Philipps-Universität Marburg

Faculty of Mathematics and Computer Science

A Graph-Based Query Language For Breakpoints

Freya Dorn

Master Thesis

May 27, 2024

Supervisor:
Prof. Dr.-Ing. C. Bockisch
Programming Languages and Tools Group

Advisor:
M.Sc. Teresa Dreyer
Programming Languages and Tools Group

A Graph-Based Query Language For Breakpoints

Institution

Philipps-Universität Marburg
Faculty of Mathematics and Computer Science
Programming Languages and Tools Group
Hans-Meerwein-Str. 6
35043 Marburg
Deutschland

About the author

Freya Dorn is a computer science graduate student at the Philipps University Marburg. Her interests are linguistics, programming language design, systems engineering, and bioinformatics. You can reach her under freya.dorn@fastmail.com.

License

Classification (ACM CCS 2012)

- **Applied computing** → **Education**; **Document preparation**;
- **Software and its engineering** → *Software notations and tools*;

Keywords

Breakpoints, Debugging, Graph Databases, Graph-Based Query Languages, Temporal Graphs

Acknowledgment

I would like to thank my advisor Teresa Dreyer for her extensive support and encouragement. Her valuable feedback helped me greatly to overcome the many challenges I encountered while finishing this thesis.

I would also like to thank Elena for her deep consideration and love. You're the best sister I could ask for.

I don't recall a change in plans being in the original plan!

(Dale Gribble, King of the Hill, Season 6, Episode 2)

Abstract

Breakpoints are a crucial tool to debug programs. In order to explore the possibility of a unified powerful breakpoint language, we developed a graph-based breakpoint backend for an existing trace-based debugging framework and adapted the Cypher query language for the specification of breakpoints.

We evaluated whether the mature general-purpose Neo4j graph database and Cypher query language are a good fit for our breakpoint backend. Furthermore, we extended the graph model with temporal annotations and improved the support for modelling concurrency in the execution stream.

We found the Cypher query language a good fit for powerful breakpoints, even though the query engine provided by Neo4j was ultimately found lacking in terms of its latency and ability to efficiently support the temporal and concurrency constraints necessary for more advanced queries.

The use of the Neo4j graph database backend proved successful, with acceptable performance overall and great potential for future extensions due to the large ecosystem of available tools and abstractions supported by the graph database.

Zusammenfassung

Haltepunkte (Breakpoints) sind ein essenzielles Werkzeug zum Debuggen von Programmen. Zur Erforschung des Potenzials einer einheitlichen, leistungsstarken Haltepunktsprache haben wurde ein graph-basiertes Backend für ein bestehendes trace-basiertes Debugging-Framework entwickelt und die Abfragesprache Cypher für die Spezifikation von Haltepunkten angepasst.

Es wurde evaluiert, ob die etablierte Allzweck-Graphdatenbank Neo4j und die Abfragesprache Cypher gut für das Haltepunkt-Backend geeignet sind. Darüber hinaus wurde das Graphenmodell um zeitliche Annotationen erweitert und die Unterstützung für die Modellierung von Nebenläufigkeit im Ausführungsstrom verbessert.

Die Abfragesprache Cypher stellte sich als gut für leistungsstarke Haltepunkte heraus, auch wenn die von Neo4j bereitgestellte Abfrage-Engine sich letztlich als unzureichend erwies, was die Latenzzeit und die Fähigkeit zur effizienten Unterstützung der zeitlichen und nebenläufigen Anforderungen betrifft, die für mächtigere Abfragen erforderlich sind.

Die Verwendung des Neo4j-Graphdatenbank-Backends erwies sich als erfolgreich, mit insgesamt akzeptabler Performanz und großem Potenzial für zukünftige Erweiterungen aufgrund des großen Ökosystems verfügbarer Werkzeuge und Abstraktionen, die von der Graphdatenbank unterstützt werden.

Contents

1 Introduction

Breakpoints are user-defined moments during the execution of a program when the execution should be suspended. They are commonly used in software development to inspect the state of a live program during critical moments of its execution to facilitate debugging and program analysis. Developers typically specify breakpoints through their development environment, most commonly by selecting lines of source code or by setting value conditions for a variable such that if the execution reaches the selected line of code or the value condition becomes true, the program is suspended.

Different development environments and debugging tools typically have their own idiosyncratic breakpoint capabilities and interfaces. Because capabilities are defined in an ad hoc manner, breakpoint conditions often lack composition (eg. "*only stop when line 7 is reached* ***and*** *x is greater than 5*"). There are no established standard specifications or encodings for breakpoints that could be shared between different development tools or that could be used for additional analysis. This thesis will explore a new approach for breakpoints based on graph query languages and graph databases.

Our general ontology of breakpoints was introduced by [Boc+21]. In order to abstract broadly over the wide range of languages and runtimes in use today, we model execution as a simplified event stream of all possible suspension points during execution that are of interest to the developer (eg. method[1] calls or variable assignments). Additionally, events have attributes that represent the static and dynamic properties of the execution at that point (eg. source code position, method name, dynamic value of a variable).

Breakpoints are then modelled as search queries over this event stream that match specific events. If an event matches the query, execution should be suspended for the corresponding event. Queries may refer both to the current event in the stream, but may also refer back to past events. They may also be conditioned on the event's type and any of its attributes. Conditions and queries may also be logically composed into complex queries.

[Boc+21]'s ontology defines seven base events that sufficiently powerful breakpoints should support, summarized in table 1 (cf. [Dor21]).

There are two ways to evaluate a given breakpoint query. When live debugging, the query can attempt to match against the latest event during execution. If the query matches, the program is suspended and the may be investigated further by the programmer. This represents the typical debugging experience where the developer wants to inspect the program state immediately as soon as a specific condition is fulfilled. Because this match is attempted with every new event, we call this mode incremental search. Incremental search is essential for live debugging and particularly sensitive to the performance overhead of the breakpoint backend.

[1] Because our implementation is based on the Java programming language, we will restrict our analysis to methods and treat functions as equivalent constructs. Further distinctions could be made for other languages with any significant changes to our approach.

Table 1 Base Events used by [Boc+21]

kind	parameters
line	file name, line number
change	field or local variable
read	field or local variable
method enter	method
method exit	method
class load	class
exception	type

Alternatively, queries may perform a global search over the whole event stream to find all possible breakpoints of an execution trace. By using a complete execution trace of a previous program run, global search may be used to identify particularly interesting points to be recreated and inspected by the debugger.

Additionally, if a time-travelling debugger[2] is used, execution may then be continued from any of those points. However, we will focus primarily on incremental search as it is both more common and much more performance-sensitive than global search.

[Boc+21]'s original approach modelled the event stream in XML and implemented queries in XQuery[Cha02]. While conceptually straightforward and easy to implement, this approach led to poor runtime performance and a large memory overhead. In order to properly support incremental search, Bockisch et al. also had to implement a new XQuery processor that could handle incomplete XML documents. Overall, the translation to XQuery led to a leaky abstraction that, while sufficiently powerful to support the free composition of breakpoints, did not fit the performance demands of incremental search.

[Dor21] improved on those weaknesses by designing and implementing a custom domain-specific language for breakpoint queries with its own search backend. Queries are defined in a language that closely mirrors the source code used to write the program and then compiled to an efficient automaton to be evaluated against the current event during execution.

Even though [Dor21]'s query language would be fairly intuitive for programmers used to the Java programming language used in the implementation of its backend, its syntax would not transfer directly to other programming languages and alternative dialects might have to be developed if the same ergonomic fit was desired. Additionally, the custom backend did not support redefining or changing breakpoints during execution and had no support for global searches.

[2] A time-travelling debugger allows the programmer to rewind the execution of a program to a previous state for further investigation, often with the ability to continue execution from that point instead of the current one.

Both approaches were forced to implement their own backend to achieve acceptable performance. This dramatically increases the implementational burden for development environments wishing to support breakpoints. Adoption and implementation quality would be helped greatly by the use of a mature search backend. Additionally, neither formalization supports concurrent execution.

[Dre23] explored a different formalization using temporal logic. This approach fits event streams more naturally and may be extended in the future to support concurrency. Temporal logic uses only a small number of powerful operators, simplifying implementation efforts. However, the temporal logics analyzed by Dreyer cannot easily model global search or the more complex composition of queries. Furthermore, no implementation was provided to explore the resulting performance and possible implementation difficulties and there is no good candidate for an established search backend for temporal logic.

1.1 Research Question

The main focus of this thesis is to explore a new modelling approach for breakpoint specifications using a graph database and graph query language.

Our primary question is whether an implementation using the mature general-purpose graph database Neo4j and its accompanying Cypher query language are a practical solution for a breakpoint backend and for breakpoint specification.

Previous modelling approaches suffered from poor performance, limited expressiveness of the query language and a lack of support for concurrency. Graph databases are designed to handle complex queries and very large datasets efficiently and ergonomically, and therefore seem like a promising solution for our concerns.

Furthermore, this thesis evaluates whether a mature graph database and graph representation of the event stream are suitable for modelling breakpoints. A practical graph-based approach needs to support fast incremental search and a compact graph representation to reduce the runtime and memory overhead of the debugger, so we also perform an appropriate benchmark in line with [Dor21] to evaluate whether our approach is suitable for live debugging.

Furthermore, temporal graphs promise a very useful extension to graph databases that can capture the time component of execution. An important focus of our work is to explore a new use case for temporal graphs. We will leverage temporal annotations to model the event stream and extend the capabilities of the supported breakpoint queries.

Breakpoint queries will be represented in the widely-used Cypher graph query language. That way we can leverage existing technologies and formalizations while retaining the ability to represent complex queries with sophisticated composition. Additionally, our approach also explores the suitability for modelling of concurrent execution.

We will first provide an overview of graph databases and graph query languages, including temporal graphs (‹Graph Databases› (section 2)). We will establish our detailed requirements for powerful breakpoints (‹Breakpoint Requirements› (section 3)). Next, we will discuss how to model the event stream as a graph (‹Modelling the Event Stream› (section 4.1)) and how to represent breakpoints in the Cypher query language (‹Modelling Breakpoint Queries› (section 4.2)). After that, we will present crucial aspects of our specific implementation (‹Implementation› (section 4)), evaluate it against our requirements and compare it to the previous approaches mentioned earlier (‹Evaluation› (section 5)). Finally, we will conclude this thesis with an extensive overview of related works (‹Related Works› (section 6)).

2 Graph Databases

Graph databases are used in a wide range of domains that process large datasets of rich and diverse data. In particular, they are designed for data that follows an overall graph structure, such as social networks, transportation networks or biological interactions. These datasets contain entities we represent with nodes, and relationships between these nodes we represent with edges in the graph.

We will now specify more precisely what kind of graphs are used in graph databases. Then we will discuss the databases themselves and ways to query and analyze the data within them.

2.1 Graphs

First, we define[3] a basic graph and then derive two more useful types of graph from it to model more sophisticated metadata.

Definition 2.1 (Basic Graph)**.** A graph G is a pair (V, E), where:

1. V is a finite set of nodes (or vertices)
2. E is a finite set of edges, where $e \in E$ is a tuple (a, b) with $a, b \in V$

While a basic graph can certainly represent complex relationships between entities, we usually want to store additional metadata. One important limitation of a basic graph is that we cannot distinguish different kinds of relationships.

Consider the following simple social graph (figure 1) with three people in it. Hank is friends with Dale and works for Buck, so we add two edges to represent these relationships.

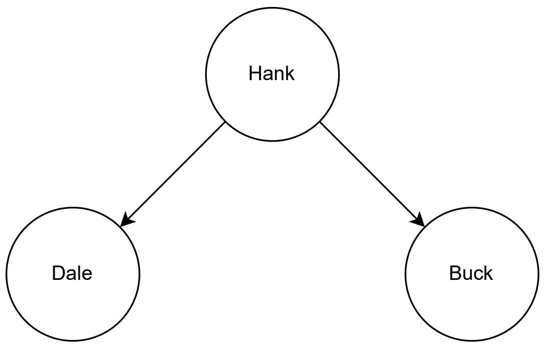
However, there is no way for us to just find Hank's friends. To overcome this limitation, we add labels to the edges and define the edge-labelled graph.

Definition 2.2 (Edge-labelled Graph)**.** An edge-labelled graph G is a tuple (V, E, L), where:

1. V is a finite set of nodes (or vertices)
2. E is a finite set of edges, where $e \in E$ is a tuple (a, b, l) with $a, b \in V$ and $l \in L$[4]
3. L is a finite set of labels

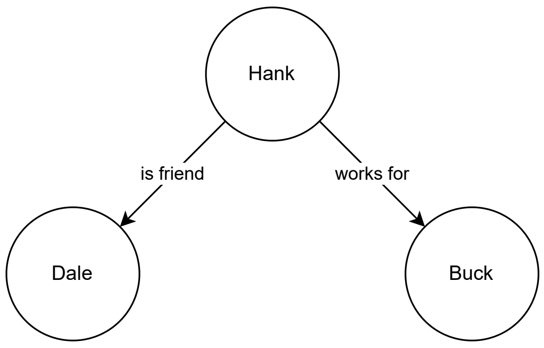
[3] The following definitions broadly follow [Ang+17] and [Bes+23], with minor adjustments for consistency with the rest of this thesis.
[4] We restrict edges to only one label per edge, as almost all graph databases enforce this restriction (cf. [Ang+17]).

Figure 1 Basic Social Graph



We can now extend our previous social graph to distinguish between friendships and work relationships (figure 2).

Figure 2 Edge-Labelled Social Graph

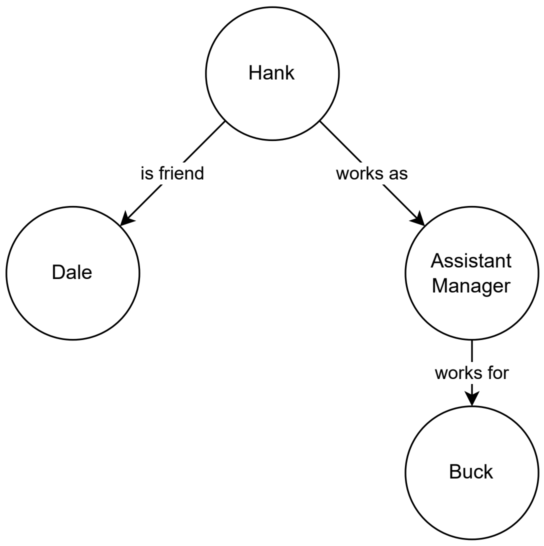


Edge-labelled graphs are the basis of the popular Resource Description Framework (RDF) standard[GS04]. In RDF, a graph is at its core a set of tuples *(subject, predicate, object)* that represent the edges between two entities (subject and object) and an identifier (predicate) for the type of relationship. There are further restrictions on the exact form of entities, but those are of no concern to us in this overview.

RDF has been adopted by a wide range of applications, in particular knowledge graphs like Wikidata[Wik24] and collaborative web databases such as MusicBrainz[Met24]. While RDF is capable of modelling very complex relationships, it is cumbersome to store metadata about relationships in a compact way.

For example, if we also want to represent that Hank's job title is that of an assistant manager, we have no easy way of doing so. We could add a new tuple *(Hank, Job Title, Assistant Manager)*, but that would not be connected to the specific employment by Buck and we couldn't distinguish between multiple simultaneous jobs Hank might have. To solve our problem, we would have to add new nodes to group this information together and restructure our graph (figure 3).

Figure 3 Edge-Labelled Social Graph with Attribute Nodes



Our example demonstrates the difficulty of adding new metadata to an edge-labelled graph and how doing so will often introduce new dummy nodes or reorganize existing nodes dramatically. To overcome this limitation, we extend our previous definition further, introducing property graphs that allow nodes and edges to directly contain a set of properties indexed by keys.

Definition 2.3 (Property Graph)**.** A property graph G is a tuple (V, E, L), where:

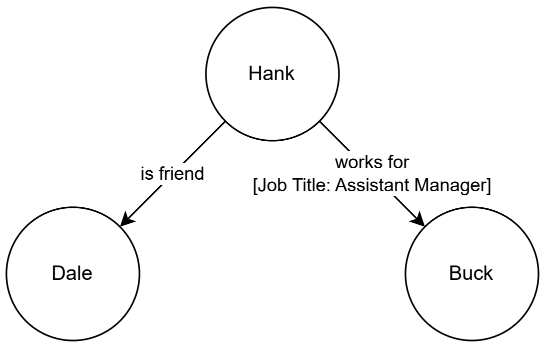
1. V is a finite set of pairs (v, P_v), where v is a node (or vertex) and P_v is a finite set of property pairs (key, val) with $key \in P, val \in Values$
2. E is a finite set of edges, where $e \in E$ is a tuple (a, b, l, P_e) with $a, b \in V$, $l \in L$ and P_e is a finite set of property pairs (key, val) with $key \in P, val \in Values$
3. L is a finite set of labels
4. P is a finite set of properties
5. $Values$ is a set of values (usually restricted to a small finite set of pre-defined types)

Beyond our definition, many graph databases that use property graphs also allow nodes to be labelled directly. This improves the clarity of the graph and separates nodes into subtypes, which can also be useful for a range of performance optimizations. However, it does not fundamentally enhance the expressiveness of the graph.

Additionally, most systems enforce further restrictions on properties or labels that this overview will skip over. However, the overview papers summarized in ‹Related Works› (section 6) contain further pointers into the detailed differences between graphs used in graph databases.

Thanks to properties, we can now modify our initial social graph to represent all the information of interest to us (figure 4) without changing its overall structure.

Figure 4 Property Social Graph



Fundamentally, both edge-labelled graphs and property graphs are equivalent in terms of expressive power and can be converted into each other (see [Hay+04]), although property graphs will generally be more compact and allow the addition of new metadata without having to significantly reorganize the graph.

We restrict our approach in this thesis to property graphs because we need to compactly represent event attributes and additional metadata. Furthermore, most graph databases of interest to us use property graphs as their underlying abstraction (cf. ‹Temporal Graphs› (section 2.3)).

2.2 Graph Databases

The distinguishing feature of graph databases compared to other kinds of databases is that the provided data structures and schemas support graphs natively.

Graph databases are systems that store and manipulate graph-like data structures, and that provide ways to query and further analyze that data. Graph queries can be either local, such as the retrieval of a particular node, its properties, or the path between two nodes, or they can be global queries such as the calculation of PageRank or the traversal of the full graph.

Furthermore, they support interfaces (APIs) for graph-focused operations and typically support the use of dedicated query languages to express graph queries and large-scale operations.

Because graph databases are used for very large datasets that underlie user-facing applications, they must enable high throughput and low latency for their most common operations. That makes them a plausible candidate for our needs to represent and query the event stream of a running program.

Graph databases are more powerful than relational databases because they can model rich graph-like data more directly. They provide a great deal of flexibility for organizing and annotating that often heterogeneous data while retaining its overall graph structure. While NoSQL can be very flexible as well, they rarely provide native support for graph algorithms and queries such as finding paths between two nodes or retrieving a particular neighborhood of a node. They also model complex relationships more directly and succinctly than relational databases and without the need for complex join operations[Bes+23].

While there are many graph databases that have been implemented on top of other databases, especially key-value or document stores, there are also established native graph databases nowadays. In this thesis, we restrict ourselves to one such native graph database, Neo4j[Neo24c].

The core features of graph databases that we are interested in within the scope of this thesis are the ease of embedding it into our breakpoint backend, its internal graph representation, the ways to query the graph, and general database guarantees such as consistency and support for concurrency. The overview papers in ‹Related Works› (section 6) provide more detailed taxonomies.

Graph databases usually use either the edge-labelled or property graph model we defined earlier as their fundamental abstraction.[Ang+17] Nonetheless, there are many internal representations of graphs used by graph databases today, although two of them make up the majority [Bes+23] — adjacency matrices and adjacency lists.

In an adjacency matrix, we store the connection between two nodes in a two-dimensional matrix of nodes by setting the entry (a,b) for two nodes a and b to 1 if both nodes are connected.

Definition 2.4 (Adjacency Matrix)**.** An adjacency matrix is a matrix $M \in \{0,1\}^{n,n}$ for the graph G with edges E and $n = |V|$, such that $M_{a,b} = 1 \iff (a,b) \in E$.

Alternatively, we can store the edge between two nodes by using a list for each node in the graph and placing any outgoing edges in their corresponding list. That way, each adjacency list represents the immediate neighborhood of its corresponding node.

Definition 2.5 (Adjacency List)**.** An adjacency list is a finite set A for each node v in the graph G with edges E, where $a \in A$ if there is an edge $(v,a) \in E$.

Adjacency matrices allow checking whether two nodes are connected in $O(1)$ time, but require $O(n^2)$ space and are typically sparsely populated. Additionally, if a bitmap is used to reduce the wasteful memory layout, then there is no obvious place to also store the properties of an edge, further complicating the design.

An adjacency list however only uses $O(n+m)$ space and can check for connectivity in $O(|A|) \subseteq O(d)$ time (with d the maximum degree of the graph) [Bes+23]. In particular, an adjacency list is an efficient way to store and access the immediate neighborhood of a node and can also conveniently store the edge properties for a given relationship.

Many graph databases use some form of adjacency list for its compact memory footprint and because it allows for an efficient traversal of neighboring nodes, which is also the most common use case when querying the graph.

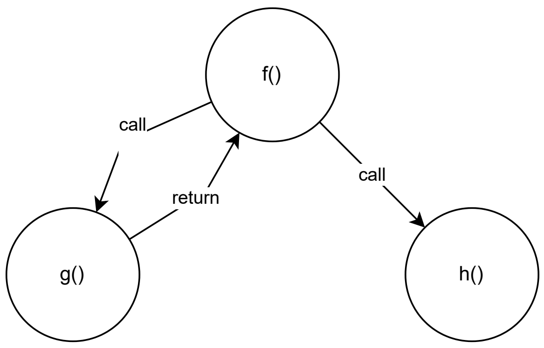
A very different approach is graph streaming[Bes+21], where the graph data is processed as a (typically distributed) stream of graph updates. While this is a very powerful and efficient way to handle adding or removing edges, graph streaming approaches rarely support rich data types or properties, instead being limited to fairly uniform and simple data. They also rarely support data persistence or consistency [Bes+23].

Because graph streaming highly restricts the richness of the model and the amount of metadata that can be used, and because it lacks the fundamental database guarantees we require for our design goals, we didn't focus on them and did not investigate their suitability further. However, future work in this direction may still be worthwhile.

2.3 Temporal Graphs

An important problem we face when trying to represent the event stream as a graph is that we need to model the explicit order of execution. For example, we might add a node for a method call and then return from that node when the method call is completed. The graph would then branch to another node for the next event, maybe another method call (cf. figure 5).

Figure 5 A Branching Event Graph



In the previous work by [Boc+21] and [Dor21], the event stream was modelled as a tree with a deterministic order for the children of any given node. The XML implementation in particular relied on this implicit order to traverse the tree in the correct sequence. This approach does not work for graph databases however, as there is no such implicit order and we will have to define an explicit property to preserve the execution order at branching nodes.[5]

Our solution is to annotate the transitions between states — ie. the edges of the graph — with the logical time of the corresponding event. This approach only requires a single global counter that is increased monotonically for each event.

We could alternatively define an edge property `Order` and enumerate branching edges in increasing order as they are added. This local counter would be logically equivalent to the global counter as one can be readily converted into the other when traversing the full graph, but this local counter would drastically complicate the query logic as detailed in ‹Implementation› (section 4) and would prevent direct global comparisons of time annotations. For example, we could tell which local branch is earlier, thereby preserving the execution order, but we could not easily tell which of two distant branches is earlier without at least calculating a path between them.

By adding a temporal annotation, our graph becomes a type of temporal graph. We will now discuss multiple specific problems a temporal graph has to address and discuss the specific design we chose for our thesis. We mostly follow [Deb+21] as our primary influence. For an overview of some the other existing work on temporal graphs, see ‹Related Works› (section 6).

A temporal graph is a graph that has been annotated with a time property that represents when certain nodes or edges are valid.

For example, in a flight graph we might want to represent at what time a flight between two nodes is available, so instead of just storing which destinations can be reached from a given airport, we also store the specific times of scheduled flights so we plan routes between locations.

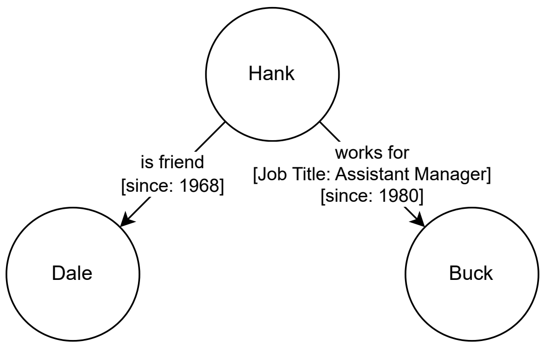
Another example would be a social network graph where we might want to store when two people are friends. Instead of just noting, like in our previous simple example, that Hank is friends with Dale, we would also annotate that edge with the time they became friends and maybe also the time they stopped being friends, if any (figure 6).

By adding temporal annotations directly to the graph, it lets us store metadata about when information in the graph is valid or has been changed, which also lets us represent the evolution of the graph in a more direct and embedded way. For example, if Dale changes his name to Rusty, we might want to preserve both pieces of information with the correct time annotation instead of overwriting his name completely, losing the history of the graph.

None of the established graph databases support nested properties, so temporal annotations can only be added directly to nodes and edges.

[5] Alternatively, we could model the stream such that there are no branches in our graph, but then we would've reduced it to a mere linear sequence, losing all the advantages a graph model would give us.

Figure 6 Temporal Social Graph



Consider the previous example of Dale/Rusty, which we might initially model as a Person node with a Name property. However, once Dale changes his name, we want to store his new name without losing his previous one, instead keeping the time when each of the names is valid.

Because we cannot add annotations to the Name property directly, a common solution (as used by [Deb+21] and others) is to introduce attribute and value nodes for each property we would like to annotate in that way. The Person node would now be connected to an Attribute node called Name, which in turn would be connected to two separate Value nodes, each storing one of Dale's names. Each of those edges would then contain the corresponding temporal annotation, as depicted in figure 7.

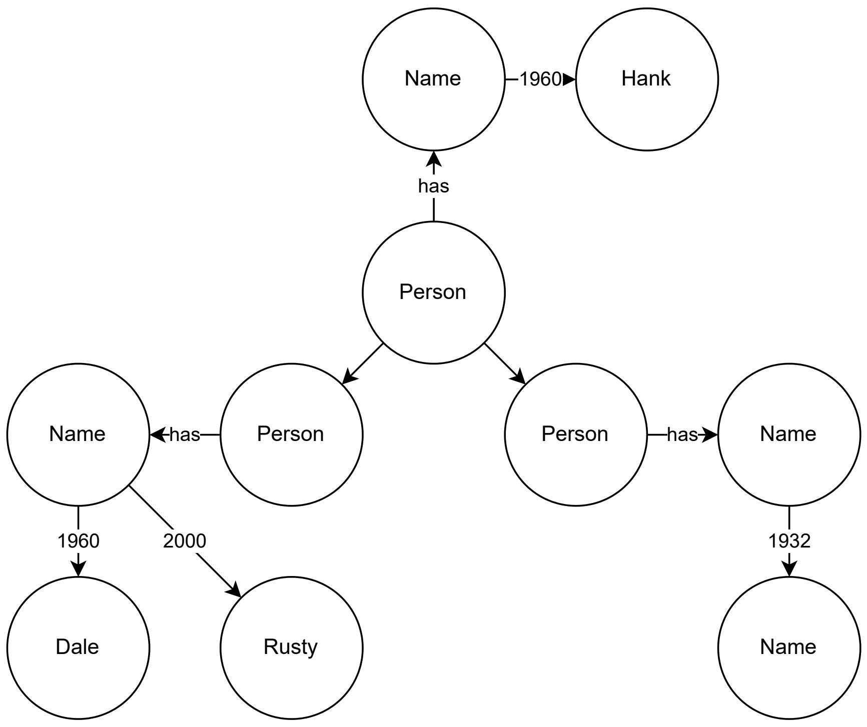
In our case however, the attributes of an event do not change once the event has been emitted and added to the graph. Therefore, we do not need to use value and attribute nodes but can instead store our properties directly in the event nodes and edges they belong to (cf. ‹Modelling the Event Stream› (section 4.1)). The only kind of change in our graph is the addition of new nodes and edges over time. We only need to annotate edges with the logical time they correspond to in order to preserve the execution order.

For simplicity's sake and because it is not required for the modelling approach described in this thesis, we will assume that only edges are ever annotated with time and will not discuss time annotations for nodes.

Additionally, this use of temporal data lets us compress the graph significantly. Any repeated program information that is static (like the name of methods), we can represent with a single node for the entire execution. Transitions between states are represented by edges between these static nodes, so we can then store dynamic data (such as the runtime value of a variable) as edge properties and distinguish multiple transitions by their time annotation.

In fact, the only new information we add to the graph during the execution of the program are the state transitions, their dynamic attributes, and the time annotation. This allows

Figure 7 Attribute and Value Nodes



us to use a much more compact graph representation that we can even suggest plausible bounds for. ‹Modelling the Event Stream› (section 4.1) will go into the details.

Adding an edge property based on a global counter is a relatively light extension that fits well within existing graph databases. The only significant adjustment we need is to extend the graph query functionality to take the time property into account when finding paths in the graph, something previous work has shown to be feasible with acceptable overall performance.[Deb+21]

Within the context of execution, time may represent either the real time of the particular execution (eg. "*12:04:01 on March 3, 2023*"), or the logical time of the event (eg. "*the 17th event*") or corresponding instruction (ie. the CPU time). As long as the timestamp used is monotonic (such as UTC) and has sufficient precision, the execution order will be preserved. This would allow us to support new temporal queries for breakpoints such us "*variable has changed in the last 10 minutes*".

Furthermore, temporal annotations of either kind would provide limited profiling data with no additional overhead. As ‹Evaluation› (section 5) will show, our debugging overhead is already fairly expensive, so this is not an ideal solution for profiling, but it comes at no additional cost to us, so further exploration may be worthwhile.

Lastly, in order to support concurrency, we will also have to annotate our state transitions for each strand[6] of execution (cf. ‹Modelling the Event Stream› (section 4.1)). Adding another edge annotation for the thread ID (or a similar identifier) would be a natural solution and synergizes well with the temporal annotation of the edge.

While we restrict our temporal annotations to a single point in time when the transition represented by an edge is valid, there are also other ways to encode temporal information in temporal graphs [Deb+21].

In addition to a single point, one may us an interval and represent the validity with a pair of starting and end time. Similarly, instead of providing explicit start and end points, one could also use the duration of the relationship, representing for example the necessary travel time in a transportation network.

Both of those approaches are widely used, but given that our event stream gaplessly models execution, intervals do not provide any useful additional information and durations would not allow us to reconstruct the correct execution order.

Lastly, there are also snapshot-based temporal graphs (cf. eg. [BWK20]). Here the temporal information used is the global time when a given state of the graph is valid, such as for example the editing history of a Wikipedia article. Even though this approach is similar to our temporal annotation, we are not concerned with wide-ranging changes to the graph over time and alternative versions. For some of the related work, see ‹Related Works› (section 6).

2.4 Graph Query Languages

In order to query and modify the graph data in a graph database, these systems provide not just programmatic APIs but also dedicated graph query languages. Graph query languages like Cypher or Gremlin are the graph equivalent of the popular query language SQL for relational databases.

When it comes to the taxonomy of graph query languages, we primarily follow the work of Besta et al. and Angles et al., as described in ‹Related Works› (section 6).

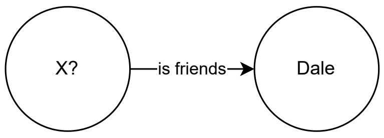
Graph query languages can be understood as a set of operators or inference rules for interacting with the database, plus several powerful ways of combining these operators into more complex queries [Ang12].

[6] The DiSL instrumentation framework and Java programming language use a thread-based model of concurrency and our implementation currently follows that design. However, so as to not limit the support to a particular model of concurrency, as opposed to for example processes, actors or green threads, we use neutral the term *strand* to refer to concurrent paths in the execution.

In particular, a basic graph query can be understood as a (sub-)graph where constants have been replaced by placeholder variables that need to be substituted with concrete values in such a way that the resulting graph is contained within our overall graph [Ang+17].

Going back to our simple social graph example, we might express the query to find people who are friends with Dale in the following way, substituting the variable X? for the unknown node (figure 8).

Figure 8 Basic Graph Query



These basic graph queries (or basic graph patterns in [Ang+17]) already provide powerful relational operations such as equality comparison and natural joins. In addition, graph query languages also commonly support complex graph queries to provide additional relational operations such as unions, optional matches and filters. All of those operations are well-supported by the three big graph query languages we will discuss later in this chapter.

To extend the power of graph queries further, we also need operators to navigate the graph and find paths, especially paths of unknown or varying length. For example, we might want to find not just Hank's friends, but also his entire extended friend group, including all friends of a friend and so on.

Alternatively, we might want to find the shortest path between two nodes[Deo16], for example to calculate routes in a transportation network, or to answer the question whether two given nodes are connected by a path or not, or when trying to identify whether two people in a social graph have some connection with each other.

Path queries like that may also be further restricted to certain edge labels or conditioned against certain edge properties, for example when trying to find a route between two locations that only relies on public transportation.

One particularly influential way to model these powerful navigational queries is known as conjunctive regular path queries (CRPQs)[Woo12]. The details of CRPQs however are beyond the scope of this overview and general thesis.

In the space of graph databases, there is a general distinction between Online Transactional Processing (OLTP) and Online Analytical Processing (OLAP) [Bes+23].

OLTP queries are usually small and restricted to local parts of the graph such as the close neighborhood of a given node. They analyze or modify only a small part of the graph and are expected to be processed with low latency.

OLAP queries on the other hand are usually concerned with the entire graph or very large parts of it, and are more concerned with the overall throughput. They include PageRank or whole-graph traversals.

Within the context of breakpoints used for live debugging, we are primarily interested in OLTP queries, both to add new events to the graph, and to match breakpoint queries incrementally against the latest event. Analysis of the execution trace through OLAP queries may certainly be of interest to the programmer, but falls outside of the scope of this thesis.

Besides the expressive power of graph queries, there are also other concerns to consider.

In order to ensure the correct modelling and storage of the event stream, especially under concurrency, we desire the well-known ACID criteria (Atomicity, Consistency, Isolation, Durability)[HR83]. Fortunately, most native graph databases support proper ACID transactions[Bes+23].

Lastly, graph databases and their query languages differ in what type systems and ontologies they support. However, for our purposes we are only concerned with a small set of primitive types (numbers, booleans, strings) widely supported by most graph databases. Future work into support for more complex data types, especially the embedding of objects for object-oriented programming environments, would benefit the overall power of breakpoints greatly.

Graph query languages fall primarily into two basic approaches, either a declarative design that attempts to represent the (sub-)graph being matched, or an imperative design that represents the traversal of the graph to find a match.

Cypher is a representative and widely used example of the declarative approach, and Gremlin of the imperative one. Both will be briefly discussed next.

Within the scope of this thesis, we needed a mature graph query language for a native graph database suitable for temporal annotations. Cypher and Neo4j fit these criteria very well. Additionally, multiple existing temporal graph approaches have extended Cypher and shown its basic suitability for temporal graphs (cf ‹Related Works› (section 6)). Our main focus will therefore be on the Cypher query language.

2.4.1 Cypher

The Cypher query language is a declarative language for the Neo4j graph database based around "patterns" expressed in a visual style influenced by ASCII art and the popular SQL query language.[Fra+18] The basic query pattern is expressed through the `MATCH` clause. While there are many additional operations for manipulating the graph and for filtering and processing query results, we will focus our overview only on the type of query of interest in the context of breakpoint specification.

To illustrate the syntax of Cypher, we will show a brief sequence of typical examples of increasing complexity.

Starting with the same kind of social graph as shown in previous examples (figure 4), we will first find all people who have friends and then specifically all people who are friends with Dale (cf. figure 8).

```
// Find all people who have a friend and return both the person and the friend.
MATCH (x:Person)-[:friends]->(y:Person) RETURN x, y

// Find all people who are friends with Dale.
MATCH (x:Person)-[:friends]->(y:Person {name: "Dale"}) RETURN x
```

Listing 1 Finding Friends

Next we will query the database for all people who are employed and then specifically only those whose income value is greater than 3,000.

```
// Find all employed people (whether by a Person or other type of node).
MATCH (x:Person)-[:works_for]->() RETURN x

// Find all employed people whose income is more than 3000.
MATCH (x:Person)-[r:works_for]->() WHERE r.income > 3000 RETURN x
```

Listing 2 Finding Employees

Lastly, we will consider a sample of more complex examples to show more of the expressive power available for patterns in Cypher.

```
// Find everyone who owns a car and who is friends with someone who owns a bicycle.
MATCH (a:Car)<-[:owns]-(x:Person)-[:friends]->(y:Person)-[:owns]->[b:Bicycle] RETURN
    ↪ x

// Find all the people who are a friend of a friend or extended relative of Hank.
MATCH (x:Person {name: "Hank"})-[:friends|related *]->(y:Person) RETURN y

// Find everyone who works for someone younger than them.
MATCH (x:Person)-[:works_for]->(y:Person) WHERE x.age < y.age RETURN x, y
```

Listing 3 Complex Queries

2.4.2 Gremlin

Gremlin is an imperative query language for the TinkerPop3 graph database[Rod15], in which queries are expressed primarily in explicit navigational form, ie. the specific traversal strategy to use to find potential matches. Nonetheless, Gremlin also supports a range of pattern matching and advanced relational operations.

For comparison, we will show a sample of equivalent examples to the ones used in the previous section on Cypher.

```
// Find all people who have a friend.
G.V().hasLabel('Person').out('friends').hasLabel('Person')

// Find all people who are friends with Dale.
G.V().hasLabel('Person').has('name','Dale').in('friends').hasLabel('Person')

// Find all employed people (whether by a Person or other type of node).
G.V().hasLabel('Person').out('works_for')
```

```
// Find all employed people whose income is more than 3000.
G.V().hasLabel('Person').where(outE('works_for'), gt(3000))
```

Listing 4 Gremlin Queries

In light of our need to ensure the correct traversal of the graph, in particular the preservation of the execution order (as already mentioned, but also discussed in detail in ‹Modelling Breakpoint Queries› (section 4.2)), one may ask whether an imperative approach might be particularly well-suited.

We certainly could've chosen it as the base for our breakpoint queries, but note that we would have to repeat our restrictions on traversal for every point in the query where edges are chosen. This would've been very cumbersome and repetitive, effectively amount to us implementing our own traversal function for the query engine. This would negate many of the advantages of the Gremlin engine. Additionally, the explicit goal of our approach was to evaluate whether a mature system could be used without the need to implement a custom backend (in parts or as a whole), and a custom traversal function would certainly negate that goal.

3 Breakpoint Requirements

We already touched upon the requirements for a breakpoint implementation in the introduction. This chapter will develop more explicitly the specific goals that a graph-based approach has to fulfill within the scope of this thesis and which we will evaluate our implementation against in ‹Evaluation› (section 5).

3.1 Performance

To be usable for live debugging, ie. the incremental matching of breakpoint queries against the event stream as events are emitted, the query backend must have a small overall time overhead, as otherwise the general performance of the program being investigated would slow down too much.

Previous work (in particular [Boc+21]) had very large overheads that were difficult to reduce further. While this initial exploration of graph-based approaches is unlikely to exhaust the full possibilities of low latency queries, we would like to identify possible bottlenecks and inherent overheads when using a mature graph database and graph query language, in our case Neo4j and Cypher.

Additionally, the performance overhead should remain stable throughout the execution. For example, if we simply performed a full global search for every event, matching against the entire graph and only filtering results after, then the runtime cost would grow with the size of the graph and length of execution, and so would soon become unacceptably slow.

We certainly cannot guarantee low runtime performance overheads for all possible queries, especially with the kind of powerful query language we are developing that allows arbitrarily complex computations to be performed when matching against an event. Nonetheless, we desire low latency for typical queries that represent the common use cases, such as basic graph patterns and small bounded path queries.

Furthermore, the memory overhead of the event graph and of the graph database itself should be relatively low. In particular, the memory size of the graph should scale at most linearly with the number of events. This constitutes a natural lower bound of the memory size absent of any graph pruning or compression techniques. The incremental search performance should nonetheless remain stable for a given query even as the graph grows in size.

Global query searches should also perform at speeds suitable for interactive use of the debugger. This requirement is a general objective for graph databases and previous work has shown this to be true for comparable queries (cf. [Deb+21] or many of the other works referenced in ‹Related Works› (section 6)).

Lastly, any startup overhead that is necessary to initialize the graph database and (if required) statically analyze the program should not impose a significant delay.

3.2 Expressive Power

An important design goal of a graph-based query language is its expressive power. The query language must be able to represent all the core parts of our event ontology and facilitate writing queries for all possible breakpoint events. It must allow matching against both the most recent event and also permit references back to past events.

The graph query language should be able to condition against all static and dynamic attributes of the execution events, allowing queries such as "*break if the method f is called*" and "*break if $x > 10$*". This includes secondary attributes such as the line number of the corresponding source code.

In addition to event attributes, queries should also be able to refer to the temporal annotations, enabling new kinds of breakpoints such as a temporal order (eg. "*break if X happens after Y*") or, if the corresponding annotations are used, real-time properties (eg. "*break if X happens within a minute of Y*").

Graph queries should support paths with a known fixed size, but also paths of unknown and potentially unbounded size.[7]

Note that [Boc+21]'s original ontology does not directly include conditional and iteration events such as if clauses or loops. The Java bytecode instruction stream used in this implementation is also stripped of most of this information and the DiSL instrumentation framework does not directly expose them to us, so we considered fully supporting them as beyond our immediate scope, but the underlying graph and graph query language should nonetheless support them for possible future extensions of this work.

Previous approaches have struggled to support symbolic references. [Dre23]'s temporal logic approach did not provide references and [Dor21]'s query language only supported a restricted set of bindings that significantly increased the complexity of the implementation. At the least, the graph query language should be able to bind matching nodes or edges to names and allow references to those names in other parts of the query. Preferentially, queries should also allow similar symbolic references to matching paths or sub-graphs.

In addition to simple breakpoint queries, we are also interested in the ability to compose queries into more complex ones. As described in ‹Graph Query Languages› (section 2.4), graph query languages generally provide a wealth of relational operations and logical compositions, such as union, join or filter. We would like to use these compositions for our breakpoint queries as well.

A particularly useful composition of queries is the support of nested queries to express conditions such as "*break if x is changed from within method f*".

[7] Of course, any sensible breakpoint query path is ultimately bounded by the size of the entire graph, as there is little utility in breakpoints that include infinite loops or similar unbounded conditions.

Lastly, to offer the best possible debugging experience for the programmer, breakpoints should be changeable during runtime. It should be possible to define new breakpoints or modify existing ones even after the program has already begun executing. This reflects the common debugging experience where the programmer might define a breakpoint with fairly broad conditions to start narrowing down the execution to parts that seem of tentative interest, and then after investigating the program state after a match, they might define new, more specific breakpoints that might correspond to a particular bug in the program. [Dor21]'s previous work did not allow such redefinition, which significantly restricted the usefulness of that approach to debugging.

Note however that some optimizations that improve the query performance also compete with the goal of letting the programmer redefine breakpoints during a live debugging session. For example, if we filter the event stream so that it only contains events that might be of relevance to the initial breakpoint queries, we will reduce the size of the resulting graph, but at the same time we then lose the ability to match new breakpoints against old events that might have been filtered out. We could remove the filter going forward, but we cannot retrieve the old events and recreate the missing parts of the graph retroactively without also rewinding the program state.

3.3 Ease of Implementation

A major weakness of previous approaches was the need to implement a full search backend from scratch. This problem already affected [Boc+21]'s original approach as the existing XQuery implementation failed to meet the performance requirements of incremental search and so motivated the development of [Dor21]'s more efficient but also more restricted custom backend.

In order to reduce the implementation burden for developers and ensure the overall correctness of the search backend, especially under complex conditions such as concurrency, we would like to reuse an existing mature system as much as possible.

While some adaptation of the graph database will likely be necessary, it should not circumvent the existing graph query engine or duplicate core functionality already supported by the graph database for our search backend.

3.4 Support for Concurrency

Another important design goal is the ability to model concurrency. Previous approaches were fundamentally constrained to single-threaded execution. In order to support concurrency, we must be able to distinguish multiple parallel strands of execution in the event stream. At the very least, it should be possible to include multiple strands of execution in the same graph, and to access and modify the graph database from multiple independent strands.

Queries should not accidentally match against events from different strands in a way that violates the actual execution of the program. More complex extensions may be possible, such as query conditions that refer to other threads, but those are beyond the scope of this exploratory thesis. Our initial goal is only the correctness of query results with regards to concurrency.

Ideally, the graph query language should not fundamentally commit us to a particular concurrency model, although the underlying programming language will likely impose such a model, such as the processes and threads model in common use today.

3.5 Representation and Ergonomics

[Dor21]'s previous work was predominantly concerned with the ergonomics of the breakpoint query language, intended for it to be used directly by the programmer. We do not share that goal in this thesis to the same extent and expect the queries to be at least partially composed by the debugging environment, maybe via compilation from a more ergonomic user language or through a GUI interface.

Nonetheless, the graph query language should allow relatively straightforward representations of breakpoints and avoid overly verbose or error-prone constructions. This would simplify the implementation burden for developers of debugging tools and allow more widespread adoption.

This last design goal is somewhat subjective, as all ergonomic concerns are. As this is a more secondary goal for our present thesis, we refer to [Dor21] as an alternative approach with a focus on ergonomic optimization and general usability.

4 Implementation

Our implementation builds on the work of [Boc+21] and [Dor21], using the same DiSL instrumentation framework[Mar+12]. That way, the results are more directly comparable with previous works and reduce the implementation burden. We extended the system with a new graph-based breakpoint backend, powered by the Neo4j graph database [Neo24c], in which breakpoint queries are expressed through the Cypher query language [Fra+18].

First, we will describe how to model the event stream as a graph in the Neo4j graph database. Then we will discuss the breakpoint queries and their implementation concerns. Finally, we will present several important optimizations we attempted and their overall impact on the system's design and its performance. The next chapter contains a detailed evaluation of the complete backend.

4.1 Modelling the Event Stream

Events can be divided into three parts — static event properties (eg. the event type), dynamic event properties (eg. the variable value) and the transition from the previous event to the new one. The static properties are unchanged throughout the execution of the program and only have to be stored once in the event graph.

The dynamic properties only remain valid for a given event and are therefore constrained to the transition to the new event. They can be stored as properties of the transition itself. This way, both the dynamic properties and the transition properties, in particular the time of the transition, can be stored as properties of the same edge, reducing the number of entities in the graph.

There is one exception for the static properties. Entities such as variables may be referenced from multiple parts of the underlying source code, so the static source code location of an event containing an entity may differ depending on which part of the program code is being executed.

An alternative implementation may distinguish entities further such that, for example, "*method f called from line 37*" and "*method f called from line 142*" would each receive their own unique node, representing all static properties only once in the graph. While this would remove some redundancy for the edges, it would multiply the number of nodes in the graph.

One disadvantage of this approach is that it would require two properties to look up existing nodes — their name and their source code location.

Additionally, the dynamic values of an event type would no longer be stored in the immediate neighborhood of a single node, but potentially multiple nodes. When using only a single static node, all the associated past dynamic values are in the immediate neighborhod of the corresponding event node and can be easily retrieved and sorted chronologically. This information can be used to efficiently assemble debugging information such as the value history of a variable or the range of method arguments used by the program, to be displayed and used in the debugging environment. Splitting nodes based on their source code location, however, would make it more computationally expensive to collect this information.

Because of those disadvantages and informed by preliminary benchmarks, we favored using a single node for each program entity, storing the source code location as an edge property as if it were a dynamic property.

By modelling the event transition separately as an edge, [Boc+21]'s ontology reduces to four static event types — methods, variables, classes, and exceptions — which we represent with four node labels. This allows us to reuse the nodes for multiple events, reducing the size of the graph dramatically (cf. ‹Evaluation› (section 5)).

There are only seven kinds of transitions in our breakpoint ontology, namely calling and returning from a method, reading and writing a variable, throwing and catching an exception, and loading a class. We represent them with seven edge labels.

Further differentiations might be possible, such as between different kinds of variables, but because the DiSL instrumentation framework does not readily expose this information to us (cf. [Dor21]), we chose to limit ourselves to this simple but fully functional prototype.

We initially start the graph with a dummy node to ensure that the first edge has a node to originate from. The graph database is then updated whenever the instrumentation framework emits a new event. The corresponding static node is looked up and added to the graph if necessary. Lastly, a new edge is added from the previously active node to the new node.

In addition to the event types already discussed, it would be possible to extend the breakpoint ontology and introduce nodes for control flow operators (eg. loops or conditionals). However, the DiSL instrumentation framework does not natively provide this information to us, reporting only low-level jumps and branches contained in the program bytecode, and not the original operators as they are expressed in the original source code. This would muddy the resulting graph significantly, obscuring the program structure rather than enhancing it. Extending the ontology is therefore beyond the scope of this thesis, but a better bytecode model and instrumentation framework are a high priority for future research and are under active development.[Yil+18]

After adding the necessary nodes and edges to the graph, we save the event attributes as properties. The static properties (except for the source code location) are added to the node when the node is first added to the graph and otherwise remain untouched. The dynamic event properties (such as the variable value) as well as the source code location and the temporal annotation are added to the new edge.

It is important to remember that graph representations don't generally define an explicit ordering over the children of a node, as noted in ‹Temporal Graphs› (section 2.3). Therefore an order has to be explicitly reconstructed based on the available edge properties, in our case relying on the temporal annotation.

In principle, nodes could also carry temporal annotations, but it is not relevant to the correct traversal of the graph and would either prevent the nodes from being shared between different events or require the modification of existing nodes, both of which would conflict with important optimizations (‹Optimizations› (section 4.3)). Additionally, all the necessary temporal annotation to reconstruct the event sequence is already contained in the edge properties.

As our model is only concerned with an event stream that does not travel back in time (as might be the case for a time-travelling debugger), properties of existing nodes and edges do not change. As such, having dedicated attribute and value nodes as in [Deb+21] (cf. ‹Temporal Graphs› (section 2.3)) is not necessary and all properties can be stored directly in their respective nodes and edges.[8]

Additionally, our demands for temporal information are fairly limited and mostly concerned with the preservation of the execution order. We do not perform any modifications or deletions within the existing graph. As a result, our temporal graph model is not as complex as [Deb+21] or [Mas22].

There remains the question of how to exactly encode the temporal annotation. We restrict ourselves to a single point in time that will be added as a property to each edge. Temporal graphs also commonly use intervals[Deb+21], defining a start and end point instead, but our events are too fine-grained and span too short a duration to meaningfully benefit from intervals. As touched on in ‹Temporal Graphs› (section 2.3), there are two fundamental time representations that might be of interest to the breakpoint backend.

The simplest solution is to use the logical time of an event, which corresponds to the number of the event in the event stream and which can be easily implemented through a global event counter. This way, a simple 64bit machine integer is sufficient to establish the temporal order of events.

Alternatively, the temporal order can be established through the external time of the event, either through the use of a real-time clock or a CPU instruction counter. In order to avoid any ambiguities or contradictions however, the time annotation has to increase monotonically and has be provide enough precision to uniquely identify each event. A sufficiently high-resolution UTC-based clock or non-wrapping CPU instruction counter would fulfill this requirement.

Retrieving this external information would add to the overhead of the breakpoint backend, which is why we decided to employ the basic event counter.

To support basic concurrency, we assign a unique ID to each concurrent strand of the program and mark every edge of the graph with the ID of the strand the event belongs to. In the DiSL-based Java implementation, this ID corresponds to the thread ID, but depending on the language and its concurrency model, a different ID might have to be

[8] Another advantage cited in [Deb+21] for moving some event attributes from edges into dedicated value nodes was to utilize the existing indexing architecture of Neo4j, as indices over edge attributes were unsupported at the time. This limitation does not exist anymore, starting with Neo4j version 4.3. [Stu21]

chosen. When matching a breakpoint query against the graph database, we ensure that candidate paths only span a single thread ID for the entire subgraph, analogous to the way the execution order is preserved (listing 5).

```java
private boolean isValidPath(Path path) {
  if (path == null) return true; // accept empty paths

  for (var rel : path.relationships()) {
    // check thread safety
    int id = (int) rel.getProperty("thread", 0);
    if (id > 0 && id != tracerID) {
      return false;
    }
  }

  return true;
}
```

Listing 5 Enforcing Concurrency

Finally, there is the general issue of how to encode the dynamic and static attributes of events. While some attributes map easily to a native Neo4j type, such as the line number or method name, others can have complex types that do not correspond to any native type in the graph database. This affects primarily the method arguments, return values and the dynamic values written to or read from variables.

The DiSL-based instrumentation framework that the breakpoint query relies on distinguishes between basic Java types, such as booleans, integers or strings, and objects that may belong to any arbitrary class and may be user-defined.

The basic types all have corresponding native types in the Neo4j graph database and can be stored easily and without loss of information. However, objects cannot in general be serialized without loss of fidelity, and may even be mutable and cannot be copied. Additionally, even when they could be accurately serialized in principle, there is no single standardized interface in the Java programming language to do so. Exploring ways to represent these values and make them available to the programmer for comparison may be a fruitful opportunity for future research.

[Dor21]'s previous work stored such values as object references and only allowed direct comparisons with other objects, which had only very limited utility for breakpoints. As Neo4j does not allow the direct embedding of references, we decided to encode objects as strings, using the standard toString() method to derive a string representation. This way, many common classes such as lists are represented in an intuitive manner that can be compared with common string operations. Furthermore, this string representation is also immutable and reflects the object state at the time of the event. This limitation is not unusual for debuggers, which rarely provide more sophisticated object operations, especially outside the domains of language-enforced value semantics and immutability.

4.2 Modelling Breakpoint Queries

To illustrate the range of possible queries supported by the breakpoint backend, we are going to present a range of examples. They cover queries that depend on the event type and the static and dynamic event attributes, but also past events with both a known and an unknown distance to the most recent event. Note that for all these examples, the complete matching path p in the graph is returned to be processed afterwards in the breakpoint backend, as we will discuss shortly.

```
// Break if the event type is a method call.
MATCH p = (m:METHOD) RETURN p

// Break if any method is returned from.
MATCH p = ()-[:RETURN]->(m:METHOD) RETURN p

// Break if the method "f" is called.
MATCH p = ()-[:CALL]->(m:METHOD {name: "f"}) RETURN p

// Break if the method "f" is called with the arguments 3 and 17.
MATCH p = ()-[:CALL {args: "[3, 17]"}]->(m:METHOD {name: "f"}) RETURN p

// Break if the variable "x" receives the value 10.
MATCH p = ()-[:WRITE {value: 10}]->(v:VAR {name: "x"}) RETURN p

// Break if the variable "x" receives a value less than 4.4.
MATCH p = ()-[r:WRITE] ->(v:VAR {name: "x"})
WHERE r.value <= 4.4
RETURN p

// Break on source code line 60.
MATCH p = ()-[r {line: 60}]->() RETURN p

// Consider two basic method calls, an outer and an inner one:
// public void outer() { inner();  }
// public void inner() { /* ... */ }
//
// Break when the inner method is called directly inside the outer one.
MATCH p = (o:METHOD {name: "outer"})-[:CALL]->(i:METHOD {name: "inner"})
RETURN p

// Consider two method calls:
// public void init() { /* ... */ }
// public void use()  { /* ... */ }
//
// Break if "use" is called at some unknown point before "init":
MATCH p =
  ()-[:CALL]->(x:METHOD {name: "use"})
  -[*]->()->
  [:CALL {}]->(y:METHOD {name: "init"})
RETURN p
```

Listing 6 Breakpoint Queries

For incremental searches — the typical mode of operation when using a debugger — the breakpoint backend attempts to match all defined breakpoint queries immediately after adding a new event to the graph.

In order to include the latest event in the potential match and prevent suspending the execution for matches entirely located in the past, queries are anchored to the most recent event. We accomplish this by providing two parameters for our queries, `$node` and `$time`, which refer to the last node and event time respectively. The breakpoint query must then include one or both of them in its condition to properly anchor the query. Consider the following basic example, first without anchoring and then properly anchored:

```
// Break if any method is called. Return the full matching path, ie. two nodes and
    ↪ one edge between them.
MATCH p = ()-[:CALL]->(m:METHOD) RETURN p

// Anchor the previous query against the most recent event.
MATCH p = ()-[:CALL {time: $time}]->(m:METHOD) RETURN p
```

■ **Listing 7** Anchoring a Query

Alternatively, instead of anchoring the query itself through an explicit condition, the traversal function of the query engine could be modified so that it always starts its search at the most recent node / edge and aborts an attempted match if that edge is not included in the result. This would remove the need for an anchoring reference in the query, although an explicit anchor might still improve the readability.

Because one of the design goals of this implementation is the use of the existing Cypher query engine, we cannot modify or replace its traversal function and must instead influence its search strategy only through the query conditions.

For global searches that would return all possible breakpoints in a given graph, we can simply run the same search without anchoring it, and if desired extract the earliest event included in each match to identify which specific events would constitute a breakpoint.

Naively, global search could also be implemented by passing every possible anchor time as the parameter to the query (ie. by enumerating all the events included in the graph) and attempting a match. That approach would obviously be very inefficient and work around many optimizations of Neo4j's query engine.

If we had control over the traversal function, then a further optimization for global search would be possible. Attempted matches could be aborted if, after finding a matching result, further attempts to find new matches use the same earliest event, , ie. they only contribute possible extensions to a known match in the graph without changing its starting point. Such a constraint is not practical to enforce through Cypher directly. As noted in ‹Evaluation› (section 5), there is clearly a lot of potential to improve the search performance by using a purpose-fit traversal function.

Besides anchoring, the breakpoint backend also has to enforce the correct event ordering when choosing branches. For simple queries, this is not an issue, as the desired sub-graph will not include any potential looping paths that pass through the same node more than once.

In fact, the most common breakpoints only match against a single event and therefore include only a single edge, which cannot by itself violate the temporal constraint. Consider for example the following three queries, representing the most common breakpoints supported by debugging environments:

```
// 1. Break if method f is called.
MATCH p = ()-[:CALL {time: $time}]->(m:METHOD {name: "f"}) RETURN p

// 2. Break if variable x has the value 10.
MATCH p = ()-[:WRITE {time: $time} {value: 10}]->(v:VAR {name: "x"}) RETURN p

// 3. Break if source code line 12 is reached.
MATCH p = ()-[{time: $time, line: 12}]->() RETURN p
```

Listing 8 Basic Breakpoint Queries

All three of those queries contain just a single edge in their result and so cannot match paths that do not follow the correct execution order. However, to allow for more sophisticated breakpoints that refer to past events, the breakpoint backend must support queries that match multiple edges. To do so, we require all queries to return the final matching path so that the backend can check it for temporal contradictions. This check is a simple iteration over all edges in the traversal order returned by the graph query engine, with a condition that the time property is monotonically increasing (listing 9).

```
private boolean isValidPath(Path path) {
  if (path == null) return true; // accept empty paths

  for (var rel : path.relationships()) {
    // check temporal ordering
    long last = 0;
    for (var rel : path.relationships()) {
      long t = (long) rel.getProperty("time");
      if (t < last) {
        return false;
      }
      last = t;
    }
  }

  return true;
}
```

Listing 9 Enforcing the Temporal Order

Alternatively, one might implement this traversal constraint through a Cypher plugin and a function call within the Cypher query, analogous to the path functions used by [Deb+21]. This solution would require all breakpoint queries to explicitly use this function on any matched paths. If the query author forgot to apply the constraint function to a path included in the query, the result might violate the execution order.

Fundamentally, both approaches would perform the same iteration and conditional over the candidate paths, resulting in no clear performance advantage for either in most situations.

We decided to use the internal check after the Cypher engine returns all plausible paths in order to improve the readability of the queries and to ensure that the correctness check cannot be forgotten. Additionally, we further improved the robustness of the query results by always running the check, even for simple queries that do not contain multiple edges or pass through branching nodes multiple times. We found no significant performance impact of this check compared to the large overhead of the query engine itself (as discussed in ‹Evaluation› (section 5)).

Instead of filtering the potential paths after the fact, the optimal solution would use a traversal function for the graph that only ever traverses edges in their correct temporal order. In fact, all that is required is for the traversal function to consider outgoing edges in their temporal order when attempting to match a query. This could be accomplished either through a simple sort whenever a node has multiple outgoing edges, or even more efficiently, by ensuring that the graph database stores edges in the same order they are added to the graph and then traversing the edges in insertion order. As edges are added chronologically and never removed, no additional sorting would be required.

Such a solution would be possible with a custom traversal function, replacing the one used by the Cypher query engine, as noted before. Neo4j does in fact support custom traversal of the graph, but not in combination with the Cypher query language. As a custom query engine with its own query language violates one of the core constraints of this exploratory thesis, we chose not to implement one.

It might seem like matches could be sped up by limiting the number of results through the use of `LIMIT 1` in the query, as the breakpoint backend only needs to demonstrate the existence of one matching path, but not necessarily all of them. Because the temporal ordering is only enforced after Cypher returns the potential matches to the breakpoint backend, it is not possible to guarantee that the first result found is an actual match. Our preliminary tests did not find a measurable performance impact from this limitation, however.

The breakpoint backend also supports complex queries that include more than a single path. Consider a query of the form “*break if A and B*”, where A and B are each a distinct path through the graph. To enforce the temporal ordering, we require the query to return all included paths for its final result and then only accept the match after iterating over all parts of the result and executing the correctness check.

4.3 Optimizations

4.3.1 Embedded Operation

Neo4j’s standard mode of operation is as a separate database process, possibly running on a different machine entirely. Communication is handled either through a custom TCP protocol or through the Java Bolt protocol[Neo24a]. The latency overhead that results from using a network protocol and separate process space is considerable. Therefore this process separation is well-suited only for distributed applications or applications that can tolerate higher query latencies.

Additionally, Neo4j forces all requests to be separate transactions when run in this mode. As we will discuss in the next section, this also adds a large performance overhead to every request.

In order to avoid this overhead, we implemented the graph database backend so that it uses Neo4j in its embedded mode. This has multiple advantages besides the much lower query latency. When embedded directly as a Java library, Neo4j provides direct access to its nodes and edges, and also fine-grained control of what queries should be grouped into a transaction and when those transactions should be committed to the database.

With direct access to nodes, we can use a simple hash map to retrieve nodes based on the static attributes of an event, bypassing the need to query the database for existing method or variable nodes. This cache is also significantly faster than Neo4j's own indices. Additionally, we can add new nodes and edges to the graph directly through the low-level interface instead of through Cypher queries, reducing the overhead further.

This embedded operation does increase the complexity of the backend implementation, but not by much. Additionally, Neo4j still allows remote connections to its database even in this mode, so the graph can still be accessed through all of the existing Neo4j tools as long as the debugger is actively running.

In total, this optimization reduced the runtime overhead of maintaining the graph by a factor of at least 1000 and facilitated multiple additional optimizations.

4.3.2 Minimizing Transactions

A very large portion of the initial runtime overhead of creating the graph database is due to the use of transactions. As the graph is only ever appended to and queries do not modify it, there is less of a need for transactions. Additionally, the graph only represents a single execution and commonly does not persist beyond that. Data loss due to an unexpected program crash is a less important concern for a debugging run.

Therefore, in order to reduce this overhead, the most efficient solution is for the breakpoint backend to use a single global transaction that is only committed at the end of the execution. The main complication with this solution is due to concurrency.

Concurrent strands of the program need to access a consistent shared state of the graph database. Even if we restrict breakpoints such that queries can only reference paths that belong to the same strand, there is still shared state if new strands can be started, as is usually the case. This problem can be solved, however, by enforcing a transaction commit whenever a strand is created or destroyed, forcing the graph database to merge all open changes. That way, breakpoint queries would remain consistent and most transactions could still be avoided, as the creation of new strands makes up a minority of the execution events even for a program such as a web server that dispatches a new strand for each request it receives.

For our initial exploration of a graph-based query backend, we did not try to achieve full support of concurrency while minimizing its overhead, so we restricted our evaluation to the best case scenario — single-threaded execution.

A future extension of this work towards full support for concurrency is an interesting direction to pursue. Due to the remaining overhead as described in ‹Evaluation› (section 5), however, there are far greater obstacles to overcome first, possibly necessitating the use of a different graph database or even a custom solution. Implementing correct concurrency would not be a priority goal under those circumstances.

4.3.3 Filtering

As already noted in [Boc+21], a promising general optimization for a breakpoint backend is the ability to filter which instructions should actually generate events, trying to reduce the size of the event stream to the smallest size that still contains all the events that should trigger a breakpoint while avoiding any superfluous events that cannot possibly do so.

[Dor21]'s previous work attempted a basic filter by extracting the names of methods and variables referenced in a query and ignoring any events that do not match those names. A similar optimization is possible for Cypher, as long as nodes are explicitly identified with their type label attached. If queries are automatically generated either through a different user-facing language or a GUI interface, then such a restriction would be easy to enforce. Additionally, the DiSL instrumentation framework is capable of dynamically adjusting which bytecode instructions should be instrumented.

As most queries are fairly simple and should not contain redundant components that are not actually part of the query, ie. the query equivalent of dead code, such a filter seems very promising. The main drawback of a filtered event stream is that we cannot easily regenerate previous events. The graph would be permanently reduced, which would limit the ability to change breakpoints during the execution.

That would mean that new or modified breakpoints cannot safely refer to past events as those events may have been pruned from the event stream. This would directly violate our breakpoint requirements as laid out in ‹Breakpoint Requirements› (section 3), namely the ability to define new breakpoints after a match. Additionally, due to the complexity of the Cypher query language, there is no guaranteed way to extract all properties to ensure that no indirect or obfuscated reference to a method name or other property is overlooked.

Due to those limitations, we chose not to implement an event filter for this thesis. As most of the test cases for the evaluation contain only the essential code necessary for the individual query under investigation, this will not significantly affect the results as described in ‹Evaluation› (section 5).

Note that our design allows for easy pruning of the graph by dropping older edges, effectively limiting the event history to be considered by a breakpoint query. That way, the memory use of the graph database can be bounded to some fixed number of events. This might benefit particularly long-running programs. In fact, old edges could be saved to disk for later analysis and simultaneously pruned from the memory database, so that a full execution trace could be recovered afterwards.

Pruning the graph would limit the ability of queries to access the event history. While this may be an acceptable tradeoff in general, we chose to not to do so in favor of maximizing the expressive power of the queries that can be formulated in the graph backend. More practice-oriented future work might certainly enforce some kind of maximal graph size.

4.3.4 Static Pre-Generation

As described in the previous chapter ‹Modelling the Event Stream› (section 4.1), we encode the dynamic attributes of events through immutable edge properties, and model the particular control flow of the execution through temporal annotations on those edges. Therefore, all the other information in the graph, in particular all the nodes, are statically known and unchanging between executions.

Because of this design choice, it is possible to pre-generate the nodes and even all possible edges between them using conventional source code analysis and derive a large part of the graph from the abstract syntax tree of the source code. Building the syntax tree is a necessary step of program compilation regardless and would not itself impose an additional overhead for the typical debugging scenario.

The number of nodes of the graph is bounded and known in advance. It is at most the total number of methods, variables, exceptions and classes in the program. We did not perform a comprehensive survey, but previous work [ATT16] suggests an average number of about 5000 functions for larger open source projects, implying a likely total number on the order of one million nodes for even very large projects.

Note that the resulting graph may include nodes that will not be reached during a particular execution. This optimization is therefore a tradeoff between total memory use and the runtime overhead of adding nodes to the graph.

For better comparison with previous approaches and to limit the scope of this thesis, we did not add a static analysis framework to our breakpoint backend. As the DiSL instrumentation framework does not expose the necessary information about the program, we chose to simulate this optimization pass by running the program once in a warmup mode that collects all nodes and their static properties, and then re-running the program with those nodes preserved but all edges removed, thereby measuring the performance characteristics of a graph with pre-generated nodes. That way, we can still analyze the potential benefits of such a pre-processing step.

It is also important to keep in mind that the Neo4j graph database has not been specifically optimized for graphs with a finite number of nodes known in advance and a list of edges that is only ever expanded, but whose members are never modified or deleted. Specialized graph representations (cf. ‹Graph Databases› (section 2)) might be of great benefit, but go against the design goal of using a mature general purpose graph database.

4.3.5 Indices

Indices are a major component of graph performance optimization.

It might seem promising at first to add indices for node or edge properties, as they are commonly accessed in queries. However, because of the nature of the incremental matches that must be anchored to the most recent event, nearly all queries only check the immediate neighborhood of the anchor node. Neo4j already uses direct pointers[Bes+23] to optimize accessing neighboring nodes, so indices would not actually benefit this use case.

We found no performance improvement from the use of property indices. Additionally, as described earlier in ‹Embedded Operation› (section 4.3.1), we used an internal hash map to efficiently look up nodes based on their name property when constructing the graph, eliminating another potential use of indices.

Some particularly complex queries, particularly queries that reach back far in the event history, may benefit from property indices. We chose to reduce the overall memory use and performance overhead of the graph database by avoiding the use and maintenance of indices instead of optimizing for a niche use case.

Previous work on temporal graphs has explored the possibility of indices for paths and (sub-)graphs (cf. [Kui+22]). It is still an open research question, however.

We did not investigate the problem further as it did not seem like a particularly worthwhile optimization to us, as typical queries construct fairly small paths and rarely leave the close neighborhood of the anchor node. Retrieving path information from an index would be unlikely to speed up the query rather than just computing it from scratch. Additionally, there would be a considerable memory overhead for such an index. Only queries that might span a long distance across the graph or which are otherwise expensive to compute, such as when finding the shortest path in a transportation network or when performing global searches, might benefit from a path index.

5 Evaluation

5.1 Performance

For our performance evaluation, we used the same test cases as [Dor21] for better comparison with previous results and converted all queries to the new Cypher backend. Overall a set of nine artificial code samples was used to cover all event types and attributes, including nested method calls and the combinations of queries through logical operators (AND and OR).

In addition to small test cases, we also generated an artificial call graph to simulate a complex program with a large number of method calls. The call graph was generated with a recursive graph generator using random typing as described in [AF09]. This approach generates graphs with realistic graph characteristics and was easy to tune to our needs. The basic implementation of the call graph generator is included in the appendix (‹Recursive Realistic Graph Generator using Random Typing› (appendix A)).

Additionally, to avoid infinite recursion or the exhaustion of the Java callstack, we converted all methods to finite recursion by passing an explicit counter to every method call. Each generated method decreases this counter as the first instruction and immediately returns if the counter reaches zero. It also passes this counter to any other method it might invoke itself. That way, we can limit the total call depth of any of our artificial methods.

We would've preferred to evaluate the performance with a realistic test case, for example by running the full test suite of a large open source program or by using a comprehensive test suite such as the Renaissance Suite[Pro+19]. Unfortunately, we could not employ such test suites in the current implementation without major changes due to the interactions between the DiSL-based instrumentation and its support for multi-threaded programs. Such an adaption would have been outside the scope of this exploratory work.

We performed all measurement on an AMD Ryzen 7 5700X 8-Core processor with 48GB of RAM running Linux 6.6.10, using Neo4j version 4.4.29 Community Edition and Java 11 through OpenJDK 11.0.23+9.

We compared the runtime performance of the null tracer (a dummy backend to factor out the overhead of the DiSL instrumentation framework), the custom query tracer developed in [Dor21] and the new graph-based backend. We separated the measurement of just the Neo4j-based graph backend concerned only with the construction of the event graph (figure 9 and table 2) from the measurement of the Cypher-based query engine (figure 10 and table 3), as their respective overheads differ greatly.

The memory use of the graph backend remained low throughout our tests. Even for the largest test case with 581k edges, the total graph size remained below 100MiB.

Without the use of merged static nodes, the number of nodes would scale roughly linear with the number of events, as only method calls act as branching nodes in the graph, requiring new nodes for any other event.

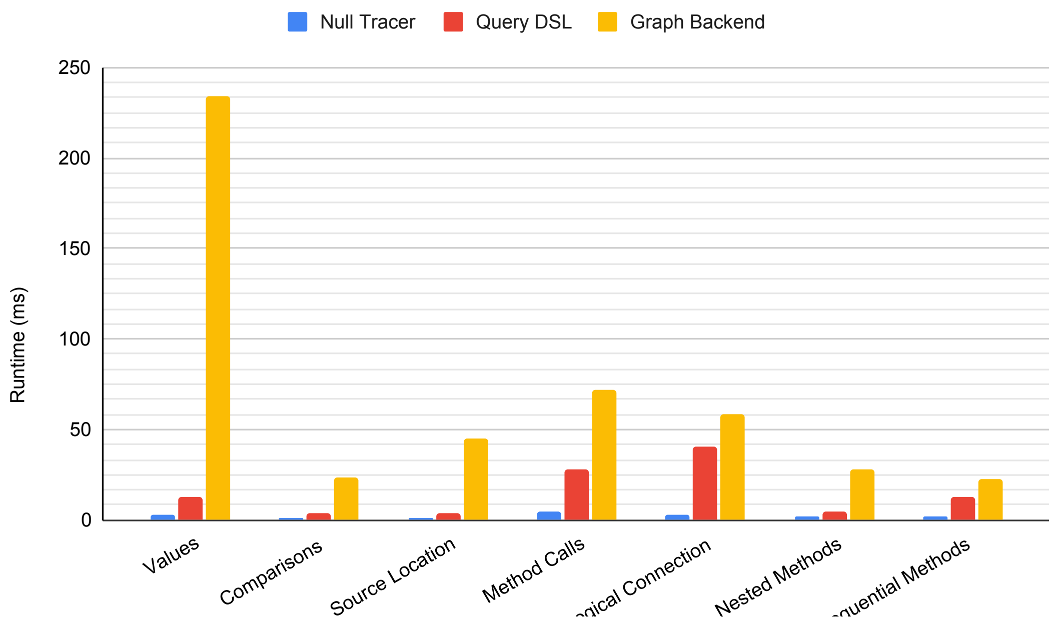


Figure 9 Backend Performance

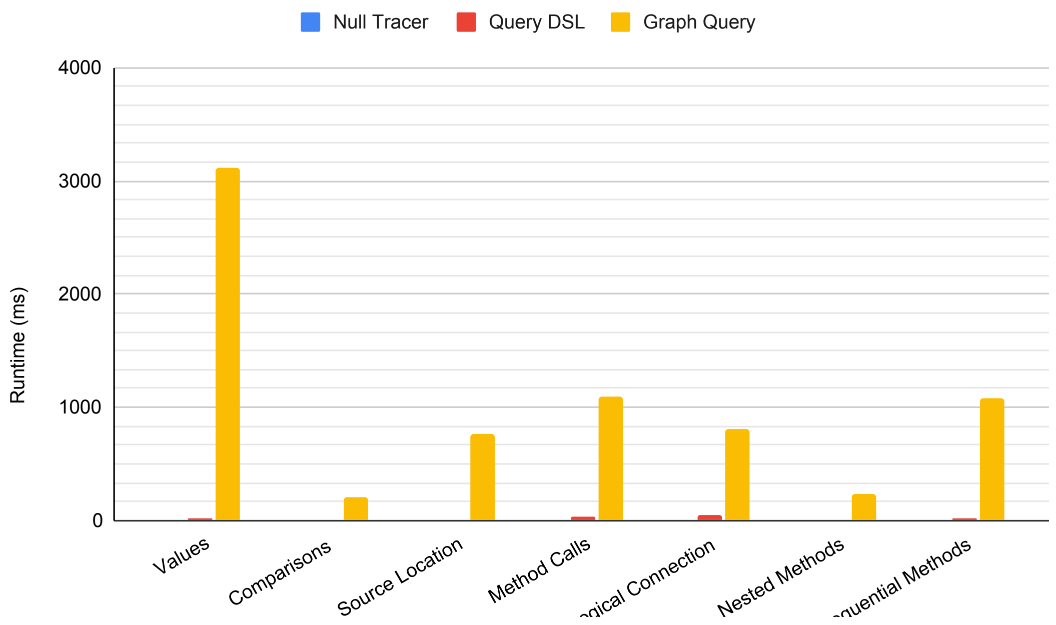


Figure 10 Query Performance

Table 2 Backend Performance (in ms)

Benchmark	Null Tracer	Graph Backend	Overhead vs Null
Values	2.6	234.1	90.4x
Comparisons	1.1	23.7	22.6x
Source Location	0.8	45.3	60.4x
Method Calls	4.5	71.8	15.8x
Logical Connection	2.6	58.7	22.9x
Nested Methods	1.7	27.7	16.7x
Sequential Methods	1.8	22.8	12.9x

Table 3 Query Performance (in ms)

Benchmark	Null Tracer	Query DSL	Graph Query	Overhead vs Null
Values	2.6	13.1	3,109.5	1,200.6x
Comparisons	1.1	4.0	201.4	191.8x
Source Location	0.8	3.9	756.7	1,009.0x
Method Calls	4.5	28.3	1,098.6	242.5x
Logical Connection	2.6	40.2	809.1	316.0x
Nested Methods	1.7	4.3	229.8	138.4x
Sequential Methods	1.8	12.8	1,081.6	611.1x

This can be seen in the following comparison for three different simulated call graphs of increasing size. Both ways to model the graph lead to the same number of edges as the number of event transitions is unchanged, but for the graph with merged nodes, the total number of nodes is independent of the actual execution trace and scales only with the number of entities in the program source code. This number is generally very low, 575 in our largest example with over 300,000 method calls (cf. table 4).

Table 4 Graph Size: Merged vs Unmerged Nodes

Benchmark	Edges	Merged Nodes	Unmerged Nodes
small	370	54	248
medium	13534	309	9024
large	581326	575	387552

Additionally, the required static information is necessarily collected during compilation for the construction of the abstract syntax tree and routinely exposed by various existing analysis tools such as Language Server Protocol tools.[Mic24] These tools are fast enough for interactive use in development environments, so making use of the same information for pre-generating the graph nodes would not add an appreciable overhead to the debugging work flow either. In fact, we found that Neo4j was easily capable of adding thousands of nodes in less than a second without any effort to optimize that aspect.

Because of the much more compact size of the resulting graph when using merged nodes, we also found a general performance improvement when matching queries of about 7x compared to unmerged nodes (table 5). The main advantage of using merged nodes is the reduced graph size, however, as the runtime difference is less pronounced when only the backend performance without the often drastic Cypher overhead is considered, as shown in table 6.

Table 5 Query Performance: Merged vs Unmerged Nodes (in ms)

Benchmark	Query Tracer	Merged	Unmerged	Unmerged Overhead
Values	13.1	3343.6	4070.6	1.2
Comparisons	4.0	225.1	645.4	2.9
Source Location	3.9	802.0	1344.0	1.7
Method Calls	28.3	1170.3	12855.6	11.0
Logical Connection	40.2	867.8	10294.1	11.9
Nested Methods	4.3	257.5	5351.3	20.8
Sequential Methods	12.8	1104.3	1552.3	1.4

Table 6 Backend Performance: Merged vs Unmerged Nodes (in ms)

Benchmark	Null Tracer	Merged	Unmerged	Unmerged Overhead
small	60.9	1131.8	1,554.2	1.4x
medium	387.7	18,509.5	26,571.2	1.4x
large	21,366.6	1,100,304.9	1,692,761.5	1.5x

The naive use of transactions for every new event added to the graph database led to completely unacceptable performance overheads, at least 1000x worse than our final implementation. We had to restrict transactions to a single global scope to achieve a runtime performance that could be usable for debugging, even if there is still a lot of room for further improvements.

The avoidance of transactions bypasses a lot of safety guarantees of the Neo4j graph database and works against the proper support of concurrency. Future improvements might introduce regular transaction checkpoints, trading limited safety against data corruption for unacceptable overheads.

Overall we found that the performance of the backend was within a factor of 10 of the highly optimized query backend developed in [Dor21]. This is a very promising result for a general purpose graph database and validates its use.

However, the performance of the Cypher query engine was much worse with an overhead of approximately 100x compared to the optimized backend. A large part of that overhead is due to the complexity of the query engine that is not tuned for our very low latency use case of many incremental searches anchored against one particular point of the graph instead of searches that investigate the whole graph from a more global perspective.

A custom traversal function would likely improve our results dramatically, but would also break compatibility with the existing Cypher query engine. Overall this rules out the straightforward use of Cypher as a breakpoint query language through the existing Neo4j architecture, even though the design of the query language itself was not a limiting factor.

5.2 Expressive Power and Ergonomics

All basic breakpoint queries, ie. those that match only against a single event or its attributes, could be represented in a very straightforward way with Cypher. The resulting queries were often comparable in conciseness and readability to the custom query language developed in [Dor21], even though they exposed the underlying graph structure of the breakpoint backend directly to the query author.

More complex queries that correspond to a path in the graph were also easy to express and interpret. As demonstrated in an earlier example in ‹Modelling Breakpoint Queries› (section 4.2), to match the calling of a method use at some indefinite time before a call to init could be expressed in the following way:

```
MATCH p =
  ()-[:CALL]->(x:METHOD {name: "use"})
  -[*]->()->
  [:CALL {time: $time}]->(y:METHOD {name: "init"})
RETURN p
```

Listing 10 A Complex Query

We did not need to extend the Cypher language in any way, preserving full compatibility. Only some of the more sophisticated relational operations such as logical connections did not have obvious representations in our version of Cypher, as they required matching against multiple independent sub-queries. Nonetheless, starting with Neo4j version 5, Cypher supports the use of sub-queries, which would likely improve the readability further.[Neo24b]

Because incremental queries need to be anchored to the most recent event, there is an additional burden on the query author to include the anchoring property comparison in the query. This restriction makes the query language less suitable for casual users, as the probability of making a mistake is too high. As we're not targetting a user-facing query language within the scope of this thesis, we do not consider this a major limitation. If the query is constructed automatically through the development environment or (as in [Dor21]) by compilation from a more convenient domain-specific language, then this constraint is easy to enforce without any explicit intervention by the user.

Ensuring the correct execution order of any potential match was accomplished through a temporal ordering constraint, such that all edges along the found path must increase monotonically in their time annotation. This restriction was easy to implement and did not impose any major performance overhead for typical queries.

The enforcement of the constraint does not create an additional cost for the query author, but due to its late application after the Cypher query engine, it can lead to unnecessary performance overheads due to generation of false matches that later have to be dismissed. A modified traversal function would not require any such late check and so this overhead could be entirely avoided in future work.

5.3 Ease of Implementation

The implementation of the new breakpoint backend required only minimal effort besides the necessary optimizations mentioned above. Most of the implementation effort went into ways to bypass the use of transactions because their frequent use created too much latency for a debugging backend.

We could not use the less low-level API provided by the Neo4j Java driver or an external Neo4j server. The enforced use of transactions for every interaction and the overhead of a network protocol for communication with the graph database led to completely unacceptable performance. We had to implement the breakpoint backend using the low-level embedded API, which exposed enough of the database details to reach a more acceptable latency.

The use of a mature general purpose graph database allowed the reuse of many existing tools, such as visualizations and additional graph analysis tools. Neo4j's standard visualization tools in particular, available through Neo4j Browser[Neo24d] and Neo4j Desktop[Neo22], greatly simplified the implementation of the breakpoint backend and helped with the debugging of queries.

5.4 Concurrency

Finally, we added basic support for concurrency through the use of edge annotations that can be used to enforce paths limited to only one strand of the program. While this is an improvement over previous work, it is still a far cry from full concurrency.

The necessary use of a global transaction scope conflicts heavily with concurrent access to the graph database. As discussed, explicit event node for the forking and merging of strands would provide a more natural way to divide up updates to the graph by multiple threads in the underlying backend. Some of the existing research into snapshot-based graphs seems like a promising avenue to us for developing a better solution, as the fundamental problem of how to share data between multiple complete versions of a graph within a graph database is closely related to how to allow concurrent updates and query searches.

6 Related Works

First we will discuss the most relevant overview papers that cover the existing design space of current graph database and graph query languages. Then we will present the most important research in temporal graphs that influenced our thesis. Lastly, we will touch on a few works about graph visualization for graph databases, as that is an important additional topic for developing better debugging tools that utilize graphs as their underlying abstraction of the execution stream.

6.1 Graph Databases

[Ang+17] offers a great overview of the underlying abstractions and core distinguishing features of modern graph databases and graph query languages. Angles et al. lay out the most important graph models in modern use. They also develop an ontology for graph query language, including an overview of the three most popular languages SPARQL, Gremlin, and Cypher.

A similar but overall more basic overview can be found in [Ang12]. Despite a focus on older systems and limitations that may no longer apply in the most recent iteration of the graph databases discussed, it acts as a good introduction to the design space and basic distinguishing features of graph databases and their query languages.

An important companion piece is Besta et al.'s [Bes+23], which provides further details on the variety of graph database backend designs and an extensive breakdown of dozens of current graph databases by features such as their data representation, data organization and data distribution models, as well as the supported graph query languages and concurrent execution modes. Besta et al. develop a comprehensive ontology of graph databases that are of great help for organizing the large amount of different systems in use today. Our thesis closely the ontology established in their paper closely.

Lastly, [Woo12] focuses on primarily on the formal characteristics of graph query languages and the kind of queries supported by them. While few of the languages discussed are in common use, the ontology and analysis of path queries, as mentioned in ‹Graph Query Languages› (section 2.4), is still an important aspect of query language design today.

Additionally, all of those works provide extensive lists of references that span the entire range of fundamental research in the domain of graph databases and which are therefore invaluable for more detailed research.

Finally, [Bes+21] offers a comparable overview for graph streaming frameworks that may offer interesting insights for alternative approaches to breakpoint backends that nonetheless follow a basic graph design. Graph streaming frameworks are designed for systems that need to analyze and query the underlying graph concurrently while processing large live updates to the graph, often on the scale of millions of graph updates per second or more. While we found streaming frameworks too poor in their expressiveness in our exploration, further research into the viability of systems focused much more on throughput and dynamic updates may yield important insights for debugging designs as well.

6.2 Temporal Graphs

6.2.1 TGraph / T-GQL

The most influential research of temporal graphs for our thesis were [Deb+21] and [Sol22].

In [Deb+21], Debrouvier et al. develop a new temporal graph query language T-GQL by extending the Cypher query language with operators for selecting matches based on the temporal annotation in the graph as well as new functions for calculating paths through the graph that obey certain temporal constraints.

They extend the property graph model with attribute and value nodes (cf. ‹Temporal Graphs› (section 2.3)) to represent updates to properties that are annotated with the time of the update, solving the lack of nested properties as we discussed earlier. Additionally, they formulate important constraints for paths in temporal graphs. First, paths may be required to be valid over an interval such that all the nodes and edges included in the path are each valid in the same interval, ie. the path is continuous. They also establish the concept of pairwise continuous paths, which offers a weaker constraint that merely requires that each pair of consecutive edges overlap in the intervals defined for them. Lastly, they establish a similar constraint for temporal graphs labelled with durations instead of intervals, called consecutive paths, which can be used to represent earliest-arrival, latest-departure, fastest, and shortest paths.

Debrouvier et al. also evaluate a prototype of their system, based on the Neo4j graph database, to show that the temporal extension has acceptable performance and is a plausible direction for future research.

[Sol22] offers an overview of multiple papers based on T-GQL and its application to various domains. In particular, [KSV22] attempts to apply their temporal graph approach to sensor-equipped transportation networks and extend it to support time series.

Finally, [Kui+22] proposes an index structure to improve the calculation of continuous paths in a temporal graph. Indexing of path and sub-graph queries, in particular for temporal graphs but also more broadly graph databases as a whole, is still a very open research question. Kuijpers, Soliani et al. offer a good introduction to a typical problem in the design space.

6.2.2 T-Cypher / GDBAlive

A similar approach to temporal graph databases and graph query languages is T-Cypher, as described in [Mas22]. Massri's doctoral thesis offers comprehensive coverage of the topic of temporal graph design with a focus on the domain of Internet-of-Things systems. Developed as an extension to Cypher to make temporal operations more concise, Massri also stresses the importance of compatibility with existing graph databases to improve adoption in real-world systems.

Overall, Massri's thesis provides a highly valuable view of the many components involved in a practical temporal graph system, covering the architecture of temporal graph management, the design of the necessary graph query operations and their integration into an existing query language, as well as efficient storage solutions and query backends for temporal graphs.

Finally, Massri also presents a temporal graph generator to help evaluate the performance of various approaches through the use of realistic and large datasets. The development and integration of realistic test cases of sufficient size continues to be a major limitation of our work in developing breakpoint backends, with this thesis offering the first preliminary attempt to improve the situation by relying on recursive graph generators using random typing (cf. [AF09] and ‹Evaluation› (section 5)).

[MRM20] is a related older approach for extending an existing columnar data store to support temporal graphs. GDBAlive can offer an interesting perspective on how to reuse an existing mature graph database, in this case Cassandra, to support temporal queries and annotations.

6.2.3 Other Notable Approaches

ChronoGraph, as described in [BWK20], is a traversal-oriented system that supports point-based and interval-based temporal annotations. Byun et al. develop new algorithms to calculate breadth-first, depth-first and shortest paths in a temporal graph, with a focus on efficient and parallelized computation of these paths. Unlike the previous approaches discussed so far, ChronoGraph is based on the imperative graph query language Gremlin and its prototype is designed for the TinkerPop graph computing framework.

The concept of a shortest path in a temporal graph is further explored in [Wu+16]. Convention graph algorithms such as Dijktra's greedy algorithm for computing the shortest path cannot be used in a temporal graph, Wu et al. develop new definitions of what a shortest temporal path may mean, and explore a range of efficient algorithms for computing them. They also provide an evaluation of their approach using real-world temporal graphs.

While our thesis focused primarily on property graphs, there have also been attempts to extend edge-labelled graphs with temporal annotations, in particular RDF graphs and its standardized SPARQL query language.

One of the earliest works is the TSQL query language and its later revision TSQL2, as described in [Sno12], which seek to extend the common SQL query language. Later on, [Gra10] extends many of TSQL2's features to the SPARQL query language used for RDF graphs.

Finally, [DG+09] proposes an extension to represent temporal data in XML so that changes to XML documents can be timestamped and their history preserved. They also extend the XPath query language for XML in appropriate ways.

We ultimately pursued none of those approaches further, as we were constrained by the use of the Neo4j graph database, which seemed to best fit our requirements of ease of implementation and adoption on the one hand, and performance and suitability for temporal queries on the other.

6.3 Visual Query Languages

6.3.1 Knowledge Graph Visual Query Language)

[Liu+22] is a particularly promising visual query language developed as an extension for the SPARQL query language. The most remarkable feature of KGVQL is its bidirectionality between visual graph queries and graph results. This allows users to construct queries incrementally and in a fully visual way, avoiding the need to learn a complex textual query language. By retaining a bidirectional mapping between parts of the graph query its result, KGVQL can help users build up complex queries incrementally and through interactive exploration.

Liu et al. also evaluate the validity of their design by performing a small-scale user study, showing that KGVQL is superior to SPARQL and other established graph query languages when it comes to the construction of moderately complex queries.

Finally, Liu et al. aslo provide a very useful overview of other visual graph query languages.

6.3.2 TGV

[Orl+23] describes the design and implementation of a visualization tool for the temporal graph presented in [Deb+21] (cf. ‹TGraph / T-GQL› (section 6.2.1)). In particular, Orlando et al. provide tools to show the evolution of the graph or paths within it over time my letting the user modulate the time dimension using a slider bar. This way, query results can be investigated as they change over time, an important feature absent from conventional graph visualization tools.

Additionally, TGV abstracts away distracting aspects of the underlying graph structure necessary to model the temporal annotations. This simplifies the interaction with the graph for the user who does not have to concern themselves with all the implementation details of the graph model.

Finally, Orlando et al. also demonstrate the usefulness of using an established graph database for the development of new abstractions and tools. This is of course a major reason we considered the use of Neo4j in the first place.

7 Conclusion

The runtime overhead of the general purpose graph database Neo4j is considerable for our low latency requirements, even after significant optimizations. Still, the performance cost of the construction of the event graph was within a factor of 10 of the highly optimized query backend of [Dor21], which is still within an acceptable range, especially considering the large amount of existing visualization and analysis tools available for Neo4j.

Because some of the optimizations bypass the transactional safety and because the Cypher query engine has an inherent latency that is unsuitable for live debugging, a different graph database implementation or at least a custom traversal function would be justified. As a breakpoint backend benefits less from expensive features like full ACID safety, complex diagnostics or distributed database use, implementation burden of such a custom approach would be more manageable for a debugging tool. Alternatively, more light-weight graph databases, as mentioned in [Ang+17] and others, would be worth investigating.

A modified traversal function would also simplify the query language by removing the need to explicitly include temporal and anchoring constraints in the query, but such a modification is unfortunately not compatible with the existing Cypher query engine infrastructure. Additionally, such a modification would dramatically improve the live debugging performance as queries would only have to attempt a match against the most recent addition to the graph instead of approaching the search problem from a global perspective as is more typical for graph database queries.

Otherwise the Cypher query language was found to be very expressive and powerful and a good match overall for the specification of breakpoints. More sophisticated relational operations such as logical `AND` only required minor workarounds or extensions of the breakpoint backend. Other query languages are nonetheless worth investigating in future research.

A more comprehensive benchmark suite or realistic test case would be valuable for future evaluations, but the basic evaluation we performed already showed important bottlenecks and weaknesses to be addressed in future work. The lack of a more comprehensive benchmark therefore did not influence this thesis.

Our implementation provided basic support for modelling concurrent execution in the event graph, but is still lacking when it comes to handling more realistic multithreaded execution.

Overall we found modelling the event stream within the Neo4j graph database a very powerful and promising approach and therefore consider this exploration a success. The Cypher query language was a good fit for breakpoints with straightforward representations of all common breakpoint patterns and with useful ways to compose them and construct more sophisticated queries. The available visualization tools in particular were a great benefit to the development of the new breakpoint backend. Nonetheless, the latency of the Cypher query engine in considerable and will have to be addressed in future work before a graph-based implementation may be considered for live debugging.

7.1 Outlook

The most important limitation of our graph-based implementation was the performance of the query engine. Future research should explore alternative graph database implementations and query engines. In particular, a graph database that is more limited in scope and possibly developed with very low latency in mind seems like a promising direction for extensions of our work.

The extensive ecosystem of tools for the Neo4j graph database and other mature systems also opens up new venues for debugging tools. In particular, visualizations and graph analytics might help to improve the debugging workflow. The use of a temporal graph backend might also allow integration with time-travelling debuggers and performance profilers.

Proper modelling of concurrent execution in the event stream and the representation of breakpoint queries for concurrent programs is still an unsolved problem. As our exploration has shown, graph databases provide a promising base for future research, although the problem of how combine transactional updates of the graph with the necessary low latency for live debugging is still an open question.

Furthermore, ongoing research in temporal graphs might provide additional insights. In particular, snapshot-based approaches with shared graphs might function as a future base for representing concurrency in the event stream.

Lastly, several limitations of our previous work still have no satisfactory solution, such as the representation of complex values, especially with regards to mutability, and better instrumentation frameworks and bytecode models.

Overall, identifying or developing a better query engine with lower latency for anchored incremental searches appears be to the most promising direction for further developments, in addition to offering a better base for supporting concurrency.

8 References

[AF09] Leman Akoglu and Christos Faloutsos. “RTG: A recursive realistic graph generator using random typing.” In: ***Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2009, Bled, Slovenia, September 7-11, 2009, Proceedings, Part I 20***. Springer. 2009, pages 13–28.

[Ang+17] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. “Foundations of Modern Query Languages for Graph Databases.” In: ***ACM Comput. Surv.*** 50.5 (Sept. 2017). ISSN: 0360-0300. DOI: 10.1145/3104031.

[Ang12] Renzo Angles. “A Comparison of Current Graph Database Models.” In: ***2012 IEEE 28th International Conference on Data Engineering Workshops***. 2012, pages 171–177. DOI: 10.1109/ICDEW.2012.31.

[ATT16] Saleh Alnaeli, Amanda Taha, and Tyler Timm. “On the Prevalence of Function Side Effects in General Purpose Open Source Software Systems.” In: volume 656. June 2016, pages 141–148. ISBN: 978-3-319-33902-3. DOI: 10.1109/SERA.2016.7516139.

[Bes+21] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. ***Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems***. 2021. arXiv: 1912.12740 [cs.DC].

[Bes+23] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. ***Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries***. 2023. arXiv: 1910.09017 [cs.DB].

[Boc+21] Christoph Bockisch, Stefan Schulz, Viola Wenz, and Arno Kesper. “A unifying approach to breakpoint specification.” In: ***24th Iberoamerican Conference on Software Engineering, CIbSE 2021, San Jose, Costa Rica, August 20 - September 3, 2021***. Curran Associates, 2021, pages 234–247.

[BWK20] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. “ChronoGraph: Enabling Temporal Graph Traversals for Efficient Information Diffusion Analysis over Time.” In: ***IEEE Transactions on Knowledge and Data Engineering*** 32.3 (2020), pages 424–437. DOI: 10.1109/TKDE.2019.2891565.

[Cha02] Donald D. Chamberlin. “XQuery: An XML query language.” In: ***IBM Syst. J.*** 41 (2002), pages 597–615. URL: https://api.semanticscholar.org/CorpusID:9942318.

[Deb+21] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. “A Model and Query Language for Temporal Graph Databases.” In: ***The VLDB Journal*** 30.5 (May 2021), pages 825–858. ISSN: 1066-8888. DOI: 10.1007/s00778-021-00675-4.

[Deo16] N. Deo. ***Graph Theory with Applications to Engineering and Computer Science***. Dover Books on Mathematics. Dover Publications, 2016. ISBN: 9780486807935. URL: https://books.google.de/books?id=uk1KDAAAQBAJ.

[DG+09] Curtis Dyreson, Fabio Grandi, et al. "Temporal xml." In: ***Encyclopedia of database systems***. Springer-Verlag, 2009, pages 3032–3035.

[Dor21] Freya Dorn. "An Efficient Domain-Specific Language For Breakpoints." Bachelor's Thesis. Philipps-Universität Marburg, Dec. 2021.

[Dre23] Teresa Dreyer. "An efficient formalism for advanced breakpoints." Master's Thesis. Philipps-Universität Marburg, Jan. 2023.

[Fra+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. "Cypher: An evolving query language for property graphs." In: ***Proceedings of the 2018 international conference on management of data***. 2018, pages 1433–1445.

[Gra10] Fabio Grandi. "T-SPARQL: A TSQL2-like temporal query language for RDF." In: volume 639. Jan. 2010, pages 21–30.

[GS04] Fabien Gandon and Guus Schreiber. ***RDF/XML Syntax Specification***. 2004. URL: https://www.w3.org/TR/REC-rdf-syntax/ (visited on 2024-05-27).

[Hay+04] Jonathan Hayes et al. "A graph model for RDF." In: ***Darmstadt University of Technology/University of Chile*** (2004).

[HR83] Theo Haerder and Andreas Reuter. "Principles of transaction-oriented database recovery." In: ***ACM Comput. Surv.*** 15.4 (Dec. 1983), pages 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: https://doi.org/10.1145/289.291.

[KSV22] Bart Kuijpers, Valeria Soliani, and Alejandro Vaisman. "Modeling and Querying Sensor Networks Using Temporal Graph Databases." In: ***New Trends in Database and Information Systems***. Cham: Springer International Publishing, 2022, pages 222–231. ISBN: 978-3-031-15743-1.

[Kui+22] Bart Kuijpers, Ignacio Ribas, Valeria Soliani, and Alejandro Vaisman. "Indexing Continuous Paths in Temporal Graphs." In: ***New Trends in Database and Information Systems***. Cham: Springer International Publishing, 2022, pages 232–242. ISBN: 978-3-031-15743-1.

[Liu+22] Pengkai Liu, Xin Wang, Qiang Fu, Yajun Yang, Yuan-Fang Li, and Qingpeng Zhang. "KGVQL: A knowledge graph visual query language with bidirectional transformations." In: ***Knowledge-Based Systems*** 250 (2022), page 108870. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2022.108870. URL: https://www.sciencedirect.com/science/article/pii/S0950705122004154.

[Mar+12] Lukáš Marek, Alex Villazon, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. "DiSL: A domain-specific language for bytecode instrumentation." In: ***AOSD'12 - Proceedings of the 11th Annual International Conference on Aspect Oriented Software Development*** (Mar. 2012). DOI: 10.1145/2162049.2162077.

[Mas22] Maria Massri. "Designing a temporal graph management system for IoT application domains." Theses. Université de Rennes, Dec. 2022. URL: https://inria.hal.science/tel-04071498.

[Met24] MetaBrainz Foundation. ***MusicBrainz - the open music encyclopedia***. 2024. URL: https://musicbrainz.org/ (visited on 2024-05-27).

[Mic24] Microsoft. ***Language Server Protocol***. 2024. URL: https://microsoft.github.io/language-server-protocol/ (visited on 2024-05-27).

[MRM20] M. Besher Massri, Parvedy Philippe Raipin, and Pierre Meye. “GDBAlive: A Temporal Graph Database Built on Top of a Columnar Data Store.” In: ***Journal of Advances in Information Technology*** (2020). URL: https://api.semanticscholar.org/CorpusID:225018562.

[Neo22] Neo4j, Inc. ***Neo4j Desktop***. 2022. URL: https://neo4j.com/docs/desktop-manual/current/.

[Neo24a] Neo4j, Inc. ***Bolt Protocol***. 2024. URL: https://neo4j.com/docs/bolt/current/bolt/ (visited on 2024-05-27).

[Neo24b] Neo4j, Inc. ***Cypher Subqueries***. 2024. URL: https://neo4j.com/docs/cypher-manual/5/subqueries/ (visited on 2024-05-27).

[Neo24c] Neo4j, Inc. ***Neo4j***. 2024. URL: https://neo4j.com/ (visited on 2024-05-27).

[Neo24d] Neo4j, Inc. ***Neo4j Browser***. 2024. URL: https://neo4j.com/docs/browser-manual/current/ (visited on 2024-05-27).

[Orl+23] Diego Orlando, Joaquín Ormachea, Valeria Soliani, and Alejandro Ariel Vaisman. “TGV: A Visualization Tool for Temporal Property Graph Databases.” In: ***Information Systems Frontiers*** (Aug. 2023). ISSN: 1572-9419. DOI: 10.1007/s10796-023-10426-1. URL: https://doi.org/10.1007/s10796-023-10426-1.

[Pro+19] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. “Renaissance: benchmarking suite for parallel applications on the JVM.” In: ***Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation***. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pages 31–47. ISBN: 9781450367127. DOI: 10.1145/3314221.3314637. URL: https://doi.org/10.1145/3314221.3314637.

[Rod15] Marko A. Rodriguez. “The Gremlin graph traversal machine and language (invited talk).” In: ***Proceedings of the 15th Symposium on Database Programming Languages***. DBPL 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pages 1–10. ISBN: 9781450339025. DOI: 10.1145/2815072.2815073. URL: https://doi.org/10.1145/2815072.2815073.

[Sno12] Richard T Snodgrass. ***The TSQL2 temporal query language***. Volume 330. Springer Science & Business Media, 2012.

[Sol22] Valeria Soliani. “Models and Query Languages for Temporal Property Graph Databases.” In: ***New Trends in Database and Information Systems***. Cham: Springer International Publishing, 2022, pages 623–630. ISBN: 978-3-031-15743-1.

[Stu21] Stu Moore. ***Neo4j 4.3 Blog Series: Relationship Indexes***. 2021. URL: https://neo4j.com/developer-blog/neo4j-4-3-blog-series-relationship-indexes/.

[Wik24] Wikimedia Foundation, Inc. ***Wikidata***. 2024. URL: https://www.wikidata.org/wiki/Wikidata:Main_Page (visited on 2024-05-27).

[Woo12] Peter T. Wood. “Query Languages for Graph Databases.” In: ***SIGMOD Rec.*** 41.1 (Apr. 2012), pages 50–60. ISSN: 0163-5808. DOI: 10.1145/2206869.2206879. URL: https://doi.org/10.1145/2206869.2206879.

[Wu+16] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. “Efficient Algorithms for Temporal Path Computation.” In: ***IEEE Transactions on Knowledge and Data Engineering*** 28.11 (2016), pages 2927–2942. DOI: 10.1109/TKDE.2016.2594065.

[Yil+18] Bugra M. Yildiz, Christoph Bockisch, Arend Rensink, and Mehmet Aksit. “A Java Bytecode Metamodel for Composable Program Analyses.” In: ***Software Technologies: Applications and Foundations***. Edited by Martina Seidl and Steffen Zschaler. Cham: Springer International Publishing, 2018, pages 30–40. ISBN: 978-3-319-74730-9.

Appendix

A Recursive Realistic Graph Generator using Random Typing

```ruby
require "numo/narray"

def random_graph num_edges, num_chars: 5, beta: 0.1, prob_space: 0.5
  # based on RTG: A Recursive Realistic Graph Generator using Random Typing by
    ↪ Akoglu et al.
  chars = ('a'..'z').take(num_chars) + [" "]
  char_combi = chars.product(chars).to_a

  # generate keyboard
  keyboard = []
  prob = Numo::DFloat.zeros(num_chars + 1)
  prob_remaining = 1 - prob_space
  (num_chars - 1).times do |i|
    prob[i] = rand() * prob_remaining
    prob_remaining -= prob[i]
  end
  prob[num_chars - 1] = prob_remaining
  prob[num_chars] = prob_space

  keyboard = prob.outer(prob) * beta
  keyboard.diagonal.fill(0)
  remaining_diag = prob - keyboard.sum(axis: 0)
  keyboard[keyboard.diag_indices] = remaining_diag
  cdf = keyboard.cumsum

  # generate edges
  edges = Hash.new{|h,k| h[k] = Hash.new(0)}

  num_edges.times do
    src_finished = false
    dst_finished = false
    src = ""
    dst = ""

    while not (src_finished and dst_finished)
      s, d = char_combi.weighted_sample(cdf)

      if s == " "
        src_finished = src.length > 0
      elsif not src_finished
        src += s
      end

      if d == " "
        dst_finished = dst.length > 0
      elsif not dst_finished
        dst += d
      end
    end

    edges[src][dst] += 1
  end

  edges
end
```