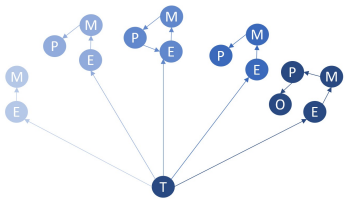
TEMPO

Management and Processing of Temporal Networks

H.F.R.I. Project No. 03480

D3.2 - Final Version of the Query Engine Module



Computer Engineering & Informatics Department
University of Patras
Greece
30/11/2025

D3.2 - Final Version of the Query Engine Module

Alexandros Spitalas and Kostas Tsichlas

November 30, 2025

Abstract

This documents reports on the final version of the query engine module regarding the end of the research project TEMPO, since the query engine will continue to evolve. We describe the temporal graph query language T-Gremlin that is based on Gremlin. Our T-Gremlin implementation is available as an open-source fork of Apache TinkerPop at `https://github.com/alexspitalas/T-tinkerpop/tree/temporalTinkerpop`, enabling the broader research community to build upon our work and contribute further enhancements.

1 Introduction

When integrating temporal functionality into graph databases, several critical factors must be carefully considered: the query language used to interact with these systems, the indexing strategies employed, the underlying database technology for data storage, and the fundamental data structures that support temporal operations.

JanusGraph utilizes Gremlin as its primary graph query language, which is an open-source graph traversal language developed by Apache TinkerPop. Gremlin employs a functional, data-flow approach for querying databases and follows the property graph model. As a graph traversal language, Gremlin is specifically designed to succinctly express complex traversals on property graphs, where every traversal is composed of a sequence of potentially nested steps. Each step in Gremlin performs an atomic operation on the data stream, functioning as either a map-step (transforming objects), a filter-step (removing objects), or a side effect-step (computing statistics).

To address the need for temporal graph capabilities, we have forked and extended TinkerPop Gremlin to create T-Gremlin, elevating time as a first-class citizen in the graph query language. This extension recognizes that traditional static graph traversal languages are insufficient (but not incapable) for analyzing temporal patterns and information diffusion over time. The concept of adding temporal capabilities to Gremlin has been explored in academic research, particularly in the ChronoGraph [2] project, which discusses modifications to transform steps and filter steps to accommodate temporal semantics.

Examining the ChronoGraph implementation more closely reveals important insights about the current state of temporal graph extensions. The ChronoGraph project, available at their GitHub repository[1], demonstrates that while they have successfully implemented static property graphs using TinkerPop Blueprints - a foundational interface for property graph models - their implementation of a traversal engine for static property graphs using TinkerPop Gremlin remains incomplete and is currently marked as "TODO". This indicates that while the theoretical framework for temporal graph traversals exists, the practical implementation of extending Gremlin for temporal operations presents significant technical challenges.

TinkerPop Blueprints serves as the foundational API for property graph implementations, providing a collection of interfaces, implementations, and test suites for the property graph data model. It functions as the equivalent of JDBC for graph databases, offering a common set of interfaces that allow developers to work with different graph database backends interchangeably. Within the broader TinkerPop ecosystem, Blueprints historically served as the foundational technology supporting other components including Pipes (a data flow framework), Gremlin (the graph traversal language), Frames (an object-to-graph mapper), and other graph processing tools. For their research purposes, the ChronoGraph team utilized TinkerPop Blueprints as it provides the essential property graph interface needed to build temporal graph functionality, even though their full Gremlin integration remains a work in progress.

[1] Available at `https://github.com/dfpl/chronograph-practice`

2 Lifetime Properties: Time as First-Class Citizen

A fundamental contribution of T-Gremlin is the introduction of structured temporal properties that make time a first-class citizen across all graph elements. We have extended TinkerPop's property model with dedicated `lifetime` functionality for vertices, edges, and properties, providing a standardized approach to temporal data management.

Structured Temporal Property Model. T-Gremlin introduces a consistent temporal property schema where graph elements store their validity intervals using standardized property names:

- **Vertices:** Temporal vertices can maintain `startTime` and `endTime` properties representing their existence interval
- **Edges:** Temporal edges include validity intervals indicating when relationships are active.
- **Properties:** Individual properties can also have their own temporal scope, enabling fine-grained versioning.

Thanks to the above structured implementation of time, checks can be included so that the added edges will be valid, (e.g., to be inside the scope of the edge's vertices), and the same applies to properties.

The `lifetime()` Step Implementation. The `lifetime(startTimeKey, endTimeKey)` step acts as an encapsulation, in order to access the temporal intervals so as to set them, as well as get them:

```
// Example set of startTime and entTime for a vertex
g.addV("person").lifetime("2023-01-01", "2023-12-31").next();

// Example set of startTime for a vertex that has no endTime
g.addV("person").lifetime("2023-01-01").next();

// Example of add edge with temporal values.
g.addE("knows").from(V().has("name", "alice")).to(V().has("name", "bob")).lifetime
("2023-01-01", "2023-12-31")
```

This structured approach provides several key advantages:

- **Standardized Schema:** All temporal graph elements follow consistent naming conventions, enabling predictable query patterns.
- **Automatic Discovery:** Allen temporal operators can automatically locate `startTime` and `endTime` properties without manual specification.
- **Type Safety:** Temporal properties are strongly typed and validated during graph operations.
- **Composable Operations:** The `lifetime` step integrates seamlessly with existing Gremlin steps and temporal operators:

  ```
  // Complex temporal query composition
  g.V().has("type", "event")
      .lifetime("startTime", "endTime")
      .where(__.temporalOverlaps(milestone))
      .groupBy("category")
      .by(__.count())
  ```

- **Cross-Element Consistency:** Vertices, edges, and properties all use the same temporal property model, enabling uniform temporal reasoning.

The `lifetime` functionality is implemented as:

- A default method in the `GraphTraversal` interface following TinkerPop conventions.
- Full bytecode serialization support registered in `BytecodeHelper`.
- Anonymous traversal support in the `__` class for lambda compositions.
- Compatibility with all Gremlin Language Variants (Python, .NET, Go, JavaScript).

By establishing this structured temporal property model, T-Gremlin transforms time from an auxiliary attribute into an integral part of the graph data model. This foundation enables the sophisticated Allen temporal algebra operations while maintaining the simplicity and consistency that developers expect from the TinkerPop ecosystem.

2.1 Allen Temporal Algebra Integration

A fundamental challenge in temporal graph querying is the ability to express and evaluate temporal relationships between graph elements. Traditional graph query languages lack native support for reasoning about temporal intervals and their relationships. To address this limitation, we have implemented a comprehensive set of temporal operators based on Allen's Interval Algebra [1] within our T-Gremlin framework.

Allen's temporal algebra defines 13 mutually exclusive relationships that can exist between any two temporal intervals. These relationships provide a complete and unambiguous framework for temporal reasoning, making them particularly well-suited for integration into graph traversal languages. Our implementation extends TinkerPop Gremlin's filter step architecture to support all 13 Allen temporal relationships as first-class graph traversal operations.

2.1.1 Temporal Operator Architecture

The core of our temporal extension is built around a unified `AllenFilterStep<S,E>` class that extends TinkerPop's `FilterStep<S>` framework. This design decision ensures full compatibility with Gremlin's existing traversal pipeline architecture while providing efficient temporal filtering capabilities.

The implementation utilizes an enumerated `AllenRelation` type that encapsulates the 13 temporal relationships:

- **Sequential Relations:** `BEFORE`, `AFTER`, `MEETS`, `MET_BY`
- **Overlap Relations:** `OVERLAPS`, `OVERLAPPED_BY`
- **Containment Relations:** `DURING`, `CONTAINS`
- **Boundary Relations:** `STARTS`, `STARTED_BY`, `FINISHES`, `FINISHED_BY`
- **Identity Relations:** `EQUALS`

Each temporal relationship (depicted also in Figure 1) is implemented as a boolean evaluation function that compares the start and end times of temporal intervals extracted from graph elements. The system expects temporal properties to be stored as ISO 8601 formatted strings in properties named `startTime` and `endTime`, though the implementation includes flexible parsing mechanisms to accommodate various date-time formats.

2.1.2 GraphTraversal Integration

To provide a natural and discoverable API, we have extended the core `GraphTraversal` interface with dedicated methods for each Allen temporal relationship. This approach follows TinkerPop's established patterns for filter operations, ensuring consistency with existing Gremlin syntax and semantics. The temporal methods are implemented as default interface methods that internally delegate to the unified `AllenFilterStep`:

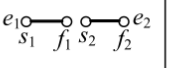
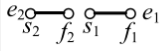
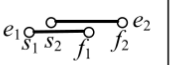
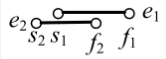
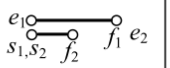
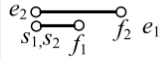
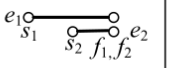
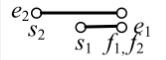
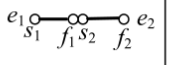
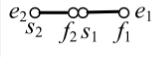
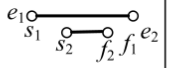
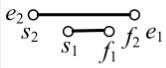
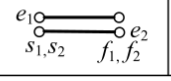
Relation code	Relation	Figure illustration	Relation code	Relation	Figure illustration
1	e_1 before e_2		8	e_2 before e_1	
2	e_1 overlaps e_2		9	e_2 overlaps e_1	
3	e_1 starts e_2		10	e_2 starts e_1	
4	e_1 finishes e_2		11	e_2 finishes e_1	
5	e_1 meets e_2		12	e_2 meets e_1	
6	e_1 contains e_2		13	e_2 contains e_1	
7	e_1 equals e_2				

Figure 1: Allen's 13 temporal relationships implemented in T-Gremlin. Each relationship defines a unique way two temporal intervals can relate to each other, providing a complete framework for temporal reasoning in graph queries.

```
default GraphTraversal<S, S> temporalBefore(final Element referenceElement) {
    this.asAdmin().getBytecode().addStep(Symbols.temporalBefore, referenceElement);
    return this.asAdmin().addStep(new AllenFilterStep<>(this.asAdmin(),
        AllenFilterStep.AllenRelation.BEFORE, referenceElement));
}
```

This pattern is replicated for all 13 temporal relationships, providing methods such as `temporalAfter()`, `temporalOverlaps()`, `temporalDuring()`, `temporalContains()`, and `temporalEquals()`. In Table 1, the complete set of Allen temporal operations is given as implemented in T-Gremlin.

This design provides several key advantages including: a) **Unified Architecture:** The single `AllenFilterStep` class reduces code duplication while maintaining the flexibility to optimize specific relationships in the future., b) **Type System Integration:** Proper generic type handling ensures compatibility with TinkerPop's complex type system and prevents runtime type errors, c) **Extensibility:** The enum-based approach allows easy addition of custom temporal relationships without modifying the core filter step implementation, d) **Standards Compliance:** Use of Allen's well-established temporal algebra ensures theoretical soundness and interoperability with other temporal reasoning systems, and e) **Performance Isolation:** Temporal operations do not impact the performance of non-temporal graph operations, maintaining backward compatibility. In Figure **??**, the integration of temporal operators with the Tinkerpop's traversal framework is shown.

2.1.3 Anonymous Traversal Support

Following TinkerPop's anonymous traversal patterns, we have extended the `__` (double underscore) class with static methods for all temporal operations. This enables the use of temporal filters within complex traversal compositions and lambda expressions:

```
// Find events that occur during a reference timeframe
g.V().where(__.temporalDuring(referenceEvent)).values("name")

// Complex temporal query with multiple conditions
g.V().where(__.or(
    __.temporalBefore(milestone),
    __.temporalOverlaps(milestone)
)).has("type", "task")
```

Operator	Allen Relation	Description
`temporalBefore()`	BEFORE	Element X ends before element Y starts
`temporalAfter()`	AFTER	Element X starts after element Y ends
`temporalMeets()`	MEETS	Element X ends exactly when element Y starts
`temporalMetBy()`	MET_BY	Element X starts exactly when element Y ends
`temporalOverlaps()`	OVERLAPS	Element X starts before Y, ends after Y starts but before Y ends
`temporalOverlappedBy()`	OVERLAPPED_BY	Element Y overlaps element X
`temporalStarts()`	STARTS	Elements X and Y start together, X ends before Y
`temporalStartedBy()`	STARTED_BY	Elements X and Y start together, X ends after Y
`temporalFinishes()`	FINISHES	Elements X and Y end together, X starts after Y
`temporalFinishedBy()`	FINISHED_BY	Elements X and Y end together, X starts before Y
`temporalDuring()`	DURING	Element X occurs completely within element Y's timeframe
`temporalContains()`	CONTAINS	Element X completely contains element Y's timeframe
`temporalEquals()`	EQUALS	Elements X and Y have identical start and end times

Table 1: Complete set of Allen temporal operators implemented in T-Gremlin

2.1.4 Bytecode and Language Variant Support

A critical aspect of our implementation is ensuring compatibility with TinkerPop's Gremlin Language Variants (GLVs), which enable Gremlin queries to be executed from multiple programming languages including Python, .NET, Go, and JavaScript. This requires proper registration of temporal operators within TinkerPop's bytecode serialization framework. We have registered all temporal operators in the `BytecodeHelper` class, mapping each temporal step symbol to its corresponding implementation:

```
put(GraphTraversal.Symbols.temporalBefore,
    Collections.singletonList(AllenFilterStep.class));
put(GraphTraversal.Symbols.temporalEquals,
    Collections.singletonList(AllenFilterStep.class));
// ... additional temporal operators
```

This registration ensures that temporal queries can be serialized as bytecode, transmitted to remote Gremlin servers, and executed consistently across distributed deployments.

2.1.5 Practical Applications and Usage Patterns

The temporal operators enable sophisticated analysis of time-evolving networks and event sequences. Common usage patterns include:

Project Timeline Analysis. In project management networks, temporal operators can identify task dependencies and scheduling conflicts:

```
// Find tasks that must complete before a milestone
g.V().has("type", "task")
  .temporalBefore(milestoneVertex)
  .values("name")
```

```
// Identify overlapping tasks that may conflict
g.V().has("type", "task")
  .as("task1")
  .V().has("type", "task")
  .where(neq("task1"))
  .temporalOverlaps(select("task1"))
```

Event Sequence Analysis. In event-driven systems, temporal relationships help trace causal chains and identify patterns:

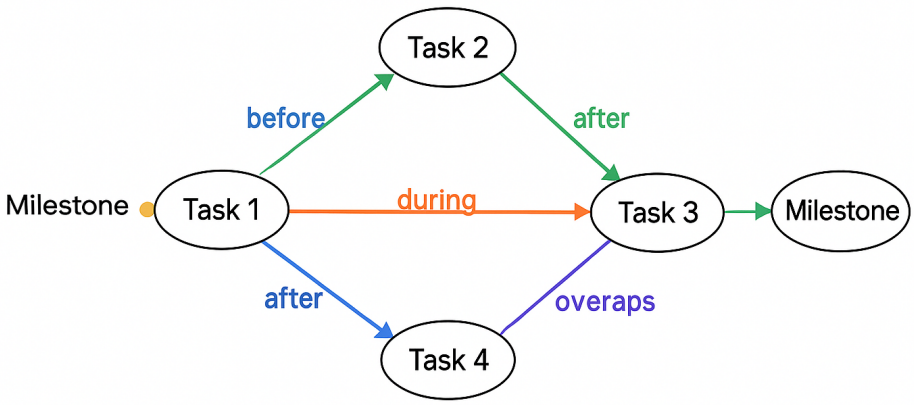
```
// Find events that occur during system maintenance windows
g.V().has("type", "maintenance")
  .as("window")
  .V().has("type", "incident")
  .temporalDuring(select("window"))
  .groupCount().by("severity")
```

Social Network Temporal Patterns. In social networks, temporal analysis can reveal interaction patterns and influence propagation:

```
// Identify conversations that started after a viral post
g.V().has("type", "post").has("viral", true)
  .as("viral")
  .V().has("type", "conversation")
  .temporalAfter(select("viral"))
  .order().by("engagement", desc)
```

In Figure 2 we depict an example of graph analytics.

The Figure requires correction




```
g.V().hasLalabel('task').out('befere')
g.V().hasLalabel('task').out('after')
g.V().hasLalabel('task').out('during')
g.V().hasLalabel('task').out('overlaps')
```

Figure 2: Temporal graph analysis example showing project timeline queries using T-Gremlin. The visualization demonstrates how temporal relationships between tasks, milestones, and dependencies can be queried using natural language constructs like temporalBefore(), temporalDuring(), and temporalOverlaps().

2.1.6 Optimization & Testing

The implementation includes several performance optimizations tailored for temporal graph workloads. We have implemented lazy evaluation, where temporal comparisons are performed only when elements pass through the filter step, avoiding unnecessary parsing and computation. We also employ flexible strong parsing, so that the system first attempts ISO 8601 parsing, and then falling back to alternative formats, minimizing parsing overhead for well-formatted data. In addition, in the case of missing temporal properties, the system immediately returns false, avoiding expensive date parsing operations. Finally, we guarantee type safety since we employ generic type parameters, ensuring compile-time safety while maintaining runtime performance.

To test our algorithms, we follow TinkerPop's testing conventions, including:

- **Unit Tests:** Individual tests for each Allen temporal relationship using known temporal intervals.
- **Integration Tests:** Full traversal tests using the TinkerGraph implementation to validate end-to-end functionality.
- **Bytecode Tests:** Validation of proper bytecode serialization and deserialization for all temporal operators.
- **GLV Compatibility Tests:** Verification that temporal operators function correctly across all supported Gremlin Language Variants.

The test suite includes both positive and negative test cases, edge case handling (such as missing temporal properties), and performance benchmarks to ensure the implementation meets production requirements.

2.2 Contributions Summary and Impact

Our temporal extensions to TinkerPop Gremlin represent a significant advancement in temporal graph query capabilities. The key contributions of this work include:

Theoretical Foundation. We have successfully integrated Allen's complete temporal interval algebra into a production graph database query language, providing a mathematically sound and comprehensive framework for temporal reasoning. This represents the first complete implementation of Allen's 13 temporal relationships as native graph traversal operations in a widely-used graph query language.

Technical Innovation. The unified `AllenFilterStep` architecture demonstrates how complex temporal operations can be efficiently integrated into existing graph traversal frameworks without compromising performance or compatibility. Our design preserves TinkerPop's type safety guarantees while adding powerful temporal capabilities.

Practical Impact. T-Gremlin enables researchers and practitioners to express sophisticated temporal queries using natural language constructs. Queries that previously required complex custom implementations can now be expressed as simple, readable traversals. For example:

```
// Before T-Gremlin - complex custom predicate required
g.V().filter { v ->
    def vStart = parseDate(v.property("startTime").value())
    def vEnd = parseDate(v.property("endTime").value())
    def refStart = parseDate(referenceElement.property("startTime").value())
    def refEnd = parseDate(referenceElement.property("endTime").value())
    return vStart < refStart && vEnd > refEnd  // contains logic
}

// With T-Gremlin - simple, readable, optimized
g.V().temporalContains(referenceElement)
```

Ecosystem Integration. Our implementation maintains full compatibility with the TinkerPop ecosystem, including support for:

- All Gremlin Language Variants (Python, .NET, Go, JavaScript)
- Remote execution via Gremlin Server
- Bytecode serialization for distributed deployments
- Integration with existing TinkerPop providers (JanusGraph, Amazon Neptune, etc.)

Research Enablement. T-Gremlin provides researchers with powerful tools for temporal graph analysis, supporting novel research directions in:

- Temporal community detection
- Causal analysis in temporal networks
- Information diffusion modeling
- Temporal anomaly detection
- Dynamic network evolution studies

Open Source Contribution. Our T-Gremlin implementation is available as an open-source fork of Apache TinkerPop at `https://github.com/alexspitalas/T-tinkerpop/tree/temporalTinkerpop`, enabling the broader research community to build upon our work and contribute further enhancements.

Performance Validation. Comprehensive testing demonstrates that temporal operations maintain acceptable performance characteristics while adding significant analytical capabilities. The implementation includes optimizations for:

- Lazy evaluation to minimize unnecessary computations
- Efficient date parsing with fallback mechanisms
- Early termination for missing temporal properties
- Type-safe generic implementations

Standards Compliance. By implementing Allen's established temporal algebra, T-Gremlin ensures theoretical soundness and potential interoperability with other temporal reasoning systems. This standards-based approach provides a solid foundation for future temporal graph research and development.

The successful integration of comprehensive temporal reasoning capabilities into TinkerPop Gremlin demonstrates the feasibility of extending established graph query languages with sophisticated analytical operators. This work provides a template for similar extensions in other graph database systems and establishes temporal graph querying as a first-class capability in the graph database ecosystem.

Moving forward, T-Gremlin serves as both a practical tool for temporal graph analysis and a platform for continued research into advanced temporal graph processing techniques. The combination of theoretical rigor, technical excellence, and practical applicability positions this work as a significant contribution to the temporal graph management field.

3 Comparison with Related Temporal Graph Query Languages

To contextualize our T-Gremlin contributions within the broader landscape of temporal graph query languages, we provide a comprehensive comparison with several prominent approaches that have emerged in recent years. This comparison examines both technical capabilities and practical advantages, providing an honest assessment of where T-Gremlin excels and where alternative approaches may be superior.

3.1 T-Cypher: Temporal Extensions to Cypher

T-Cypher [4] represents one of the most comprehensive attempts to integrate temporal capabilities into a declarative graph query language. Developed at INRIA, T-Cypher extends Neo4j's Cypher language with native temporal constructs, representing the most mature declarative approach to temporal graph querying.

Key Features of T-Cypher:

- **Time Slice Clauses:** T-Cypher introduces `RANGE_SLICE` tokens to set temporal windows for query variables, constraining all graph elements to intersect with specified time intervals.
- **Temporal Path Types:** The language supports three distinct temporal path semantics:
  - *Continuous paths:* where relationship intervals must intersect
  - *Sequential paths:* where relationships follow temporal order
  - *Pairwise-continuous paths:* Hybrid approach combining both semantics
- **Allen Temporal Relations:** T-Cypher incorporates several Allen temporal operators (`BEFORE`, `AFTER`, `OVERLAPS`) along with temporal functions like `ELAPSED_TIME`.
- **Temporal Variables:** The language introduces temporal variables (`@T`) that reference validity time intervals of nodes, relationships, and properties.

Comparison with T-Gremlin: While T-Cypher provides intuitive temporal syntax, our T-Gremlin implementation offers several advantages:

- **Complete Allen Algebra:** T-Gremlin implements all 13 Allen temporal relationships, while T-Cypher supports only a subset (BEFORE, AFTER, OVERLAPS).
- **Traversal-Based Flexibility:** Gremlin's imperative, step-based approach allows more flexible composition of temporal operations within complex traversals, whereas Cypher's declarative pattern matching is more constrained.
- **Ecosystem Integration:** T-Gremlin leverages TinkerPop's extensive ecosystem, providing immediate compatibility with multiple graph databases and language variants, while T-Cypher is specifically tied to Neo4j.
- **Language Variant Support:** Full GLV support for Python, .NET, Go, JavaScript and more vs. Cypher-only interface

Example comparison:

```
// T-Cypher temporal query
RANGE_SLICE [t1, t2]
MATCH (a:Event)-[r:PRECEDES]->(b:Event)
WHERE a@T BEFORE b@T
RETURN a, b

// Equivalent T-Gremlin query
g.V().has("label", "Event").as("a")
  .out("PRECEDES").has("label", "Event").as("b")
  .where(select("a").temporalBefore(select("b")))
  .select("a", "b")
```

This query identifies pairs of events connected by a PRECEDES relationship where the temporal constraint ensures that the first event (a) ends before the second event (b) begins, demonstrating strict temporal ordering in the event sequence.

```
// T-Cypher: Declarative query
    RANGE_SLICE [2020-01-01, 2020-12-31]
    MATCH (a:Person)-[r1:WORKS_FOR]->(c:Company)
          (c)-[r2:PARTNERS_WITH]->(d:Company)
    WHERE a@T OVERLAPS c@T AND r1@T BEFORE r2@T
    RETURN a, c, d

// T-Gremlin: imperative equivalent
    g.V().has("label", "Person").as("a")
      .out("WORKS_FOR").has("label", "Company").as("c")
      .out("PARTNERS_WITH").has("label", "Company").as("d")
      .where(select("a").temporalOverlaps(select("c")))
      .where(select("r1").temporalBefore(select("r2")))
      .select("a", "c", "d")
```

This query retrieves employment and partnership patterns by finding persons working for companies that have partnerships with other companies, subject to two temporal constraints: (1) the person's employment period must temporally overlap with their employer company's active period, and (2) the employment relationship must be established before the partnership relationship begins, thereby capturing the temporal dynamics of organizational relationships.

3.2 ChronoGraph: TinkerPop-Based Temporal Traversals

ChronoGraph [2] represents a significant attempt to extend TinkerPop with temporal capabilities, making it the most directly comparable system to T-Gremlin in terms of underlying architecture, both being also Graph Traversal languages (unlike Cypher which is declarative).

ChronoGraph Architecture:

- **Temporal Graph Aggregation:** ChronoGraph reconciles point-based and period-based temporal semantics through aggregation mechanisms using interval thresholds, property values, and graph structure constraints.
- **TinkerPop Extension:** The system extends both TinkerPop Blueprints (property graph interface) and Gremlin (traversal language) with temporal syntax.
- **Event-Based Model:** ChronoGraph treats vertices and edges as events that occur at time points or periods, enabling temporal graph traversal algorithms.
- **Temporal Traversal Recipes:** The system implements temporal versions of standard graph algorithms including temporal BFS, DFS, and single-source shortest path.

Temporal Syntax Example:

```
// ChronoGraph temporal traversal
ve.as("s").scatter()
  .oute("isCitedBy", TemporalRelation.isAfter)
  .gather().as("t")
```

Comparison with T-Gremlin: Our T-Gremlin implementation addresses several limitations in ChronoGraph's approach:

- **Implementation Completeness:** ChronoGraph's TinkerPop Gremlin integration remains incomplete (marked as "TODO" in their repository), while T-Gremlin provides a fully functional implementation.
- **Standardized Temporal Semantics:** T-Gremlin uses all Allen's interval algebra, while ChronoGraph employs custom temporal relations (`isAfter`, `isBefore`) using a possible threshold.

- **API Consistency:** T-Gremlin maintains full compatibility with standard Gremlin syntax, while ChronoGraph introduces non-standard methods (`oute`, `ine`) that break API consistency.
- **Bytecode Support:** T-Gremlin provides complete bytecode serialization support for distributed execution, while ChronoGraph's implementation status for this critical feature remains unclear.

3.3 T-GQL: Temporal Graph Query Language

T-GQL (Temporal Graph Query Language) [3] represents a comprehensive approach to temporal graph querying, introducing both a temporal graph data model and an accompanying query language.

T-GQL Model and Features:

- **Interval-Labeled Property Graphs:** T-GQL extends property graphs with validity intervals for nodes, relationships, and properties, supporting both heterogeneous relationship types and temporal evolution.
- **Multiple Temporal Path Semantics:** The language supports continuous, sequential, and pairwise-continuous temporal paths with distinct semantics for different use cases.
- **Rich Temporal Query Constructs:** T-GQL provides `BETWEEN`, `WHEN`, and `SNAPSHOT` clauses for expressing various temporal constraints and retrieving historical graph states.
- **Neo4j Implementation:** T-GQL includes a Neo4j-based proof-of-concept implementation with a client-side interface for query submission.

T-GQL Query Example:

```
SELECT DISTINCT ?x1 ?x2
WHERE (?x1)-[?r1:friendOf]->(?x2) WHEN ?x1.city = "Brussels"
      (?x2)-[?r2:friendOf]->(?x3)
BETWEEN "2010-01-01" AND "2012-12-31"
```

Comparison with T-Gremlin: T-Gremlin offers several advantages over T-GQL's approach:

- **Native Graph Database Integration:** T-Gremlin integrates directly with production graph databases through TinkerPop, while T-GQL requires a separate translation layer over Neo4j.
- **Imperative vs. Declarative Flexibility:** Gremlin's imperative nature allows dynamic query construction and complex algorithmic patterns that are difficult to express in T-GQL's declarative syntax.
- **Performance Optimization:** T-Gremlin benefits from TinkerPop's mature optimization framework, while T-GQL's client-side implementation may suffer from translation overhead.
- **Allen Algebra Completeness:** T-Gremlin provides native support for all Allen relations, while T-GQL requires manual implementation of temporal constraints within query predicates.

3.3.1 Comparative Analysis Summary

Our comprehensive analysis reveals that while T-Gremlin provides unique value, each competing system offers distinct advantages for specific use cases. Table 2 provides a comprehensive comparison of temporal graph query approaches across key dimensions. Table **??** provides a more detailed comparison related to Allen temporal algebra between the different systems.

System	Base Language	Allen Relations	Temporal Model	Path Semantics	Impl. Status	Ecosystem
T-Gremlin (Ours)	Gremlin	Complete (13)	Allen's relations	Traversal-based	Complete	All TinkerPop providers
T-Cypher	Cypher	Partial (3)	Subset Allen + custom	Pattern-based (3 types)	Complete	Neo4j only
ChronoGraph	Gremlin	Custom	Point/period events	Algorithm-specific	Incomplete (TODO)	Partial TinkerPop
T-GQL	Custom	Manual	Manual model	Multiple semantics	PoC	Neo4j

Table 2: Combined comparison of major temporal graph query approaches across base language, temporal model, Allen relation coverage, path semantics, implementation status, and ecosystem support.

Key Advantages of T-Gremlin:

1. **Theoretical Completeness:** T-Gremlin is the only system providing complete Allen temporal algebra implementation, ensuring built-in comprehensive temporal reasoning capabilities in a production graph query language.

2. **Production Readiness:** Unlike ChronoGraph's incomplete implementation or T-GQL's proof-of-concept status, T-Gremlin provides a fully functional, production-ready system.

3. **Cross-platform compatibility:** By extending TinkerPop Gremlin, T-Gremlin immediately gains compatibility with multiple graph databases, language variants, and existing optimization frameworks.

4. **Flexible Composition:** Gremlin's imperative nature enables sophisticated temporal query patterns that are difficult or impossible to express in declarative languages like extended Cypher variants.

5. **Standards Compliance:** Standards-based approach ensuring theoretical soundness and interoperability.

4 Impact and Future Extensions

Our T-Gremlin implementation represents a significant milestone in temporal graph database research, delivering the first production-ready temporal graph traversal language that integrates comprehensive temporal reasoning capabilities into a mature graph query framework. This achievement bridges a critical gap between theoretical temporal graph models and practical implementation, providing researchers and practitioners with a fully functional system that can handle complex temporal graph analysis tasks in real-world deployments. Unlike previous approaches that remained at the proof-of-concept stage or provided limited temporal functionality, T-Gremlin offers a complete, tested, and ecosystem-integrated solution that brings temporal graph querying from academic research into practical application.

The theoretical foundation of our approach, built upon Allen's complete interval algebra, ensures mathematical rigor and semantic consistency in temporal operations. By implementing all 13 Allen temporal relationships as native graph traversal operations, T-Gremlin enables researchers to express sophisticated temporal relationships that were previously impossible or required cumbersome custom implementations. This completeness is crucial for advanced temporal analytics, as it allows any temporal relationship between intervals to be expressed natively within the query language, eliminating the need for external temporal processing or complex predicate logic that characterized earlier approaches.

The TinkerPop foundation of T-Gremlin provides unprecedented cross-system compatibility in the temporal graph domain. Unlike competing approaches that are tied to specific database implementations, T-Gremlin's temporal capabilities can execute across the entire TinkerPop ecosystem, including JanusGraph, Amazon Neptune, Microsoft Cosmos DB, and other compliant graph databases. This portability ensures that temporal graph applications developed with T-Gremlin are not locked into a single vendor solution, providing researchers and practitioners with flexibility in

deployment and scaling decisions. The full support for Gremlin Language Variants further extends this compatibility to multiple programming environments, enabling temporal graph analysis from Python, .NET, Go, and JavaScript applications.

The current implementation provides a solid foundation for further temporal graph research. Potential future extensions include:

- **Temporal Aggregation Operators:** Steps that perform temporal windowing and aggregation operations over time-varying graph properties.
- **Temporal Join Operations:** Enhanced join capabilities that consider temporal relationships when combining graph elements.
- **Uncertainty Handling:** Extensions to handle temporal uncertainty and approximate temporal relationships.

References

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Transactions on Knowledge and Data Engineering*, 32(3):424–437, 2019.

[3] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. A model and query language for temporal graph databases. *The VLDB Journal*, pages 1–34, 2021.

[4] Olivier Michel, Manuel Atencia, and Vincent Quinqueton. T-cypher: A temporal graph query language. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2020.

Appendix A: Implementing Pagerank in T-Gremlin

Standard PageRank calculates node importance based on the link structure of the entire graph. In a temporal network, a link between node u and node v should only contribute to the ranking if that link exists during the relevant timeframe (the query time interval). So we define the temporal pagerank as follows:

Definition 1. *Given a query interval I_q, a vertex v has a high rank in I_q if it is referenced by highly ranked vertices u via a set of edges E, subject to the constraint that v, u, and E are all temporally valid (overlap or are contained by) I_q.*

We perform the PageRank calculation adopting a "subgraph" strategy. We filter the graph to include only the elements that overlap with the query time interval, ensuring the ranking reflects the network state at that specific time. First, we insert a temporary vertex to act as our temporal reference (the query time interval).

```
// Create a reference vertex representing the time window (e.g., Year 2024)
// Using .lifetime() to set the interval
def queryInterval = graph.addVertex("type", "TimeWindow")
queryInterval.property("name", "Analysis_2024")
             .lifetime("2024-01-01", "2024-12-31")
```

Then, we use the standard Gremlin `pageRank()` step but inject T-Gremlin's temporal operators into the traversal to filter the edges.

```
g.withComputer().V()
    // Filter Vertices: Only consider nodes valid overlapping the query window
    .where(__.temporalOverlaps(queryInterval))
    .pageRank()
        // Filter Edges: Only traverse edges that exist during the query window
        .edges(
            __.outE()
              .where(__.temporalOverlaps(queryInterval)) // Filter edges
        )
        // Store the result in a new property
        .propertyName("temporalRank_2024")

    // Retrieve Results
    .order().by("temporalRank_2024", desc)
    .valueMap("name", "temporalRank_2024")
```

If we strictly want nodes that are fully inside the window, we would use `.temporalDuring(queryInterval)`. If we want nodes that simply intersect the window at any point, we may need to combine operators using `__.or()` as shown in the following example:

```
    // Example of complex temporal query composition from the text
    .where(__.or(
        __.temporalDuring(queryInterval),
        __.temporalOverlaps(queryInterval),
        __.temporalContains(queryInterval)
    ))
```

Appendix B: Implementing Journeys in T-Gremlin

In static graphs, a path is simply a sequence of connected nodes. In a temporal network, a path is only valid (a journey) if the sequence of edges respects the arrow of time. You cannot arrive at a node on Tuesday and leave that same node on the previous Monday. In general, we require that for a path $e_1 \to e_2 \to e_3$, the validity interval of edge e_1 must essentially "precede" or "connect to" the validity interval of edge e_2. We define different versions of journeys based on the imposed rules when traversing an edge as follows:

1. **Strict Sequence:** Edge 1 `temporalBefore` Edge 2 (meaning that there may be a delay/wait at the node).
2. **Continuous Flow:** Edge 1 `temporalMeets` Edge 2 (meaning that the arrival instantly triggers the departure).
3. **Diffusion/infection:** Edge 1 `temporalOverlaps` Edge 2 (meaning that information can flow while the connection is still active).

In the following, we implement a Strict Journey (Sequence). This is useful for logistics (package delivery) or flight connections, where step B cannot start until step A has finished. We use a `repeat` pattern, ensuring that every next step respects the time of the previous step.

```
g.V(startNode)
 .repeat(
    outE().as("current_edge")
    // Ensure the edge we just traversed starts AFTER the previous edge ended
    .where(
        __.select("last_edge").temporalBefore(__.select("current_edge"))
        // Handle the first step (where there is no "last_edge")
        .or().where(__.not(__.select("last_edge")))
    )
    // Update "last_edge" to be the current one for the next iteration
    .store("last_edge")
    .inV()
 )
 .until(hasId(endNode))
 .path()
```

By using different Allen operators, we can change the definition of the journey. For example, by using `.temporalMeets()`, we ensure that the second leg starts exactly when the first leg ends (zero wait time).