A Hybrid Query Language for Digital Twins

Philipp Zech[1][0000-0002-4952-4337], Manuel Burger[1], Linus Wald[1], Philipp Pobitzer[1], Sascha Hammes[2][0000-0001-5821-5053], and Judith Michael[3][0000-0002-4999-2544]

[1] University of Innsbruck, Department of Computer Science, Austria
[2] University of Innsbruck, Unit of Energy Efficient Building, Austria
`firstname.lastname@uibk.ac.at`
[3] University of Regensburg, Programming and Software Engineering, Germany
`judith.michael@ur.de`

Abstract. The full potential of Digital Twins (DTs) is hindered by the challenge of integrating complex engineering models with high-frequency runtime data from disparate sources. Existing approaches lack a unified mechanism to query across the boundary of static models and dynamic data, leading to fragmented and inconsistent DT systems. We introduce a hybrid query language that unifies model and data retrieval, translating queries into SPARQL for model access and SQL for data access within a single declarative statement. Our evaluation demonstrates that this approach overcomes the models-meet-data challenge by enabling scalable, near real-time queries, thereby paving the way for more robust and integrated DT applications.

Keywords: Digital Twins · Models-meet-data · Domain-specific Language · Query Language · Data Integration

1 Introduction

Existing Digital Twin (DT) solutions generally fail to adequately address the *models-meet-data* challenge [17], which involves synchronizing models and data across heterogeneous sources, managing their co-evolution, and providing unified access. Current approaches largely capitalize on manual integration efforts [21] which yields error-prone solutions that lack interoperability and mechanisms for efficient synchronization, leading to fragmented and disharmonized DT systems. The disconnect between models and operational data remains one of the primary obstacles to full-scale DT implementation [8,15]. This challenge is further complicated by the inherent heterogeneity of DT ecosystems, where cyber-physical systems operate across multiple abstraction levels and temporal scales [4,24]. Commensurate with this, the central question of our work arises, viz., *how can we develop a unified mechanism to enable seamless integration, management, and provisioning of heterogeneous models and data for DT engineering while ensuring scalability and interoperability?*

To this end, we propose a hybrid query language atop a repository for DTs [22,23] for integrated model and data retrieval, to resolve the models-meet-data challenge. Our approach enables efficient aggregation of models and runtime data with unified access, fostering interoperability and operational efficiency in DT systems. Our solution builds upon established information integration principles from cooperative information systems, including schema integration, distributed query processing, semantic interoperability, and data fusion [7,19], to address the models-meet-data challenge in DT engineering.

Organization Section 2 outlines the context of our work, followed by Section 3 where we outline our challenges and contributions. Section 4 discusses our contributions in detail. We evaluate our proposal in Section 5 and discuss our results in Section 6. We conclude in Section 7.

2 Background and Related Work

The effective implementation of DTs is hindered by a fundamental challenge: the seamless integration of heterogeneous engineering models with dynamic runtime data, known as the *models-meet-data* challenge [17]. This problem stems from the historical separation between model-driven engineering (MDE) and data management technologies [22,23,9], which has led to fragmented tools and manual integration efforts that lack scalability [21]. Consequently, models are managed in specialized repositories with distinct query languages while runtime data resides in separate databases. This fragmentation creates significant barriers to DT operations by impeding the co-evolution of models and data [10], preventing unified data access, and complicating the execution of hybrid queries essential for consistency checking and comprehensive analysis.

Recent work in DT data management consistently highlights the unresolved challenge of integrated model-data access [14]. While semantic approaches [13], multi-scale modeling [24], and data-level integration using GraphQL [5] have been explored, they lack a unified mechanism for querying across both engineering models and runtime data. Our approach builds on foundational data integration principles [7,19] to provide a concrete solution to the *models-meet-data* challenge articulated by van den Brand et al. [17]. To the best of our knowledge, no hybrid query language targeting both models and data in DTs currently exists; existing solutions rely on manual, non-scalable integration techniques [21,2], reinforcing the novelty of our unified query interface.

Our work extends the DAIRY repository for DT engineering [22,23] in the construction domain by introducing a unified *models-meet-data* representation via knowledge graphs, and a Domain-Specific Language (DSL) with a query generator and executor to enable hybrid querying across versioned models and runtime data.

3 Challenges and Contributions

We aim to fill the research gap of seamlessly integrating models and data for DT engineering with our proposed hybrid query language. It builds upon established information integration principles while addressing the specific challenges of DT environments, viz.,

1. the integration of heterogeneous data sources,
2. the dynamic evolution of models and data, and
3. a lack of unified access mechanisms to models-and-data.

In resolving these challenges, we deliver the following contributions

1. advancing the DAIRY repository to support model-data integration,
2. a hybrid query language for integrated model and data retrieval, and
3. a prototype addressing the models-meet-data challenge.

thereby investigating the following research questions:
RQ1: What mechanisms need to be implemented within the DAIRY repository to ensure seamless synchronization between evolving models and dynamic data sources in DT systems?
RQ2: How can a hybrid query language be designed to effectively integrate and retrieve heterogeneous model and runtime data in DT environments?

Our work is grounded in Design Science Research (DSR) [20] where the central contribution is a hybrid query language embedded in a repository for DTs. Our artifact evolved through multiple iterative cycles, each responding to insights from the problem space, system feedback, and user trials [22,23], and is implemented as a solution to the following design science problem [20]:

Improve *DT engineering* (context)
by designing *a hybrid query language for models and data* (artifact)
that satisfies *the models-meet-challenge* (requirement)
to deliver *a unified representation on a physical twin.* (goal)

4 A Hybrid Query Language For Digital Twin Engineering

Our solution capitalizes on a global-as-a-view (cf. unified schema, local views) [16] approach where a global data model for building runtime data (cf. Section 4.1) is defined in terms of local data sources (e.g., devices in a building). This data model is linked to a building model which establishes a knowledge graph and lays the foundation for our hybrid query language (cf. Section 4.2).

4.1 Data Modeling

We link a static, ifcOWL-based building model (persisted in DAIRY) with a manually defined dynamic data model following the star schema [3]. This integration is achieved by generating a semantic data model using the SOSA/SSN ontology from the dynamic data model, which creates a unified knowledge graph representing the building's structure and its corresponding data schemas. Runtime data is persisted in a dedicated relational database, to avoid the computational overhead and scalability issues associated with frequent, high-volume data ingestion in a knowledge graph. This design enables a *lazy-linking* approach, inspired by lazy evaluation [6], where combined model-data queries leverage the knowledge graph and redirect data retrieval to the appropriate SQL tables. This design ensures that models and data can be queried independently without performance degradation while allowing for their efficient, on-demand combination.

4.2 A Hybrid Query Language for Models and Data - BSQL

Our Basic Sensor Query Language (BSQL) is designed to retrieve both model and runtime data by providing a simple and intuitive way to specify the information to retrieve. BSQL queries are structured into two main components, viz., the `SENSOR` part and the `DATA` part. Consequently, queries are executed in a two-step process:

(1) The `SENSOR` part is translated into SPARQL and executed against the model store. It filters specified sensors and their associated static information from the models (contained in `IfcPropertySets` at every sensor).
(2) The `DATA` part is translated into SQL and is executed against the data warehouse to retrieve runtime data for each sensor retrieved in (1).

BSQL's novelty lies in unifying querying across structured model repositories and runtime data stores using a lightweight, composable syntax with lazy evaluation semantics [6]. BSQL is implemented using Xtext [1] for developing the query language and Xtend [1] for query generation.

```
<Statement> ::= 'SENSOR_SELECT' <PropertySelectType>
                ['WHERE' '{' <PropertyConditions> '}' ]
                ['ORDER_BY' <PSID_PID> [<Order>] {',' <PSID_PID> [<Order>]}]
                ['LIMIT' <INT>]
<PropertySelectType> ::= 'BASIC' | '(' <Clause> {',' <Clause>} ')'
<Clause> ::= 'PROPERTY_SET' <PSID> 'PROPERTIES' '(' <PID> {',' <PID>} ')'
<PropertyConditions> ::= ['NOT'] <PropCond> {('AND' | 'OR') ['NOT'] <PropCond>}
<PropCond> ::= <PSID_PID> <PropertyComparison> | '(' <PropertyConditions> ')'
<PropertyComparison> ::= (* comparison against a value that the property holds *)
<PSID_PID> ::= <PSID> '.' <PID> | 'Sensor.Id'
<PSID> ::= <String>
<PID> ::= <String>
<Order> ::= 'ASC' | 'DESC'
```

Listing 1: Simplified syntax of the `SENSOR` part of our DSL (in EBNF) for querying models-and-data. `<String>` denotes any valid UTF-8 strings, `<INT>` denotes an integer, `<PSID>` a property set ID and `<PID>` a property ID. `<PropertyComparison>` allow for detailed comparison and filtering of retrieved data. Best viewed on a computer screen.

Sensor Queries. Listing 1 shows the grammar of the `SENSOR` part of queries formulated using our hybrid models-meet-data query language. The `SENSOR` part of our queries is initiated with the keyword SENSOR_SELECT (Listing 1, line 1), followed by optional clauses that specify property sets, each associated with one or more properties of a sensor (l.5-6). The query can be refined using a WHERE clause (l.2), which includes conditions that filter sensors based on whether certain properties fall within specified sets of values, which then can also be logically connected using AND, OR, and NOT (l.7-9). Additionally, the results can be ordered using the ORDER_BY clause (l.3), which sorts the data in ascending (ASC) or descending (DESC) order (l.13). The number of returned results can be controlled using the LIMIT clause (l.4) followed by an integer. This enables the construction of flexible and powerful queries to retrieve sensors from building models based on various criteria. <PSID> (l.11) denotes a property set ID and <PID> (l.12) a property ID, respectively.

```
1  <Statement> ::= 'DATA_SELECT' <DataSelectType>
2                  ['WHERE' '{' <DataCondition> '}']
3                  ['GROUP' 'receiveTime' <GroupByTime>]
4                  ['ORDER_BY' 'receiveTime' ['ASC' | 'DESC']]
5                  ['LIMIT' <INT>]
6  <DataSelectType> ::= '*' | <AggregationType> {',' <AggregationType>}
7  <AggregationType> ::= 'MAX' | 'MIN' | 'AVG' | 'COUNT' | 'SUM'
8  <DataCondition> ::= [<AllSensorsConditions>] ['[' <SensorSpecificConditions> ']']
9  <AllSensorsConditions> ::= ['NOT'] <AllCond> {('AND' | 'OR') ['NOT'] <AllCond>}
10 <AllCond> ::= <ReceiveTimeCond> | '(' <AllSensorsConditions> ')'
11 <ReceiveTimeCond> ::= 'receiveTime' <ReceiveTimeComparison>
12 <ReceiveTimeComparison> ::= (* comparison against the receive_time path *)
13 <SensorSpecificConditions> ::= <SensorCondition> {',' <SensorCondition>}
14 <SensorCondition> ::= <PayloadCondition> 'ON_SENSOR_WITH_ID' <String>
15 <PayloadCondition> ::= ['NOT'] <PathCond> {('AND' | 'OR') ['NOT'] <PathCond>}
16 <PathCond> ::= <Path> <PathComparison>
17 <PathComparison> ::= (* comparison against a path of modeled sensor *)
18 <GroupByTime> ::= 'PER_YEAR' | 'PER_MONTH' | 'PER_DAY' | 'PER_HOUR' | 'PER_MINUTE'
19 <Path> ::= ':' <String> {':' <String>}
```

Listing 2: Simplified syntax of the DATA part of our DSL (in EBNF) for querying models-and-data. <String> denotes any valid UTF-8 string and <INT> an integer. <ReceiveTimeComparison> and <PathComparison> allow for detailed comparison and filtering of retrieved data. Best viewed on a computer screen.

Data Queries. Data queries complement sensor queries by enabling the retrieval of runtime data to combine static model information with dynamic runtime data. Listing 2 outlines the grammar of the DATA part of our hybrid query language for selecting and filtering sensor data. The DATA_SELECT statement (Listing 2, line 1) is the core construct, allowing users to select data using either predefined types or custom selections (l.6-7). The query can include a WHERE clause (l.2) to filter data based on conditions, which can be logically connected using AND, OR, and NOT operators (l.8-17). These conditions can pertain to the receiveTime or specific measurements associated with sensors. The GROUP clause (l.3) allows data to be aggregated by time intervals such as year, month, or day (l.18), while the `ORDER_BY` clause (l.4) specifies the sorting order of the results, e.g., based on the `receiveTime`, or aggregations. Additionally, the `LIMIT` clause (l.5) restricts the number of results returned.

Our hybrid query language for models-meet-data offers a powerful framework for precise sensor and data selection and analysis. It allows users to define complex queries with conditions, aggregations, and temporal groupings, enabling targeted and efficient data retrieval from DTs via DAIRY. BSQL supports advanced filtering, sorting, and limiting of results, making it ideal for efficient operational monitoring as showcased in Section 5.

5 Evaluation

We evaluated our approach using a real-world setting: a Living Lab at the University of Innsbruck, equipped with a heterogeneous network of 32 sensors measuring environmental conditions (e.g., temperature, CO2, air quality), occupancy (PIR, radar), and imaging data (thermal array). The evaluation focuses on two aspects: (i) the performance and scalability of query execution (cf. Section 5.1), and (ii) a practical use case from building management, identifying rooms where temperature exceeds a predefined threshold (cf. Section 5.2). Following the star schema (cf. Section 4.1), his scenario utilized a data model with one dimension table for the sensors (`devices`), and two fact tables for both room conditions (`room_conditions`) and occupancy (`occupant_data`), respectively.

5.1 Technical Evaluation

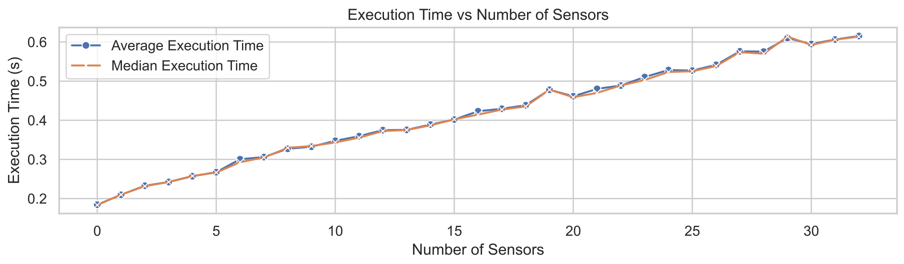

Fig. 1. Summary of query execution times.

To evaluate query efficacy, we benchmarked a hybrid query retrieving static model data (room location) and the most recent dynamic sensor payload for a variable number of sensors, from N=0 to 32. The primary execution overhead stems not from the delegated SQL and SPARQL queries, but from the preceding code generation and validation phase. This phase performs essential syntactic and semantic checks (e.g., verifying property IDs against the model) and incorporates optimizations such as reduced knowledge graph traversal and placeholder replacement for efficient SQL generation [18]. As confirmed by the results shown in Figure 1, the total execution time exhibits efficient linear scaling, validating the scalability of our architecture.

5.2 Operational Runtime Monitoring.

Our use case in a larger context addresses occupant comfort and safety [2] by querying the DT for rooms which temperature is not within a specified range (e.g., due to a broken heater). Doing so allows to adapt building operations continuously to better meet occupant comfort [2]. Specifically, we (i) check the model of the aforesaid Living Lab into DAIRY [23], (ii) create the relevant data model (cf. Section 4.1), and (iii) automatically collect data from the Living Lab in DAIRY [23]. Observe that for this the sensors in the Living Lab are configured to send data to the repository via MQTT. This establishes the foundational infrastructure of a DT of the building which subsequently can be used for operational runtime monitoring.

We employ our DT to identify rooms which temperature falls outside a specified range (observe that in our example we only have one room which however does not negatively impact the correctness of our results and later conclusions in Section 6). Given BSQL, we next formulated a query that allows to identify sensors in a building model and subsequently filter for those, whose temperature readings fall outside a specified range. By being located inside a room in the building model we can further extract room information via the sensor. This results in a list of sensors alongside their location (e.g., the room) in the building whose temperature readings fall outside the specified temperature range. Listing 3 (see p. 8) shows the corresponding query. First, in the `SENSOR_SELECT` part we extract relevant temperature sensors from the building model using `family` and `type`; in the subsequent `DATA_SELECT` part we then query for those sensors whose temperature measurements fall outside a specific range (e.g., 20 °C – 22 °C).

Listing 4 (see p. 8) shows the raw JSON reply as a list of sensors and their locations, cf. the rooms they are located in, falling outside the specified temperature range. In our example, `Room A` needs attention. Facility managers thus using a few known relevant sensors IDs can easily check and identify anomalies, and react appropriately (e.g., by fixing a broken heater). The total execution time of the query was $\sim$0.235 seconds once again showcasing the performance of our proposal in that it allows to establish near real-time monitoring cycles (ignoring any latency related to the retrieval of building data which is outside our control).

The query from Listing 3 readily can be adapted to other use cases, e.g., querying for occupied rooms where the temperature level reaches a critical threshold that could negatively affect an occupant's health, depending on the available sensors. As an example, Listing 5 (see p. 9) shows a query for identifying occupied rooms whose temperatures exceed a threshold of 30 °C.

```
SENSOR_SELECT (
    PROPERTY_SET 'Allgemein' PROPERTIES ('Room'),
    PROPERTY_SET 'Andere' PROPERTIES ('Familie und Typ')
)
WHERE {
    Sensor.Id IN ('41','42','43') AND
    'Andere'.'Familie und Typ' = 'Temperature_Sensor_Surface: XENSIV PAS CO2 - Temperature'
}
DATA_SELECT * WHERE {
    receiveTime BETWEEN '2024-12-30 23:00:00' AND '2024-12-31 00:00:00'
        [
            :'temperature' NOT_BETWEEN 20 AND 22 ON_SENSOR_WITH_ID '41',
            :'temperature' NOT_BETWEEN 20 AND 22 ON_SENSOR_WITH_ID '42',
            :'temperature' NOT_BETWEEN 20 AND 22 ON_SENSOR_WITH_ID '43'
        ]
}
ORDER_BY receiveTime DESC LIMIT 1
```

Listing 3: Hybrid query for retrieving all rooms whose current temperature readings fall outside specified ranges.

```
{"bsql_response":{"sensors":[{"static_dimension":{"Andere":{"Familie und
↪  Typ":"Temperature_Sensor_Surface: XENSIV PAS CO2 - Temperature"},"Allgemein":{"Room":"Room
↪  A"}},"dynamic_dimension":{"reading_count":1,"metadata":{"temperature":"DOUBLE","receive_time":
↪  "TIMESTAMP"},"sensor_readings":[{"temperature":17.74743080882481,"receive_time":"2024-12-31
↪  00:00:00.0"}]},"sensor_id":"42"},{"static_dimension":{"Andere":{"Familie und
↪  Typ":"Temperature_Sensor_Surface: XENSIV PAS CO2 - Temperature"},"Allgemein":{"Room":"Room
↪  A"}},"dynamic_dimension":{"reading_count":1,"metadata":{"temperature":"DOUBLE","receive_time":
↪  "TIMESTAMP"},"sensor_readings":[{"temperature":17.59644148648948,"receive_time":"2024-12-31
↪  00:00:00.0"}]},"sensor_id":"43"},{"static_dimension":{"Andere":{"Familie und
↪  Typ":"Temperature_Sensor_Surface: XENSIV PAS CO2 - Temperature"},"Allgemein":{"Room":"Room
↪  A"}},"dynamic_dimension":{"reading_count":1,"metadata":{"temperature":"DOUBLE","receive_time":
↪  "TIMESTAMP"},"sensor_readings":[{"temperature":17.598542091060978,"receive_time":"2024-12-31
↪  00:00:00.0"}]},"sensor_id":"41"}],"sensor_count":3},"execution_time":"PT0.235859844S"}
```

Listing 4: Raw JSON response to our query for operational runtime monitoring of room temperatures, showing rooms where sensor readings fall outside the specified range (cf. Listing 3). `Room A` is falling outside the range.

6 Discussion

Our work provides a concrete solution to the models-meet-data challenge [17] by introducing a hybrid query language that unifies static model representations with dynamic runtime data. In contrast to existing DT solutions that maintain separate infrastructures and rely on ad-hoc integration [2,21], our approach is grounded in established data integration principles [7]. By linking an ifcOWL-based building model with a SOSA/SSN-based data model, we establish a unified semantic framework that enables a single declarative language, BSQL, to retrieve data across these distinct paradigms using SPARQL and SQL.

In answering our research questions, we extended the DAIRY repository with two key design strategies: a clear architectural demarcation between model and data stores (cf. RQ1), and a lazy-linking mechanism for on-demand data fusion

```
SENSOR_SELECT (
    PROPERTY_SET 'Allgemein' PROPERTIES ('Room'),
    PROPERTY_SET 'Andere' PROPERTIES ('Familie und Typ')
)
WHERE {
    (Sensor.Id IN ('41','42','43') AND 'Andere'.'Familie und Typ' = 'Temperature_Sensor_Surface:
    ↪  XENSIV PAS CO2 - Temperature') OR
    (Sensor.Id IN ('55','56') AND 'Andere'.'Familie und Typ' = 'PIR_Sensor_Surface: HC-SR501')
}
DATA_SELECT * WHERE {
    receiveTime BETWEEN '2024-12-30 23:00:00' AND '2024-12-31 00:00:00'
        [
            :'temperature' > 30 ON_SENSOR_WITH_ID '41',
            :'temperature' > 30 ON_SENSOR_WITH_ID '42',
            :'temperature' > 30 ON_SENSOR_WITH_ID '43',
            :'occupant' = TRUE ON_SENSOR_WITH_ID '55',
            :'occupant' = TRUE ON_SENSOR_WITH_ID '56'
        ]
}
ORDER_BY receiveTime DESC LIMIT 1
```

Listing 5: Adapted query from Listing 3 to check from occupied rooms with a critical temperature level.

(cf. RQ2). Our evaluation demonstrates that this approach not only maintains consistency and operational efficiency but also ensures scalable query performance. The ability to handle advanced scenarios, such as temporal and semantic filtering, through a single interface simplifies the analytical processes for practitioners. This work therefore represents a significant step toward overcoming the fragmented data representations and synchronization challenges that have hindered the development of robust, integrated DT applications.

In direct contrast to related work (cf. Section 2), our approach establishes a unified semantic framework by linking an ifcOWL-based model with a SOSA/SSN-based data model. This enables a single declarative query language, BSQL, that applies formal schema integration principles rather than relying on ad-hoc mediation. By embedding this language within the DAIRY repository, our solution overcomes the limitations of fragmented data representations and inconsistent model-data synchronization seen in previous work. The key architectural decisions—separating model and data stores while enabling on-demand fusion via lazy-linking—ensure both scalability and practical applicability in resolving model and data integration in DTs.

6.1 Limitations and Future Work

Future work will address three key limitations. First, the expressiveness of BSQL is currently limited to properties directly defined via `IfcRelDefinesByProperties`; we plan to investigate property inference through path traversal [11] and OWL reasoning [12]. Second, we will work to mitigate the initial query compilation latency (cf. Section 5.1) by optimizing the query planning and code generation pipeline to better support highly dynamic settings. Finally, the generalizability of our approach will be assessed by extending our evaluation to other domains, such as industrial automation and healthcare, which is facilitated by the repository's underlying RDF-based architecture [22,23].

7 Conclusion

In our work, we presented a hybrid query language designed to address the challenge of integrating heterogeneous models with dynamic runtime data in DT engineering. We developed our approach within the context of the DAIRY repository, where static building models represented in ifcOWL are seamlessly connected to runtime data stored in a relational database through a unified semantic framework. By mapping static model information to SPARQL queries and runtime data to SQL queries within a single declarative query language, cf. BSQL, our solution not only simplifies access but also enhances consistency across different data sources.

Our evaluation, conducted in the context of building operations, revealed that the proposed hybrid query language is capable of efficiently retrieving and aggregating sensor data while maintaining scalable performance as the number of sensors increases. Overall, our contribution presents a solution to resolving the models-meet-data challenge in DTs. The unified querying approach developed in our work paves the way for more reliable and efficient integration of diverse data sources, facilitating real-time monitoring and comprehensive analysis without imposing the complexity of multiple, disjointed query interfaces.

Acknowledgments This research has received funding from the Austrian Research Promotion Agency (FFG) under Grant Agreements No.: 910225, BOREALIS, and 898708, TwinLight.

References

1. Bettini, L.: Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd (2016)
2. Hauer, M., Hammes, S., Zech, P., Geisler-Moroder, D., Plörer, D., Miller, J., van Karsbergen, V., Pfluger, R.: Integrating digital twins with bim for enhanced building control strategies: A systematic literature review focusing on daylight and artificial lighting systems. Buildings **14**(3), 805 (2024)
3. Iqbal, M.Z., Mustafa, G., Sarwar, N., Wajid, S.H., Nasir, J., Siddque, S.: A review of star schema and snowflakes schema. In: Int. Conf. on Intelligent Technologies and Applications. pp. 129–140. Springer (2019)
4. Jia, W., Wang, W., Zhang, Z.: From simple digital twin to complex digital twin Part I: A novel modeling method for multi-scale and multi-scenario digital twin. Advanced Engineering Informatics **53**, 101706 (2022)
5. Koren, I., Jansen, N., Michael, J., Rumpe, B., Böse, E.: A Low-Code Approach for Data View Extraction from Engineering Models with GraphQL. In: Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C). ACM/IEEE (2023)

6. Launchbury, J.: A natural semantics for lazy evaluation. In: 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 144–154 (1993)
7. Lenzerini, M.: Data integration: A theoretical perspective. In: 21st ACM SIGMOD-SIGACT-SIGART Symp. on Principles of database systems. pp. 233–246 (2002)
8. Liu, X., Jiang, D., Tao, B., Xiang, F., Jiang, G., Sun, Y., Kong, J., Li, G.: A systematic review of digital twin about physical entities, virtual models, twin data, and applications. Advanced Engineering Informatics **55**, 101876 (2023)
9. Michael, J., Bork, D., Wimmer, M., Mayr, H.: Quo Vadis Modeling? Findings of a Community Survey, an Ad-hoc Bibliometric Analysis, and Expert Interviews on Data, Process, and Software Modeling. Journal Software and Systems Modeling (SoSyM) **23**(1), 7–28 (2024)
10. Michael, J., David, I., Bork, D.: Digital Twin Evolution for Sustainable Smart Ecosystems. In: MODELS Companion '24: Int. Conf. on Model Driven Engineering Languages and Systems. pp. 1061–1065. ACM (2024)
11. Mojžiš, J., Laclavík, M.: Srelation: fast rdf graph traversal. In: Knowledge Engineering and the Semantic Web: 4th Int. Conf., KESW 2013. Proc. 4. pp. 69–82. Springer (2013)
12. Polleres, A., Hogan, A., Delbru, R., Umbrich, J.: Rdfs and owl reasoning for linked data. In: Reasoning Web International Summer School, pp. 91–149. Springer (2013)
13. Qiang, Z., Hands, S., Taylor, K., Sethuvenkatraman, S., Hugo, D., Omran, P., Perera, M., Haller, A.: A systematic comparison and evaluation of building ontologies for deploying data-driven analytics in smart buildings. Energy and Buildings **292**, 113054 (2023)
14. Tao, F., Xiao, B., Qi, Q., Cheng, J., Ji, P.: Digital twin modeling. Journal of Manufacturing Systems **64**, 372–389 (2022)
15. Tao, F., Zhang, H., Zhang, C.: Advancements and challenges of digital twins in industry. Nature Computational Science **4**(3), 169–177 (2024)
16. Ullman, J.D.: Information integration using logical views. In: International Conference on Database Theory. pp. 19–40. Springer (1997)
17. Van Den Brand, M., Cleophas, L., Gunasekaran, R., Haverkort, B., Negrin, D.A.M., Muctadir, H.M.: Models Meet Data: Challenges to Create Virtual Entities for Digital Twins. In: 2021 ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 225–228 (2021)
18. Wang, P., Shi, T., Reddy, C.K.: Text-to-sql generation for question answering on electronic medical records. In: Web Conference 2020. pp. 350–361 (2020)
19. Wiederhold, G.: Mediators in the architecture of future information systems. Computer **25**(3), 38–49 (1992)
20. Wieringa, R.: Design Science Methodology for Information Systems and Software Engineering. Springer (2014)
21. Zech, P., Clark, T., Breu, R.: An Empirical Analysis of Digital Twin Adoption. In: 58th Hawaiin Int. Conf. on System Sciences (HICSS'58). pp. 6253–6262 (2025)
22. Zech, P., Fröch, G., Breu, R.: A requirements study on model repositories for digital twins in construction engineering. In: Int. Conf. on Cooperative Information Systems. pp. 459–469. Springer (2023)
23. Zech, P., Pobitzer, P., Fröch, G., Breu, R.: A Proposal for a Models-Meet-Data Repository For Digital Twins in Construction Engineering. In: IEEE 21st Int. Conf. on Software Architecture Companion (ICSA-C). pp. 111–118. IEEE (2024)
24. Zhang, H., Qi, Q., Tao, F.: A multi-scale modeling method for digital twin shop-floor. Journal of Manufacturing Systems **62**, 417–428 (2022)