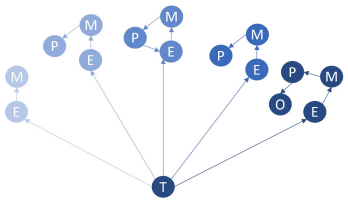
TEMPO

Management and Processing of Temporal Networks

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

D2.2 - Final Version of the MAGMA System Prototype



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

D2.2 - Final Version of the MAGMA System Prototype

Alexandros Spitalas and Kostas Tsichlas

November 30, 2025

Abstract

This report presents the final prototype of T-JanusGraph, a comprehensive extension of the JanusGraph database designed to address the challenges of managing large-scale, time-evolving graphs and efficiently implementing the HiNode model. Built upon JanusGraph's robust distributed architecture and ACID transaction support, the system introduces a novel TemporalPlacementStrategy that optimizes data distribution by assigning vertices based on temporal-weighted neighbor locality. To facilitate precise temporal analytics, T-JanusGraph integrates with Apache Solr to provide millisecond-granularity indexing, supporting efficient point-in-time and range queries, while also offering T-Gremlin extensions for temporal graph traversals. Furthermore, the system inherits JanusGraph's backend and graph-level replication mechanisms to ensure fault tolerance and is currently testing temporal adaptive replication for high-importance entities. T-JanusGraph ultimately provides a production-grade platform that lowers the barrier to entry for researchers analyzing dynamic networks and social network evolution. In this way, time will not be managed explicitly by the users, but will be a fundamental characteristic of the system providing all necessary facilities automatically. As part of our effort to address the challenges inherent in managing large-scale, time-evolving graphs, we forked and extended JanusGraph to create *T-Janusgraph*(`https://github.com/alexspitalas/T-janusgraph/tree/TemporalJanusGraph`).

1 Introduction

After conducting an exhaustive search of possible approaches described in the technical/scientific plan (D1.2) and the first progress report (D1.3) for creating a temporal graph database capable of efficiently implementing the HiNode model [3], we identified **JanusGraph** [2] as the optimal foundation. This choice was driven by its data structure compatibility with HiNode, open-source nature, and robust distributed architecture. As a result, we created the prototype of our system **T-JanusGraph**[1]. T-JanusGraph provides a novel temporal partitioning strategy, enhanced temporal indexing with Solr, and supports temporal graph traversals and analytics through T-TinkerPop extensions.

Building a temporal graph database from scratch would require implementing fundamental distributed systems components that JanusGraph already provides. JanusGraph offers:

- **Multi-Backend Storage Support:** Native integration with Apache Cassandra, HBase, and Berkeley DB provides flexibility in choosing storage backends optimized for different workloads[2].
- **Transaction Management:** JanusGraph supports ACID transactions with configurable consistency levels, handling the complex coordination required for distributed temporal updates.
- **Fault Tolerance:** Built-in replication and failure recovery mechanisms ensure system reliability without requiring custom implementation of these critical features.
- **TinkerPop Integration:** The developed T-Gremlin temporal graph traversal support described in deliverable D3.2, facilitates complex temporal queries without building custom query engines.

[1] Initially, we intended to name our system MAGMA. However, since we wanted our system to easier infiltrate the Janusgraph community, we changed the name of the system accordingly.

[2] `https://janusgraph.org`

- **Enterprise Features:** Built-in full-text search via Elasticsearch/Solr, monitoring, metrics, and security frameworks achieve production readiness.
- **Active Community and Documentation:** The active community provides continuous improvements, bug fixes, and comprehensive documentation.

In the following, we provide a description of T-JanusGraph to facilitate the maintenance and querying of temporal graphs. Particularly, in Section 2 we look at our novel, temporal-specific partitioning strategy implemented within T-JanusGraph. In Section 3, we discuss the proposed indexing implementation that facilitates temporal queries. In Section 4 we discuss the replication strategies in T-JanusGraph.

2 Temporal Partitioning Strategy Implementation

T-JanusGraph introduces a *TemporalPlacementStrategy*, which is an online partitioning approach that addresses the challenge of distributing temporal graph data by considering both structural connectivity and temporal relationships [4]. In particular, we are looking to optimize the partitioning strategy of temporal graphs along two axes:

- **Temporal Locality:** The strategy assigns vertices based on temporal-weighted neighbor locality. For each vertex, it calculates partition weights by summing the durations of edges to neighbors in each partition (using `startTime` and `endTime` properties). This ensures vertices are co-located with their most temporally significant neighbors, optimizing for both structural and temporal locality.
- **Load Balancing:** The system maintains global partition statistics to prevent imbalance. When the vertex count difference between partitions exceeds a predefined threshold (which is a parameter now set at 20%), new vertices are assigned to the least loaded partition regardless of temporal affinity. This threshold is used as a load balancing trigger.

The implementation uses concurrent data structures for thread-safe partition tracking:

```
// Global map tracking vertex-to-partition assignments
private static final Map<InternalVertex, Integer> vertexPartitionMap =
    new ConcurrentHashMap<>();

// Global counter for vertices per partition
private static final Map<Integer, Integer> partitionVertexCounts =
    new ConcurrentHashMap<>();
```

The partition selection algorithm weights each candidate partition by the cumulative temporal duration of edges to existing neighbors:

```
// Calculate edge duration (default: one year if timestamps missing)
long durationInMs = 31540000000L;
if (startTime != null && endTime != null) {
    durationInMs = (endTime.getTime() - startTime.getTime());
}

// Accumulate temporal weight for each neighbor's partition
neighborPartitionCounts.put(neighborPartition,
    neighborPartitionCounts.getOrDefault(neighborPartition, 0L) + durationInMs);
```

2.1 Java Implementation Usage

The TemporalPlacementStrategy can be configured in T-JanusGraph through a **configuration-based setup**, which configures the strategy using T-JanusGraph properties before opening the graph instance:

```
// Configure T-JanusGraph with TemporalPlacementStrategy
JanusGraphFactory.Builder config = JanusGraphFactory.build()
    .set("storage.backend", "cassandra")
    .set("storage.hostname", "localhost")
    .set("ids.placement", "temporal")

JanusGraph graph = config.open();
```

This approach enables the temporal-aware partitioning for subsequent vertex and edge operations, ensuring optimal data locality based on temporal relationships.

3 Temporal Indexing with Apache Solr

T-JanusGraph integrates with Apache Solr to provide temporal query capabilities (e.g., stab queries) through a comprehensive indexing strategy that supports both vertex and edge temporal properties. The temporal indexing system creates multiple specialized indices to optimize different types of temporal queries:

```
// Schema creation with temporal indexing
JanusGraphManagement mgmt = graph.openManagement();

// Define temporal property keys
PropertyKey startKey = mgmt.makePropertyKey("startTime").dataType(String.class).make();
PropertyKey endKey = mgmt.makePropertyKey("endTime").dataType(String.class).make();

// Edge temporal indexes
mgmt.buildIndex("EdgeIndexStartEnd", Edge.class)
    .addKey(startKey).addKey(endKey)
    .buildMixedIndex("search");

mgmt.buildIndex("EdgeIndexStart", Edge.class)
    .addKey(startKey)
    .buildMixedIndex("search");

mgmt.buildIndex("EdgeIndexEnd", Edge.class)
    .addKey(endKey)
    .buildMixedIndex("search");

// Vertex temporal indexes
mgmt.buildIndex("mixedIndexForStart", Vertex.class)
    .addKey(startKey)
    .buildMixedIndex("search");

mgmt.buildIndex("mixedIndexForEnd", Vertex.class)
    .addKey(endKey)
    .buildMixedIndex("search");

mgmt.buildIndex("mixedIndexForStartEnd", Vertex.class)
    .addKey(startKey).addKey(endKey)
    .buildMixedIndex("search");

mgmt.commit();
```

The Solr integration provides:

- **Millisecond Granularity**: Precise temporal indexing supporting fine-grained temporal queries using string-based time stamps that enable exact temporal matching and range operations.

- **Efficient Temporal Range Queries**: Multi-dimensional temporal queries leveraging combined start and end time indexes for complex temporal interval operations without requiring full graph scans.
- **Complex Temporal Predicates**: Support for temporal operators including before, after, during, and overlaps relationships through Gremlin's predicate system integrated with Solr's search capabilities.
- **Mixed Indexing**: Combination of vertex properties and temporal metadata in unified indexes enabling efficient multi-dimensional queries that combine structural and temporal constraints.
- **Automatic Real-time Synchronization**: Real-time index updates ensuring temporal indexes remain consistent with graph mutations through JanusGraph's transaction management.

3.1 Temporal Query Examples

The temporal indexing system enables various sophisticated query patterns:

Point-in-Time Queries:

```
// Find all vertices active at a specific time
GraphTraversalSource g = graph.traversal();
List<Vertex> activeVertices = g.V()
    .has("startTime", lte("2023-06-15T10:00:00"))
    .has("endTime", gte("2023-06-15T10:00:00"))
    .toList();
```

Temporal Range Queries:

```
// Find edges that existed within a specific time window
List<Edge> temporalEdges = g.E()
    .has("startTime", gte("2023-01-01T00:00:00"))
    .has("endTime", lte("2023-12-31T23:59:59"))
    .toList();
```

Complex Temporal Predicates:

```
// Find vertices that started before a time and are still active
List<Vertex> longRunningVertices = g.V()
    .has("startTime", lt("2023-01-01T00:00:00"))
    .has("endTime", gte("2023-06-15T10:00:00"))
    .toList();

// Temporal overlap detection
List<Vertex> overlappingVertices = g.V()
    .has("startTime", lte("2023-06-30T23:59:59"))
    .has("endTime", gte("2023-06-01T00:00:00"))
    .toList();
```

Temporal Graph Traversals:

```
// Find temporal neighbors within a time window
List<Vertex> temporalNeighbors = g.V(sourceVertexId)
    .outE()
    .has("startTime", lte(queryTime))
    .has("endTime", gte(queryTime))
    .inV()
    .toList();
```

3.2 Performance Optimization Features

The Solr temporal indexing provides several performance enhancements:

- **Index Partitioning**: Temporal data can be partitioned across multiple Solr cores for parallel query processing, with the *TemporalPlacementStrategy* ensuring data distribution.
- **Caching Strategies**: Intelligent caching of frequently accessed temporal ranges improves query response times by leveraging Solr's built-in caching mechanisms.
- **Incremental Updates**: Efficient handling of temporal data modifications without requiring complete reindexing, supporting real-time temporal graph evolution.
- **Query Optimization**: Combined indexes (EdgeIndexStartEnd, mixedIndexForStartEnd) enable single-index lookups for complex temporal range queries, reducing query execution time.

4 Replication Mechanisms in T-Janusgraph

A crucial aspect in the design of distributed graph systems is the ability to ensure high availability and fault tolerance, particularly as the volume and velocity of data increase. To this end, T-Janusgraph inherits and builds upon the replication mechanisms provided by JanusGraph, offering robust options for data redundancy and resilience [1].

- **Backend Storage Replication:** JanusGraph supports a range of distributed storage backends, such as Apache Cassandra and ScyllaDB. Replication at this level is managed by the backend itself. For instance, when using Cassandra, the replication factor can be configured to determine how many copies of the data are maintained across the cluster. This property is set during the creation of the keyspace (e.g., via `storage.cassandra.replication-factor` in the configuration). Adjustments to the replication factor for existing keyspaces are performed directly through backend-specific tools and not via JanusGraph's configuration interface. This approach ensures that the underlying data remains available and consistent even in the event of node failures or network partitions.
- **JanusGraph-Level Replication:** Beyond backend storage, JanusGraph provides mechanisms to replicate graph data across multiple graph instances or even data centers. This is achieved through the configuration of replication groups and strategies. Users can define replication groups to specify which instances should synchronize data, and select between asynchronous (`ASYNC`) and synchronous (`SYNC`) replication modes. These settings allow fine-grained control over the trade-off between consistency and performance, as well as the ability to tune replication rates, concurrency, and timeouts to suit the requirements of specific workloads. Monitoring facilities are also available to track the status and health of replication processes.

Key Parameters for Fine-Tuning Replication: Several parameters allow precise control over the replication process:

- `replication-rate`: Limits the number of replication operations per second, helping to avoid overwhelming the system during heavy data synchronization.
- `replication-concurrency`: Sets the number of replication operations that can occur in parallel, enabling efficient use of resources and faster data propagation.
- `replication-timeout`: Defines the maximum duration (in milliseconds) allowed for a replication operation to complete, ensuring that stalled or slow replications do not block the system.

These options provide flexibility to balance throughput, latency, and resource usage according to the specific workload and deployment environment.

Example Configuration.

```
Config config = Config.build()
    .set("storage.backend", "cassandra")
    .set("replication-group", "my-group")
    .set("replication-strategy", "ASYNC")
    .set("replication-rate", 100)
    .set("replication-concurrency", 4)
    .set("replication-timeout", 30000)
    .build();
Graph graph = new JanusGraphFactory().open(config);
```

In summary, T-Janusgraph leverages and extends JanusGraph's replication mechanisms, providing configurable options to ensure robust, scalable, and resilient management of dynamic graph data. Table 1 summarizes the replication options for T-Janusgraph. We are also looking at temporal adaptive replication, where the idea is to have a larger replication factor for entities that have large temporal importance, measured in terms of their valid interval, and smaller replication factor for the rest of the entities. We expect to test this approach in T-JanusGraph and have experimental results related to the efficiency as well as the effectiveness of this approach for replication.

Replication Type	**Mechanism in T-Janusgraph (via JanusGraph)**
Storage backend	Replication factor set in backend configuration
Graph-level	Replication groups, strategy (SYNC/ASYNC), and parameters for rate, concurrency, timeout

Table 1: Replication options in T-Janusgraph

5 T-JanusGraph Contributions and Future Directions

Our T-JanusGraph extensions provide comprehensive temporal graph capabilities through multiple integrated components. The framework introduces temporal-aware vertex placement strategies, enhanced storage backends for temporal metadata, and specialized partitioning mechanisms for temporal data distribution. Current capabilities include:

- **Temporal Data Storage:** Efficient storage and retrieval of time-stamped vertices and edges with millisecond precision temporal queries integrated with Cassandra and other backend storage systems.
- **Temporal Partitioning Strategy:** The *TemporalPlacementStrategy* enables intelligent vertex placement based on temporal-weighted neighbor locality, ensuring co-location of temporally significant graph elements across distributed partitions.
- **Temporal Indexing:** Specialized indexing mechanisms supporting efficient stabbing queries, temporal range searches, and time-point lookups with optimized data structures for temporal graph traversals.
- **Temporal Graph Query Language:** Integration with T-TinkerPop extensions providing temporal-aware traversal steps, time-range filtering operators, and chronological path analysis capabilities for complex temporal graph queries.
- **HiNode-Based Storage Architecture:** Leverages the HiNode model architecture, a storage-optimal design specifically engineered for temporal graph data, enabling efficient hierarchical organization and retrieval of temporal information while minimizing storage overhead and maximizing query performance.

T-JanusGraph lays the groundwork for a production-grade temporal graph database by introducing key capabilities - temporal partitioning, indexing for stabbing and range queries, and a

time-aware traversal API - within the familiar JanusGraph ecosystem. While still a prototype, it empowers researchers with a unified platform for exploring dynamic and temporal networks: they can now formulate and execute precise temporal queries, evaluate evolving graph metrics, and prototype time-dependent algorithms without building custom infrastructures. By lowering the barrier to entry for temporal analysis, T-JanusGraph accelerates experimentation in fields such as social network evolution, event detection, and longitudinal pattern mining.

Figure 1 depicts the current progress related to changes in the subsystems of JanusGraph. We are currently in the process of implementing the time-sensitive replication mechanism. Future enhancements include the following: a) multi-version storage and b) query planner time-sensitive optimizations. These have already been scheduled for implementation within a future (already accepted) research project (Trust Your Stars Action I: Young Researchers).

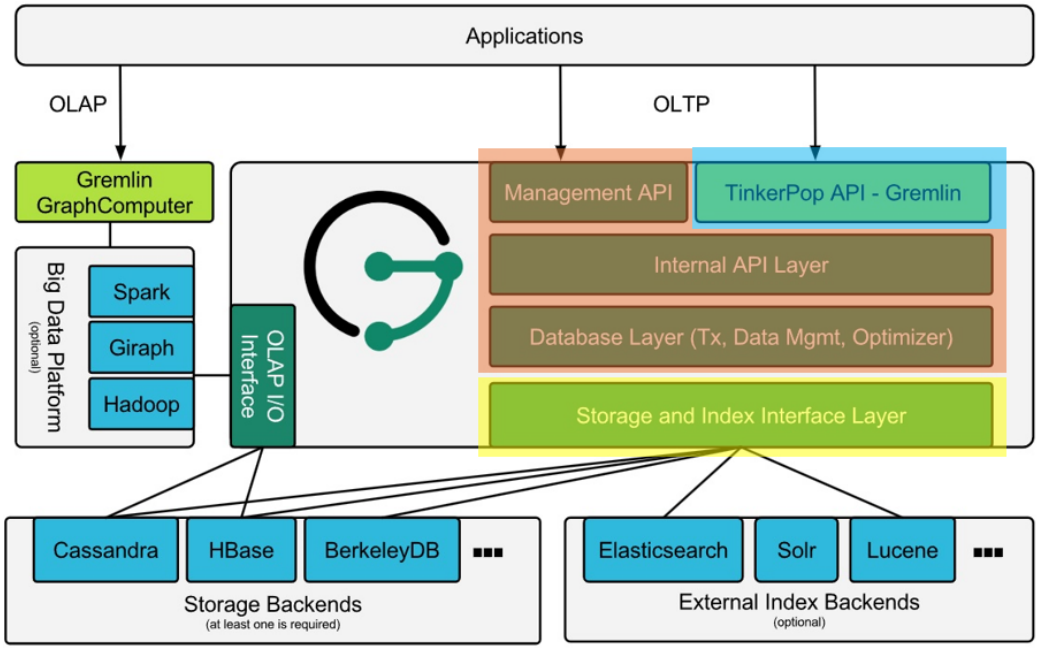


Figure 1: The architecture of JanusGraph. The highlighted orange area corresponds to our current changes for T-JanusGraph. The blue highlighted area corresponds to changes described in deliverable D3.2. Finally, the yellow highlighted area corresponds to future changes that will be combined with in-house implementations of index structures.

References

[1] The Pi Guy. Janusgraph and data replication: Replicating data across the graph. `https://the-pi-guy.com/blog/janusgraph_and_data_replication_replicating_data_across_the_graph/`, 2024. Accessed: 2025-07-09.

[2] JanusGraph Contributors. Janusgraph: an open-source, distributed graph database, 2024.

[3] Andreas Kosmatopoulos, Kostas Tsichlas, Anastasios Gounaris, Spyros Sioutas, and Evaggelia Pitoura. Hinode: an asymptotically space-optimal storage model for historical queries on graphs. *Distributed Parallel Databases*, 35(3-4):249–285, 2017.

[4] Alexandros Spitalas, Georgios Tsolas, and Kostas Tsichlas. Partition strategies for vertex-centric historical graph systems. In Jiman Hong, Sebastiano Battiato, Christian Esposito, Juw Won Park, and Adam Przybylek, editors, *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, SAC 2025, Catania International Airport, Catania, Italy, 31 March 2025 - 4 April 2025*, pages 425–431. ACM, 2025.