Validating Complex Interval-Based Temporal Queries over Streams

Philip Dasler and Sana Malik*

University of Maryland, College Park, MD 20740

E-mail: {daslerpc,maliks}@cs.umd.edu

Abstract

Streaming databases have been growing in popularity as the availability of large datasets gathered in real-time has steadily increased. Along with the ability to gather and store this data, one must also consider the various concerns involved in processing this data. More specifically, the performance of queries over such databases is highly dependent on the complexity of the query and the database itself. These considerations are common in analyses of database systems. However, one portion of query processing that seems to be often overlooked is query validation as it is generally assumed that a given query is valid.

With this in mind, this project seeks to discover whether or not the resources necessary to validate a query make it an entirely cost-prohibitive action, or, if not, whether an intelligent decision could be made on a case-by-case basis whether or not validation should occur at all. In particular cases, e.g., it may still be faster to blindly run an invalid query on a small database.

This paper discusses various methods for validating temporal queries, presents an implementation of a temporal database with range queries, and describes an experiment to determine the time costs of adding a validation step to query plans. Moreover, it can be seen that there does appear to be a point at which the size of the database is large enough to warrant query validation.

*To whom correspondence should be addressed

Introduction

The ability to query a database for temporal events has become increasingly important as the availability of streaming data has skyrocketed across multiple domains. While event stream processing has seen much work,[1–8] it has mainly focused on the relations between instantaneous events. However, the modelling of events as single points in time is not always appropriate. Take, for example, a database which tracks a real-time, multi-processor system. In this system it is important to know when processors are and are not available for use, a task which is traditionally modelled as a series of constrained intervals. Thus, a database paradigm that can handle intervals as well as instantaneous events is necessary.

Li et al. present three algorithms[9] which generalize temporal pattern matching to handle both point- and interval-based events. As presented, the performance of these algorithms is evaluated under the assumption that all queries have been pre-validated. Thus, the performance as measured by the authors does not take into account the actual cost of validation.

With this in mind, this project seeks to discover whether or not the resources necessary to validate a query make the proposed algorithms impractical, either in their entirety or under certain conditions. If it is the case that the validation is cost-prohibitive, then perhaps an intelligent decision could be made on a case-by-case basis whether or not validation should occur at all.

Given the size of the database, one may be able to predict the complexity of processing a particular query. Additionally, given the length of that query, it may also be possible to estimate the time it takes to validate it. A comparison of these two values would hopefully determine whether or not pre-validating a query is worth the cost.

It is our expectation that, for a given query, one would see performance curves similar to those in Figure 1. If this is true, then validating a query would only make sense once the database size has grown to a point at which processing a query takes longer than validating it. This threshold will occur at different database sizes depending on the complexity of the query itself.

In this paper, we discuss various methods for validating temporal queries, present our implementation of a temporal database with range queries, and describe an experiment to determine the

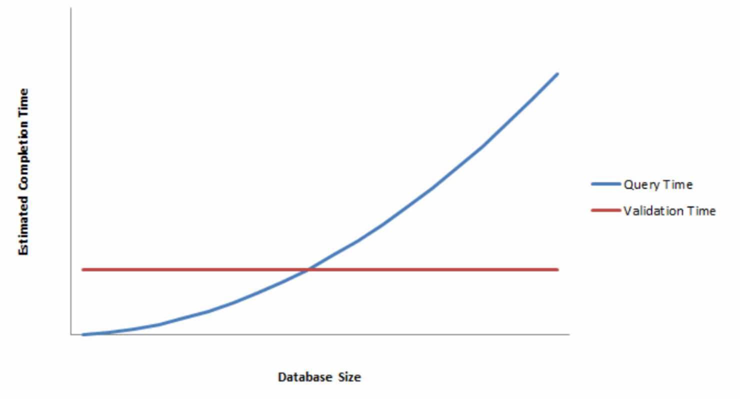


Figure 1: An expected performance comparison of validating a single query and querying databases of various sizes.

time costs of adding a validation step to query plans.

Temporal Queries over Streams

The work by Li et al. introduces a new, interval-based query operator largely based on the temporal relation classifications described by Allen.[10] The following classifications enumerate the various possible relations between intervals, with ts and te representing the start and end times of an event, respectively:

Table 1: Allen's 13 temporal relations. e is an event whose start time is $e.ts$ and end time is $e.te$.

Temporal Relation	**Temporal Algebra**
e before e'	$(e.te < e'.ts)$
e after e'	$(e.ts > e'.te)$
e during e'	$(e.ts > e'.ts)^{(}e.te < e'.te)$
e contain e'	$(e.ts < e'.ts)^{(}e.te > e'.te)$
e meets e'	$(e.te = e'.ts)$
e met by e'	$(e.ts = e'.te)$
e overlap e'	$(e.ts < e'.ts)^{(}e.te > e'.ts)^{(}e.te < e'.te)$
e overlapped by e'	$(e.ts > e'.ts)^{(}e.ts < e'.te)^{(}e.te > e'.te)$
e start e'	$(e.ts = e'.ts)^{(}e.te < e'.te)$
e started by e'	$(e.ts = e'.ts)^{(}e.te > e'.te)$
e finish e'	$(e.ts > e'.ts)^{(}e.te = e'.te)$
e finished by e'	$(e.ts < e'.ts)^{(}e.te = e'.te)$
e equal e'	$(e.ts = e'.ts)^{(}e.te = e'.te)$

Given this set of relations, Li et al define ISEQ, a query processor designed to look for sets of relations between intervals. This processor, however, makes the assumption that any query provided to it will be valid (that is to say, logically consistent). For example, a query searching for two intervals which occur before each other and after each other should be rejected by a validator before being sent to the query processor.

While validation of the query is not, strictly speaking, a complex endeavor in terms of algorithm design, it can still be computationally costly depending on the query itself. This parallels a similar concern in query optimization. If a query optimization would take longer than actually running the query on the database, then doing so provides no real benefit. This is shown by Ilyas et al,[11] who explore some methods for meta-optimization (i.e. selecting an appropriate level of optimization) in order to balance the cost of optimization with the query cost. Of course, a validation process is not really capable of adjusting its resolution in the same way a meta-optimizer might. Partially validating a query would likely provide few gains as the true goal is to throw out bad queries without having to access the database at all. This can only be done if a query is entirely validated and deemed either consistent, at which point it is processed by ISEQ, or deemed inconsistent and rejected.

Validation of Temporal Logic

The actual cost of validation depends on two factors: the query length and the validation method. Generally speaking, the validation of a temporal based query is a difficult problem. When formally represented, a query of this type leads to a conjunction of relational clauses. These clauses, each representing the relationship between to points in time, must all be concurrently satisfiable in order for the possibility of the query returning a set of satisfying records. This is simply another case of the Boolean Satisfiability Problem (SAT), a problem which has been proven to be NP-hard .[12]

Allen's Temporal Relationships[13] is an implementation of Allen's 13 relations shown in 1. The library includes path consistency validation as proposed by Allen. Due to the limited number of existing validators for Allen's Temporal Logic (ATL), however, it may be more beneficial to convert queries into a more widely used formalism. Though converting the queries is not free, the savings of using a highly optimized, existing validator may outweigh these costs. Rosu and Bensalem[14] describe a linear translation from ATL to Linear Temporal Logic (LTL), allowing LTL validation techniques to be used on relations expressed in ATL. For example, by converting the queries to LTL, it may be possible for AI planning techniques such as TLPlan[15] to be used. This method would essentially treat the temporal relationships as a partially ordered set of constraints. Then, rather than using an existing validator, ATL can be simplified to a directed graph representation, where an edge between two time points (a, b) represents a dependency, namely that a must precede b. Inconsistencies can then be detected by looking for cycles in the graph with a topological sort[16] and, if found, the query is considered invalid. The benefit of this approach is that its complexity is $O(V+E)$ where V is the number of vertices in the graph and E is the number of edges. This is at worst $O(N!)$, but is considerably less in the average case.

Unfortunately, this method will only work under a restricted set of interval relationships. It does not, for example, cover the relationships in which two event times must be equal. At best, we can approximate this by merging the vertices that represent the two values in the graph, taking the conjunction of their edges to be the edges of the new vertex. In this way we assure that each value is constrained by the same temporal dependencies, but this still does not ensure that the values are actual equal.

Additionally, this approach does not generalize well as it can not handle disjunctive operators (e.g., less than or equal to). When adding a disjunctive operator to an existing dependency graph, one can simply clone the graph and then add one of constituent operators to each graph. Thus, one graph will represent the case that the values are equal and the other graph represents when they are not. While this works, it unfortunately destroys the gains we would get from using this dependency graph approach. By creating clones for each disjunctive operator, a search tree has been generated in which each node is a graph representation of the query and each disjunctive operator creates a new branch in this tree. This leads to an overall complexity of $O(V+E)^d$, where d is the number of disjunctive operators in the query. This is much worse than we could do by simply approaching the problem as a SAT problem.

Armando et al.[17] describe SAT-based validation techniques specifically for temporal logic expressed in disjunctive normal form (DNF). As any logical formula can be converted to DNF, we can simply convert temporal queries and then run them through a SAT solver. If the DNF formula is satisfiable, then the original query is valid. This method has the advantage of allowing us to leverage the extensive work on the SAT problem without needlessly constraining our queries (as the solutions above do).

Implementation

Generating Interval Formulas

In order to prevent any unintentional bias that may be introduced by hand-crafting queries, we have instead created an automated query generator. This generator can produce any number of queries, of any size, and can ensure the satisfiability of the queries it generates. Using this, we have generated a set of queries ranging across the various parameters necessary to test our full query procedure and to gather the needed data to either confirm or contradict our hypothesis.

This generator begins by instantiating the parameterized number of intervals. For each interval a starting point and an end point is created. The system then randomly selects from one of the thirteen different interval relationships as defined by Allen. Finally, it selects two of the instantiated intervals and, using the mapping from interval relationships to endpoint relationships in Table 1, it adds a new clause to the query. It will continue to do this until the requisite number of clauses have been created. So, for example, a query might be generated looking for an interval that is "finished by" another. This query would appear thusly: `x[0] < x[2], x[1] == x[3]`. Here, each interval is represented by its start and end points as the elements `x[`$2i$`]` and `x[`$2i+1$`]`, respectively (where i is the interval number). For simplicity, all intervals are represented in a single array named `x`.

Except for the fact that the clauses are generated from a particular domain (Allen's relationships), query construction has occurred under no real restrictions thus far. While the intervals were created to fill the query they had no actual values. At this point, the query generator determines whether or not the query is satisfiable. In this way, we can control which type of query is being generated at any given time and can be selective in our testing. While it may be less efficient to generate a query, test it for satisfiability, and repeat until a query that is appropriately satisfiable is generated, the complexity of implementing a system that can deterministically generate an appropriate query is so high that it is practically infeasible. Granted, a naive approach to this problem could deterministically generate queries of a specific satisfiability, but the concern is the bias that this may introduce. For example, a system could easily make any query unsatisfiable by adding the contradiction to the first clause, but this would mean all unsatisfiable queries are unsatisfiable in the same way, which may affect the overall time necessary to determine satisfiability. This sort of bias needs to be avoided.

Determination of a query's satisfiability is done in the same way as it is done for actual runtime testing. First, the domain across which the intervals exist is specified. Given that our interest is in timestamps, the start and end points of each interval is defined to be in the real numbers. Thus, $\forall i, i_{start}, i_{end} \in \mathbb{R}$, where i is an interval. Once this is done, a series of constraints is added to the query to ensure that the intervals are properly represented. More specifically, it is also necessary that the start points of all intervals occur before the end points. If we generate a query and determine that it is satisfiable without enforcing this, we may find that with real data this query isn't actually satisfiable at all. Thus, for every interval i, the clause $\texttt{x[}2i\texttt{]} < \texttt{x[}2i+1\texttt{]}$ is added. In this way, we can be sure that when the generator produces what it considers to be a satisfiable query that it is in fact doing so.

Converting Formulas to SQL Queries

To use the formula as a SQL query, the table needed to be joined with itself depending on the number of intervals in the formula. However, with a large amount of joins, it becomes very costly to generate and store all results in memory. Thus, for experimentation purposes, we only return the *count* of the results fulfilling the formula (e.g., `SELECT count(*) ...`). In the formula representation of interval logic, even-numbered indices represent interval start times and odd-numbered indices are finish times. Then, all variables of the form x[$2n$] are translated as `x`n`.start` and variables of the form x[$2n+1$] are translated as `x`n`.finish`. These transformations are set as the `WHERE` clause in the query.

For example, the input `x[0] < x[2], x[1] == x[3]` would be converted to the query

```
SELECT count(*) FROM medium x1, medium x2 WHERE x1.start < x2.start
AND x1.finish = x2.finish
```

Experiment

We conducted an experiment to study the performance effects of including a validation step in the query plan.

Experimental Design

The experiment employed a 4x4x2 mixed factorial design. The three factors (and their treatments) were query length by number of intervals (2; 3; 4; and 6), database size by number of documents (1,000; 5,000; 25,000; and 50,000), and query satisfiability (satisfiable, unsatisfiable). The satisfiable queries were not run on the 50,000 item table or with queries of length 6 due of the exponential growth in time to run these queries.

The dataset used in the experiments were sampled from a total of 638,182 Twitter data items (tweets) collected between May 2, 2012 and June 10, 2012. Each entry contains the text of the tweet (“text”), the date-timestamp that the tweet was sent (“start”), and a “finish” date-timestamp which was fabricated by adding 1 second to the start time for every character in the tweet text. The entire dataset of 638,182 tweets was randomly sampled with a uniform distribution to create smaller databases of size 1,000, 10,000, 25,000 and 50,000 tweets to create the small and medium databases, respectively. The large database consists of the entire Twitter dataset.

Experimental Setup

This project combines several pieces of off-the-shelf software in order to build a complete query system. Below is a list of this software and a brief justification of their selection:

Server: All experiments were performed on a KVM virtual machine running Gentoo Linux with one 3GHz CPU and 8GiB of RAM.

Database System: The database server was Percona Server,[18] a fork of MySQL,[19] a free, open source, relational database management system. The tables were loaded as `MEMORY` tables because they were small enough to fit into memory and this decreased speed to query. For indexes, B-Trees were created and used in place of the default Hash indices because they are more efficient for range and inequality queries.[20]

SAT Solver: Boolean satisfiability will be determined by Z3,[21] a boolean satisfaction and optimization problem solver developed and maintained by Microsoft Research. This state of the art library is flexible, full featured, and robust and will allow us to concentrate on our experimentation rather than on implementing the necessary tools. Additionally, it comes with an easy to use Python API.

Project Management: For project management and software version control, we use a private repository on GitHub,[22] a web-based hosting service which uses the Git versioning system. Github also offers a bug tracking system and a corresponding wiki which will allow for smoother collaboration.

Language: The temporal logic and validator was implemented in Python, which was chosen for its familiarity to the developers and because many packages exist for boolean logic validation.

Results

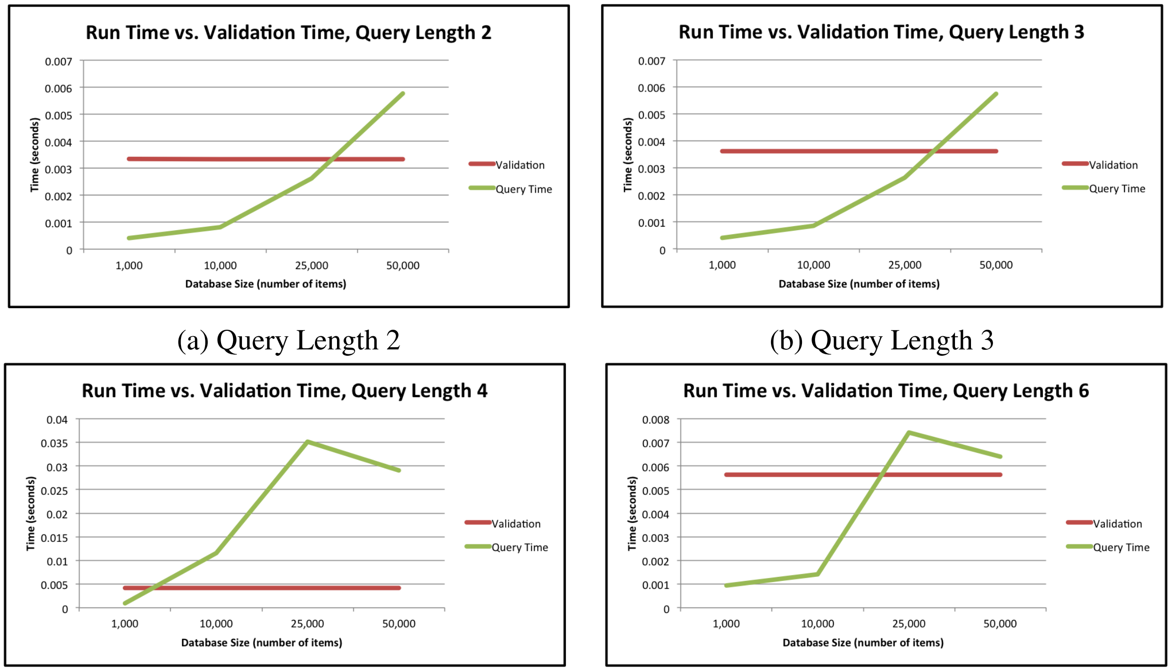
Unsatisfiable Queries

Timing results for the unsatisfiable queries are shown in Table 2.

Table 2: Timing results (in seconds) of validating and running unsatisfiable queries.

Query Length (intervals)	Validation Time (seconds)	Query Time (seconds)			
		1,000	10,000	25,000	50,000
2	0.003340	0.000403	0.000812	0.002631	0.005768
3	0.003615	0.000406	0.000851	0.002636	0.005755
4	0.004189	0.000943	0.011535	0.035187	0.029022
6	0.005631	0.000942	0.001406	0.007416	0.006392

Overall, both validating and running unsatisfiable queries were extremely quick, with times under 0.01 seconds in most cases. Graphs of the results for each query length are shown in Figure 2.



(a) Query Length 2

(b) Query Length 3

(c) Query Length 4

(d) Query Length 6

Figure 2: Graphs of validation and query time results (in seconds) for each query length versus database size.

Satisfiable Queries

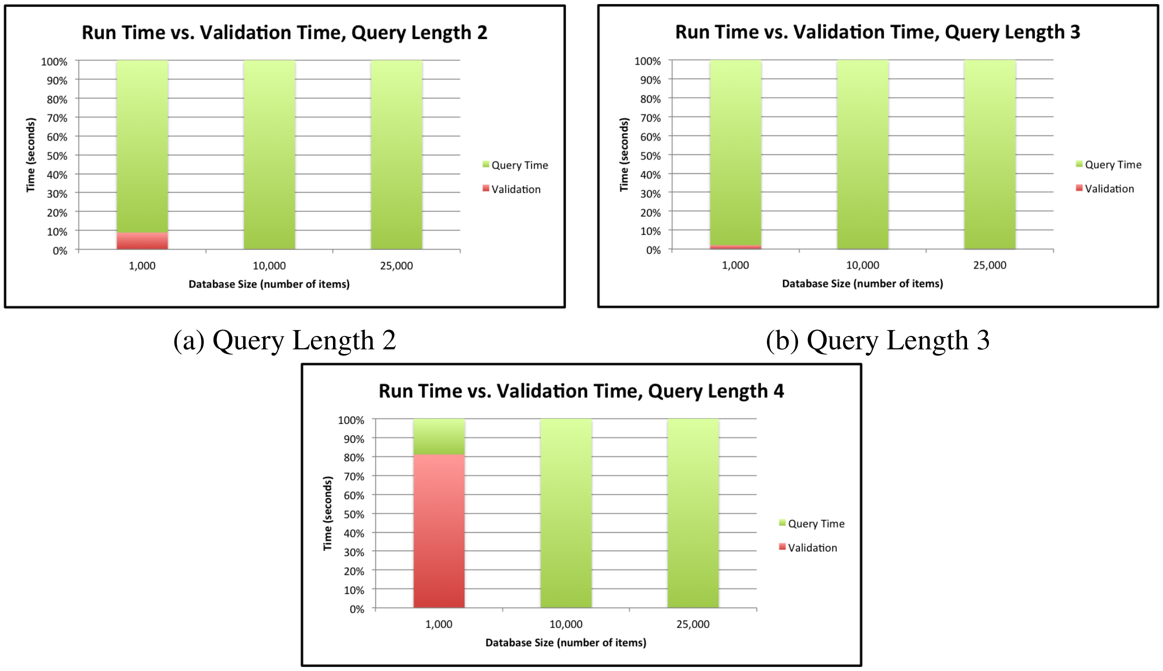
Timing results for the unsatisfiable queries are shown in Table 3.

Table 3: Timing results (in seconds) of validating and running satisfiable queries.

Query Length (intervals)	Validation Time (seconds)	Query Time (seconds)		
		1,000	10,000	25,000
2	0.002887	0.029402	2.930172	16.173374
3	0.00384	0.182679	767.943567	5777.629936
4	0.004405	0.001026	71.94892	641.795355

Validation and Query Times versus Database Size per Query Length

Overall, both validating and running unsatisfiable queries were extremely quick, with times under 0.01 seconds in most cases. Graphs of the results for each query length are shown in Figure 3.



(a) Query Length 2

(b) Query Length 3

(c) Query Length 4

Figure 3: Graphs of validation and query time results (in seconds) for each query length versus database size.

Conclusions and Future Work

Our original prediction is supported by the results of the unsatisfiable query running times. In all cases, there was a point where validating an unsatisfiable query was more efficient than running it. For query lengths 2 and 3, this point was between 25,000 and 50,000 items, for length 4 it was between 1,000 and 10,000 items, and for length 6, querying became more inefficient with between 10,000 and 25,000 items.

It appears that longer queries (length 4 and 6) combined with a larger database (50,000 items) runs more efficiently than the same query on smaller databases (25,000 items or less).

This additional is justified in the satisfiable query step as well, because all validations times are a very small percentage of the query run time, meaning negligible performance effects of running the validator, as per Amdahl's Law.[23] The exception is query length 4 on a small database, which takes up to 80% of the run time, but the actual values of the runtimes are so small (0.0028s) that this is still a very small impact on runtime.

Due to time and system constraints, there are many limitations to our experiment. Firstly, we only ran a single query of each class and length combination. In future evaluations, a more diverse set of queries may provide more accurate real-world behavior. Additionally, each query was only timed for one run, whereas it would be more accurate to average multiple iterations of the same query. Lastly, due to limited system resources, our database sizes and query lengths were quite small and short. A more realistic and beneficial evaluation would need to be done on larger database sizes with queries involving more intervals.

References

(1) Agrawal, J.; Diao, Y.; Gyllstrom, D.; Immerman, N. Efficient pattern matching over event streams. 2008; `http://doi.acm.org/10.1145/1376616.1376634`.

(2) Akdere, M.; Çetintemel, U.; Tatbul, N. *Proc. VLDB Endow.* **2008**, *1*, 66–77.

(3) Ding, L.; Chen, S.; Rundensteiner, E. A.; Tatemura, J.; Hsiung, W.-P.; Candan, K. S. Run-

time Semantic Query Optimization for Event Stream Processing. 2008; http://dx.doi.org/10.1109/ICDE.2008.4497476.

(4) Wu, E.; Diao, Y.; Rizvi, S. High-performance complex event processing over streams. 2006; http://doi.acm.org/10.1145/1142473.1142520.

(5) Brenna, L.; Demers, A.; Gehrke, J.; Hong, M.; Ossher, J.; Panda, B.; Riedewald, M.; Thatte, M.; White, W. Cayuga: a high-performance event processing engine. 2007; http://doi.acm.org/10.1145/1247480.1247620.

(6) Li, M.; Mani, M.; Rundensteiner, E. A.; Wang, D.; Lin, T. Interval event stream processing. 2009; http://doi.acm.org/10.1145/1619258.1619302.

(7) Li, M.; Mani, M.; Rundensteiner, E. A.; Lin, T. Constraint-aware complex event pattern detection over streams. 2010; http://dx.doi.org/10.1007/978-3-642-12098-5_16.

(8) Liu, M.; Li, M.; Golovnya, D.; Rundensteiner, E.; Claypool, K. Sequence Pattern Query Processing over Out-of-Order Event Streams. 2009.

(9) Li, M.; Mani, M.; Rundensteiner, E. A.; Lin, T. Complex event pattern detection over streams with interval-based temporal semantics. 2011; http://doi.acm.org/10.1145/2002259.2002297.

(10) Allen, J. F. *Commun. ACM* **1983**, *26*, 832–843.

(11) Ilyas, I. F.; Rao, J.; Lohman, G.; Gao, D.; Lin, E. Estimating compilation time of a query optimizer. 2003; http://doi.acm.org/10.1145/872757.872803.

(12) Cook, S. A. The complexity of theorem-proving procedures. 1971; http://doi.acm.org/10.1145/800157.805047.

(13) Franke, J. Allen Interval Relationships. 2011; https://code.google.com/p/allenintervalrelationships/.

(14) Roşu, G.; Bensalem, S. Allen linear (interval) temporal logic &#8211; translation to LTL and monitor synthesis. 2006; `http://dx.doi.org/10.1007/11817963_25`.

(15) Bacchus, F.; Kabanza, F. *Annals of Mathematics and Artificial Intelligence* **1998**, *22*, 5–27.

(16) Pearce, D. J.; Kelly, P. H. J.; Hankin, C. *Software Quality Control* **2004**, *12*, 311–337.

(17) Armando, A.; Castellini, C.; Giunchiglia, E. SAT-Based Procedures for Temporal Reasoning. 2000; `http://dl.acm.org/citation.cfm?id=647868.737113`.

(18) Percona, Percona Server. `http://www.percona.com/software/percona-server`.

(19) Oracle, MySQL, the world's more popular open-source database. `http://www.mysql.com`.

(20) Oracle, Comparison of B-Tree and Hash Indexes. `http://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html`.

(21) De Moura, L.; Bjørner, N. Z3: an efficient SMT solver. 2008; `http://dl.acm.org.proxy-um.researchport.umd.edu/citation.cfm?id=1792734.1792766`.

(22) Inc., G. GitHub. `http://github.org`.

(23) Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. 1967.