gains from optimization strategies like leveraging search engines and denormalization. This last set of questions was less evaluative and more illustrative in nature.

To tackle the evaluation questions we created TitanLab, a prototype of graph database applications using Titan as an embedded database, with the support of Elasticsearch and Cassandra as backends. Within TitanLab we implemented different functionality to enable the execution of microbenchmarks carefully designed to answer our evaluation questions.

The findings of our experiments using these microbenchmarks are included in Chapter 4, Chapter 5, Chapter 6 and Chapter 7. From these findings we concluded a set of best practices for developers, summarized in Section 8.1.

The results of this work are likely to have a practical immediate relevance for application developers that use Titan.

Regarding other graph databases, there might be no direct transfer of the best practices we propose. However, our core evaluation questions and the approach that we present for microbenchmarking graph database applications could be valid solutions to determine best practices for other graph database offerings. Specifically, for applications using Gremlin as a query language.

Our experimental results could also have some relevance for work on evolving the specific systems that we used in our evaluations.

Concerning these systems, our experimental observations suggest that improving the integration between graph databases and search engines might be a valuable endeavour. Bettering the sub-optimal index selection studied in Section 5.2.2 might also be of value.

We would like to conclude by emphasizing that the focus of this work has not been directly about improving graph databases, instead we've aimed to contribute to the understanding of how developers can work efficiently with these databases. Concretely, we addressed how developers can face the problem of finding good ways to traverse an existing graph database. While this might be a less pertinent issue in today's mature relational database products -which rely on query optimizers and runtime adaptive improvements-, it is a pressing matter for graph databases, where standards for query languages and query optimizations are still evolving, and small developer choices (e.g., which index to use for a given query) can have enormous impact on the performance of applications.

Looking back at the evolution of databases from the network model into the relational model, it becomes evident that the standardization of query processing with the relational model was a big step forward that enabled vendors to focus on building finely-tuned adaptive database products, while developers could concentrate on writing their applications using a high-level language.

Looking forward, we can reasonably imagine that the standardization of graph query languages, their underlying models and the processing frameworks in charge of running the graph queries, will have an impactful, positive effect on the evolution of graph databases and the applications that can be built with them.

This standardization, though, might only take place in the midst of communities actively involved in working with existing systems and striving to create best practices in the process. The standardization of graph technologies might furthermore depend on connecting practical experience, the understanding of the queries needed, with deep technical knowledge about the graph database offerings. Standardization might depend on connecting the perspective of the developer with that of the query optimizer and the database expert.

In this context we suggest that work on microbenchmarking application-level query optimizations might nowadays be specially benefitial in helping the standardization process. Microbenchmarking this kind of optimizations provides an opportunity to connect the developer perspective with that of the database expert. In our study we've shown that this connection of perspectives might produce some valuable knowlege, like knowing that leveraging search engines can turn a 70 minutes query into a 28 seconds one. Or knowing that upon the circumstance of hopping from one node to another, filtering on denormalized properties can turn a two minutes process into one that runs in less than a tenth of a second. As with most things in life, it's that little hop at the end, which can make a world of difference.

8.4 Future Work

- **Evaluation of further functionality:** In our experiments we only considered basic query functionality. We can propose that future work following our microbenchmarking approach might benefit from addressing a broader set of queries, complexity of query predicates and graph algorithms. Other ideal tasks to microbenchmark could be data loading, data integration processes and the distinctions between projections and whole vertex retrieval. Furthermore, configuration aspects of the databases (e.g. the use of database caches during a transaction) were notably missing from our study. Our understanding of the performance tradeoffs in these technologies could benefit from considering configuration choices.

- **Study on how the best practices might be applied for standard benchmark queries:** Another area of future work could be the evaluation of how our proposed best practices could be applied to standard sets of graph queries, such as those included for benchmarks of large-scale graph analysis in [CHI+15].

- **Microbenchmarking other databases:** TitanLab could be extended to become a more platform-agnostic prototype. In this way it could be employed for microbenchmarking application-level optimizations of any graph database supporting Gremlin and the TinkerPop framework.

- **Contributions to the systems used in our evaluation:** Our experimental results and practical observations sugest that there might be room for contributing to Titan's codebase in different aspects. One aspect worthy of consideration could be the creation of indexes on-the-fly. Currently Titan supports index creation during runtime, but this was not evaluated in our studies. Plausibly a framework could be proposed to estimate when creating an index might be valuable, according to its usefulness and cost, characteristics of the data (e.g. selectivity) and the expected queries. Another worthy aspect could be index selection. While our studies pointed out that index selection could benefit from choosing first the subquery with the lowest selectivity (i.e., the one that filters out the most items) Section 5.2.2, some development work and experiments are still needed to determine how common selectivity estimation tools could be integrated into the overall workflow of Titan.

- **Research on further leveraging the multi-store nature of some graph databases:** Among the considerations in our work, the integration of graph databases emerged as an interesting line of inquiry. As in other systems, the overcoming of the impedance mismatch between the multi-store systems remains a significant challenge.

- **Maturity of best practices into design patterns:** Best practices for using a system, such as the ones we outlined, could mature into useful design patterns that could ease development and the transfer of knowledge between graph database applications.

A. Degree Centrality Distributions for Vertices in the Openflights and Basic datasets

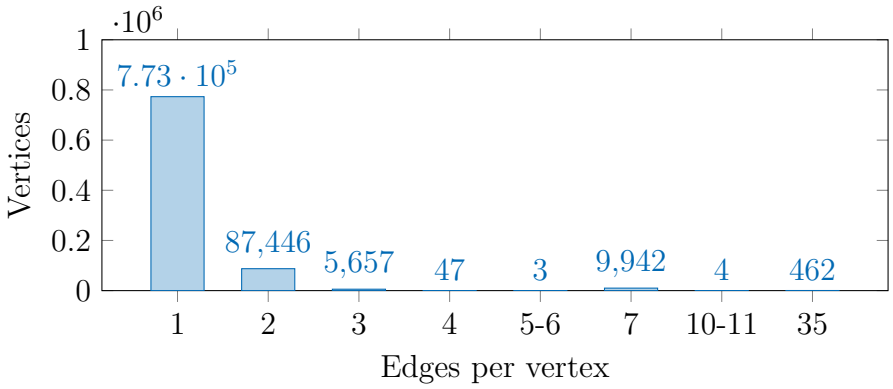


Figure A.1: Degree centrality distribution for vertices in the basic dataset, first relation, in-degree

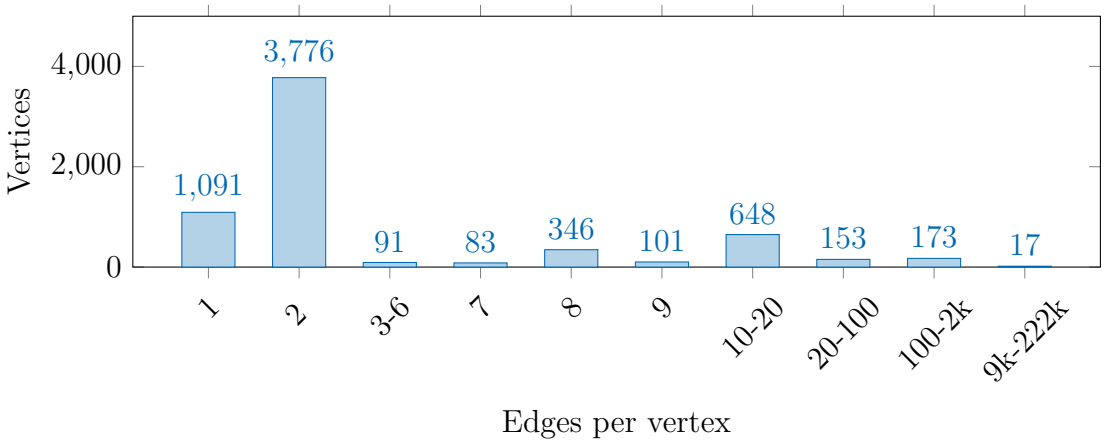


Figure A.2: Degree centrality distribution for vertices in the basic dataset, first relation, out-degree

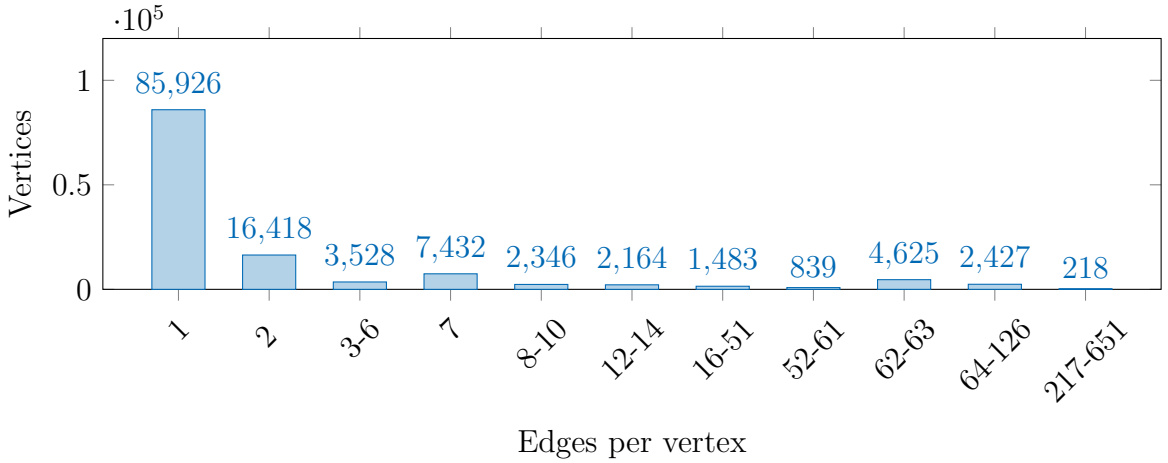


Figure A.3: Degree centrality distribution for vertices in the basic dataset, second relation, in-degree

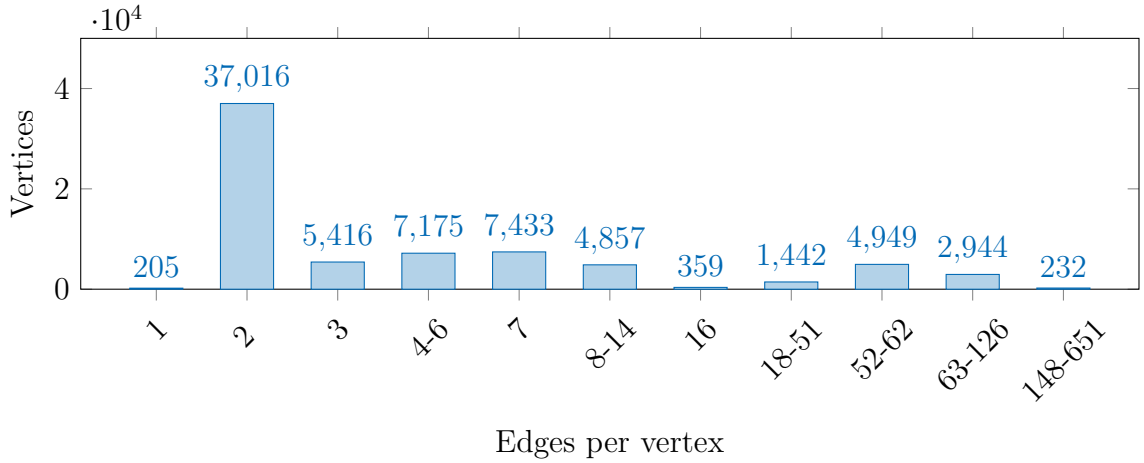


Figure A.4: Degree centrality distribution for vertices in the basic dataset, second relation, out-degree

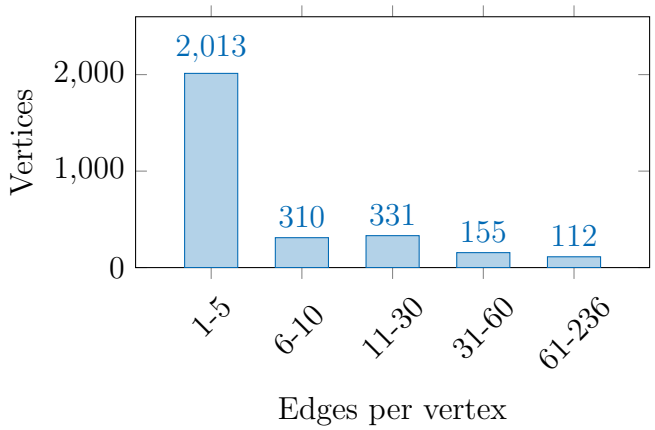


Figure A.5: Degree centrality distribution for vertices in the Openflights dataset, in-degree

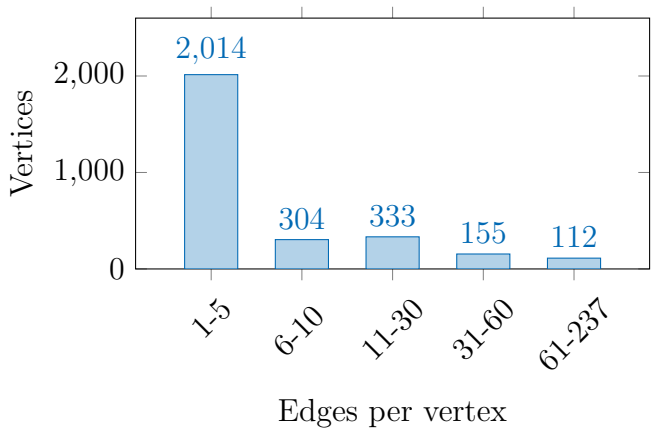


Figure A.6: Degree centrality distribution for vertices in the Openflights dataset, out-degree

Bibliography

[AAB+16] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter, and Domagoj Vrgoc. Foundations of modern graph query languages. *arXiv preprint arXiv:1610.06264*, 2016. (cited on Page 1 and 8)

[AG08] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008. (cited on Page 7)

[AW+10] Charu C Aggarwal, Haixun Wang, et al. *Managing and mining graph data*, volume 40. Springer, 2010. (cited on Page 7)

[BPK15] Sotirios Beis, Symeon Papadopoulos, and Yiannis Kompatsiaris. Benchmarking graph databases on the problem of community detection. In *New Trends in Database and Information Systems II*, pages 3–14. Springer, 2015. (cited on Page 16)

[CF12] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: laws, tools, and case studies. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 7(1):1–207, 2012. (cited on Page 7)

[CHI+15] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the GRADES'15*, page 7. ACM, 2015. (cited on Page 70)

[CK04] Martin Campbell-Kelly. *From airline reservations to Sonic the Hedgehog: a history of the software industry*. MIT press, 2004. (cited on Page 4)

[CMS13] Kenneth Cukier and Viktor Mayer-Schoenberger. Rise of big data: How it's changing the way we think about the world, the. *Foreign Aff.*, 92:28, 2013. (cited on Page 5)

[Gub15] Andrey Gubichev. *Query Processing and Optimization in Graph Databases*. PhD thesis, München, Technische Universität München, Diss., 2015, 2015. (cited on Page 9)

[HFC+00] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 23(2):7–18, 2000. (cited on Page 4)

[HPL12] HP Labs-business-value white paper: Big data. www.hp.com/hpinfo/.../press.../IO_Whitepaper_Harness_the_Power_of_Big_Data.pdf, 2012. Accessed: 2016-11-30. (cited on Page 5)

[IBM] IBM what is big data? https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html. Accessed: 2016-11-30. (cited on Page 5)

[JRW+14] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. Vertexica: your relational friend for graph analytics! *Proceedings of the VLDB Endowment*, 7(13):1669–1672, 2014. (cited on Page 7)

[MAB+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010. (cited on Page 8)

[MBGVEC11] Norbert Martinez-Bazan, Sergio Gomez-Villamor, and Francesc Escale-Claveras. Dex: A high-performance graph database management system. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 124–127. IEEE, 2011. (cited on Page 8)

[Mis14] Vivek Mishra. *Beginning Apache Cassandra Development*. Apress, 2014. (cited on Page 12 and 42)

[MMI+13] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013. (cited on Page 8)

[MW04] Merriam-Webster. *Merriam-Webster's collegiate dictionary*. Merriam-Webster, 2004. (cited on Page 5)

[Ops11] Tore Opsahl. Why anchorage is not (that) important: Binary ties and sample selection. *online] http://toreopsahl. com/2011/08/12/why-anchorage-is-not-that-important-binary-tiesand-sample-selection (accessed September 2013)*, 2011. (cited on Page 17 and 20)

[Pin16] Marcus Pinnecke. Efficient single step traversals in main-memory graph-shaped data. 2016. (cited on Page 9)

[RN10] Marko A Rodriguez and Peter Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010. (cited on Page 6)

[SWL13] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM, 2013. (cited on Page 8)

[Tes13] Claudio Tesoriero. *Getting Started with OrientDB*. Packt Publishing Ltd, 2013. (cited on Page 8)

[TGRM14] Vernon Turner, John F Gantz, David Reinsel, and Stephen Minton. The digital universe of opportunities: rich data and the increasing value of the internet of things. *IDC Analyze the Future*, 2014. (cited on Page 5)

[TZ12] Lubos Takac and Michal Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, pages 1–6, 2012. (cited on Page 17, 20, and 58)

[Web12] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 217–218. ACM, 2012. (cited on Page 8)

[Wie15] Lena Wiese. *Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases*. Walter de Gruyter GmbH & Co KG, 2015. (cited on Page 6)

[WRW+13] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2013. (cited on Page 1 and 7)

[XGFS13] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013. (cited on Page 8)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 6 Jan, 2017.