Lazy Minimal Data Warehouse Refresh

Mikko Puonti
and Timo Raitalaakso
Solita
Tampere, Finland
puonti@iki.fi
timo.raitalaakso@iki.fi

Timo Aho
Tietoevry
Tampere, Finland
timo.aho@iki.fi

Toni Taipalus
Tampere University
Tampere, Finland
toni.taipalus@tuni.fi

***Abstract*—Data warehousing is a technique for integrating data from several source systems to enable pervasive data analytics. Data updates in the source systems result in data refreshes down the data stream. These data refreshes are potentially both computationally and financially costly, even if the refreshed data is not utilized before the next refresh. In this study, we present an accessible approach for selecting tables, views and materialized views for data refreshes in situations where entities have complex dependencies. While such solutions have been proposed in scientific literature in the past, their practical applications have been few and far between. We speculate that this gap between theory and practice stems from the highly theoretical presentation of such solutions. In this study, we aim to address this gap from a practice-oriented industry perspective. Utilizing this approach may, depending on the structure of the database and how it is used, present considerable improvements for the efficient utilization of, e.g., computation and networking resources, as well as energy efficiency.**

***Index Terms*—data warehouse, efficiency, data ingestion, data loading, data engineering**

I. Introduction

Data warehousing is a technique for integrating data from several organizational source data systems such as human resources, customer relationship management, and end-user facing services, to a single environment where data may be analyzed. The primary steps in this integration of data from several sources are often called *extract*, *transform* and *load* (ETL). These steps attempt to ensure that the data objects are indeed correct, they are organized in a way that serves data analytics, and there is only one version of crucial data objects [1].

Data loads from the source systems into the data warehouse can be computationally expensive, as the data objects are typically sent over a network and inserted into the data structures of the data warehouse. Consequently, this often results in updating several database *entities* such as tables and materialized views. Updating database entities periodically can result in wasteful use of computing resources, if these entities are not used (e.g., in data analytics queries) between data updates [2]. This is especially true in the cases when the load rebuilds a database entity from scratch instead of merely refreshing what has changed. In practice, this kind of waste is common: data loads are frequently scheduled daily or weekly, regardless of actual data usage.

In this study, we demonstrate an approach intended for reducing unnecessary data loads in database entities with complex dependencies with each other. In short, the approach includes automatically selecting a minimal subset of entities to update for the given use case. This facilitates more cost-effective data warehousing, especially in cases where database entities have complex dependencies. Such cases may be common in domains where the data warehouse is utilized by many stakeholders with different data needs. Our contribution is to simplify and propagate the solution from a practical perspective, in hopes of communicating the importance of implementing such solutions widely in practice.

The rest of this study is structured as follows. In Section II, we examine the current state of research on data refreshing approaches, and position our study with prior works. In Section III, we present the approach for database entity selection and its constituents. Section IV contains the practical and theoretical implications, and Section V concludes the study.

II. Background

A. Database entities in data warehouses

In the scope of this study, the data warehouse architecture consists of source systems, a staging area, the enterprise data warehouse, and data marts (cf. Fig. 1). The ETL process extracts new data objects from source systems which can each follow a different data model, have conflicting ways of storing similar data objects, and have distinct nomenclature [3]. These data objects are then loaded into a staging area, which is often a set of temporary data structures. Next, the data objects are transformed, i.e., cleaned, organized, and standardized, before they are loaded into the enterprise data warehouse data structures, which are often structured for data analytics or some other integration purpose [4]. Once the data is successfully loaded into the enterprise data warehouse, the staging area is typically cleared out to free up space for the next ETL cycle. A typical data warehouse environment also includes one or more data marts. Data marts are either physical or logical ways of separating subsets of the data warehouse for different stakeholders.

Data warehouses, especially those built upon relational database technologies, typically have at least three separate

979-8-3315-4322-8/25/$31.00 ©2025 IEEE

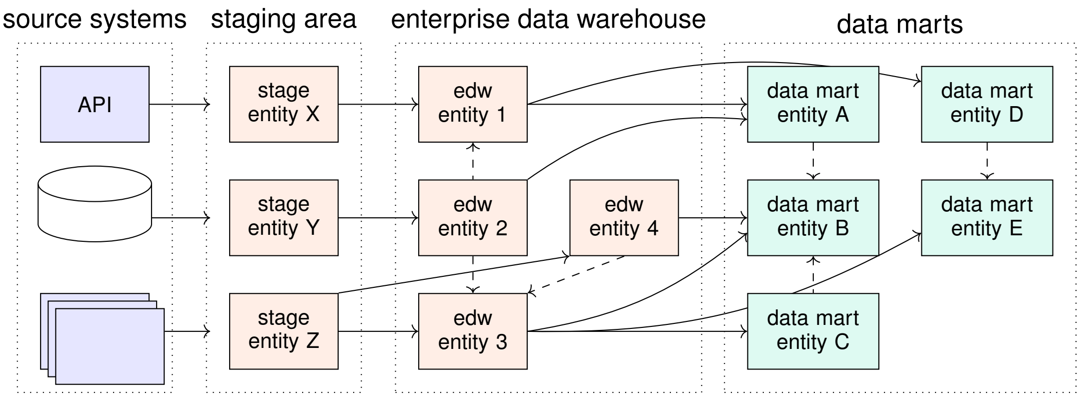

Fig. 1. General data warehouse architecture in the scope of this study; arrows represent data flow from source systems to data marts; dashed arrows represent foreign key dependencies from parent entity to foreign key; the arrows may also be interpreted as load order dependencies, e.g., the data mart entity D depends of the *edw entity 1*

logical ways to store the data, i.e., database tables, views and materialized views. We use the term database *entity* collectively for all three and other possible similar logical data structures. These three types of entities have some differences in the context of our study. First, a database view is a stored database query which can be used to simplify other queries. A view can refer to other views, tables or materialized views. A view does not itself store data and is, hence, automatically refreshed when the view is queried. A materialized view, on the other hand, stores pre-calculated datasets, making the use of these datasets computationally faster. However, a materialized view often does not refresh automatically but needs explicit refreshing. Finally, tables store database data, and enable the use of views and materialized views. In addition, instead of a refresh, the data in tables are modified with database operations such as inserts, updates, and deletes. Refreshing data in tables typically requires more effort and is more computationally intensive.

As tables and materialized views can contain stale data, they are in the center of our study. We call tables and materialized views *materialized entities* if it is important to separate them from views. Tables and materialized views offer potentially larger performance gains than views, as their contents are not calculated at query runtime, but beforehand [5]. Some work also exist on selecting which views to materialize, i.e., which views should be transformed to materialized views for performance gains [6].

B. Approaches to data loading

As data changes in the source systems, the data loads are pushed downstream into the staging area and into the entities of the data warehouse and data marts. Sometimes, especially in the case of complex computations, the entities are rebuilt from scratch when the data changes. However, modern environments typically utilize a *fast refresh* or *delta update* approaches, where only the changes are calculated and applied to the entities instead of a complete rebuild [2].

There are several methods of refreshing materialized entities. First, a *manual refresh* refreshes the materialized entity when explicitly requested [7]. Second, a materialized entity can be refreshed *on demand* [8], i.e., according to a schedule, or periodically. Third, materialized entities may be refreshed *on commit* [9], meaning that each committed transaction in the source systems also refreshes the whole downstream up to the materialized entity. Fourth, *immediate* [10] refreshes imply refreshing materialized entities as soon as the source data changes. Fifth, in the case of *lazy updates*, a materialized entity is refreshed only when someone queries it [11], [12]. The last approach requires both a log to track whether the data have changed, as well as a transaction control mechanism to apply the changes correctly [9].

C. Dependencies between entities

A database entity may *depend* on one or several entities of the data warehouse environment. There are at least three types of dependencies in database entities. First, views and materialized views refer to other entities in their definitions. Additionally, there might be logical dependencies between tables and other entities as the data in a table is updated with a load from the other entities. Moreover, entities might have constraints that relate to other entities. A typical example is a foreign key definition between two entities.

All three dependencies restrict the refresh order of the entities. If entities are refreshed in an incorrect order, the database may end up with either stale data, or data objects that violate database constraints. Thus, it is essential to account for these types of dependencies when deciding the order of entity refreshes. This network of dependencies is essential in describing the problem and it is natural to illustrate the network as a graph.

D. Comparison with other approaches

The order of the entity refreshes can be expressed as finding a data lineage type graph path to a given database entity, and there exists several graph theoretical formulations for constructing the refresh order [1], [2], [13], [14]. However, instead of describing the problem formally using graph theory, we explain the construction of the ordered graph in a simple

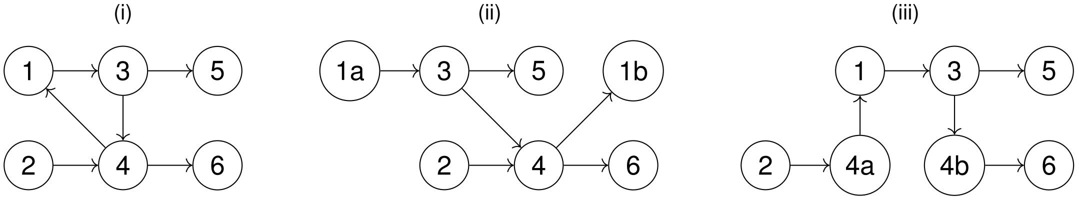

Fig. 2. Example of a cyclical graph in (i), and possible acyclic outcomes of heuristic splitting in (ii) and (iii)

way that can be implemented with relative ease on readily available programming libraries in a multitude of languages.

III. Implementation

A. Premise

As explained in the previous section, reducing unnecessary computation is often desired. Traditionally, data loads are executed from the source systems to the staging area, then to the data warehouse, and finally into the data marts. That is, when data objects change in the source systems, the updates are *pushed* down the data stream. However, as discussed in Section II-B, downstream data structures can also be refreshed only when needed. That is, in *lazy* fashion. Thus, our method focuses on lazy updates with a *pull*-type approach, according to which the needed data are pulled with specific chain of loads from the staging area (or even source systems) when up-to-date data is needed.

In our study, a query to a database entity initiates the *pull* operation, which only refreshes the necessary database entities. For brevity, in this study we assume that the refresh is initiated by a query to a database entity. This is not necessarily the case, and the method can also be utilized with other approaches described in Section II-B. Moreover, often multiple entities, like a full data mart, are refreshed at the same time.

B. Representing data loading as a load graph

When implementing a data load dynamically, we first choose an ordered set of entities to be refreshed. Data loads can be illustrated as directed graphs by expressing entities as nodes, and loads as edges. The direction of an edge defines the source and target of the load, or alternatively, a foreign key constraint where the primary (i.e., referenced) table has a directed edge to the secondary (i.e., referencing) table.

Typically this graph is also layered in the way that the nodes in each layer only have edges to the nodes in the same or next layer. For example, enterprise data warehouse entities typically do not refer to data mart entities.

To efficiently utilize computing resources, we try to minimize the amount of needed loads by finding the minimal load subgraph of the whole data flow graph (such as the one in Fig. 1). Both the full load graph and the minimal load subgraph are typically acyclic, making them *directed acyclic graphs*. However, in the next section, we explain how to treat possible cycles in the graph.

C. Dealing with cyclical data loads and graphs

There are some aspects that make representing a data load flow cyclic in nature. For example, a refresh of a database entity might cause its own refresh later. Understandably, executing an endless loop of data loads is not feasible and needs to be solved in the design. A more typical cause of cycles are database constraints such as foreign keys. A special case of this are so-called *facing foreign keys* where two entities have both foreign key references to each other. Adding more data to a database with this kind of cyclical constraints typically involves workarounds such as disabling foreign key constraints temporarily.

Hence, we need to define some heuristic for removing the cycles. Typical heuristics include executing every refresh at most once. In the case of database constraints, their validity can be checked only afterwards.

From load graph perspective, the heuristics can be expressed as removing edges or splitting the entity nodes into two or more. Some splitting examples are presented in Fig. 2. It is important to note that a heuristic does not necessarily yield unique results. For example, the cyclical graph (i) in Fig. 2 is modified with node splitting to produce the alternative acyclic graphs (ii) and (iii).

D. Selecting entities for a refresh

We now present our approach as an algorithm which gives the ordered load list of a given load graph. The selection of database entities for a refresh using the *pull*-type loading takes the data flow load graph (*graph G*) and the queried entity (*node TargetNode*) as inputs. The algorithm outputs an ordered set of database refreshes (*OrderedList*) as detailed in Algorithm 1. We call the approach of lazily selecting the ordered list of entities *pull-type deep refresh* of an arbitrary database entity. There are two main challenges to this. First, deciding which entities are necessary for a particular load subgraph, and second, deciding the suitable order for updated entities to ensure that the final result is both correct and up-to-date.

The first challenge can be solved by reverting the edges of load graph and doing full graph traversal from the target entity *TargetNode*. That is, we visit all the nodes that can be reached from *TargetNode*. Graph traversal can be done by typical graph search algorithms such as a depth-first search. That is, when an entity (i.e., node in the graph) is queried, we create a directed depth-first search on the reverted load graph from the node and return the traversed load subgraph.

Algorithm 1 Select database entities for *pull-type deep refresh* load

```
Input: A directed graph $G$ with database entities as nodes. Edges consist of load dependencies and foreign key references.
Input: A database entity node $TargetNode$ to be refreshed
Output: An ordered list of database entities to be refreshed
$G_{dag} \leftarrow$ graph $G$ with cycles removed by a given heuristic
$G_{rev} \leftarrow$ graph $G_{dag}$ with reverted edges
$S \leftarrow$ subgraph of the traversed nodes and edges given by Depth-first search from $TargetNode$ for graph $G_{rev}$
$OrderedList_{rev} \leftarrow$ ordered list of entity nodes by Topological sorting of graph $S$
$OrderedList \leftarrow$ reversed $OrderedList_{rev}$
return $OrderedList$
```

The second challenge can be solved by utilizing *topological sorting* of the load subgraph nodes. Graph topological sorting gives an ordered list of nodes in which all predecessors of a node are located earlier in the ordering. That is, if there is a directed graph path from a node a to node b, topologically ordered list has a always before b. Topological ordering is used in, e.g., scheduling problems to give an order of jobs to execute. Topological sorting can be done with readily available algorithms [15], [16].

In Algorithm 1, we combine the aforementioned steps. Fig. 3 illustrates the algorithm with a practical example. Given the ordered list of entities of the load subgraph, we have everything to execute *pull-type deep refresh* of the *TargetNode*. It is important to note that both the graph search and the topological sorting algorithms are widely available in various programming languages and libraries. For example, with Python *NetworkX* library, the Algorithm 1 could be implemented as follows.

```python
import networkx as nx
g_rev = g_dag.reverse()
s = nx.dfs_postorder_nodes(
      g_rev, source=target_node
    )
return nx.topological_sort(
         g_rev.subgraph(list(s))
       ).reverse()
```

IV. Discussion

A. Discussion on related work

The presented `pull`-type deep refresh described in this study can be considered analogous to lazy evaluation in many programming frameworks. Lazy evaluation postpones the processing to as late state as possible. This often reduces the needed overall processing — a large part of processing can often be skipped. Similarly, if a large part of the data warehouse is never used, the amount of refreshes can be reduced. At least one study showed that lazy updates coupled with “lazy updates when resources are available” had significant efficiency gains [9]. However, lazy evaluation also has typical drawbacks which include difficulties in estimating when the processing should happen and how heavy an operation it would be.

As mentioned before, describing data flows as graphs and detecting the data lineage for refreshing only the needed subgraph is not a novel idea [2], [13], [14]. However, our contribution is to explain the process in a way that can be implemented with readily available programming libraries instead of describing the problem formally.

All in all, we would like to emphasize that practical applications of selecting entities for refreshes, at least to the best of our knowledge, are scarce in data warehousing products, and our motivation for this study is rooted in accessibly communicating a solution to an old challenge. In our experience, this is a frequent and important challenge where theory has yet to be widely applied in practice, despite the age of proposed solutions.

B. Practical implications

The approach described in this study has practical implications of reducing unnecessary use of computing resources. We recommend the approach to be used with delta updates, i.e., by using the deep refresh approach to select the entities to be refreshed, and refreshing only the deltas for these entities instead of all data. However, how much this approach affects the utilization of computing resources is highly dependent on domain-specific details such as how many entities there are, what are their dependencies and foreign key relationships, how often the staging area would be used to refresh the tables, and how often the database entities are queried.

Although we have approached the subject from the point of view of data warehouses where materialized entities are relatively common, the presented approach is not limited to data warehouses. The approach is also relevant in, e.g., data lakehouses [17], transactional databases, or even data lakes as long as they have similar graph-like loading patterns.

Since efficient data analytics platforms have been one of the core themes in both industry and research [18], [19], we believe that the popularization of such an approach will be of interest to platform developers as well.

C. Technical considerations

Our load graph might not describe the loads in full because it enables a single load edge between two entity nodes. In practice, a single load might have multiple source and/or target entities. This can be solved by splitting the multi-entity load into multiple one-to-one edges.

Additionally, we might have, in multigraph fashion, multiple loads between two nodes. If the order of the loads does not matter, we can treat them as a single graph edge. However,

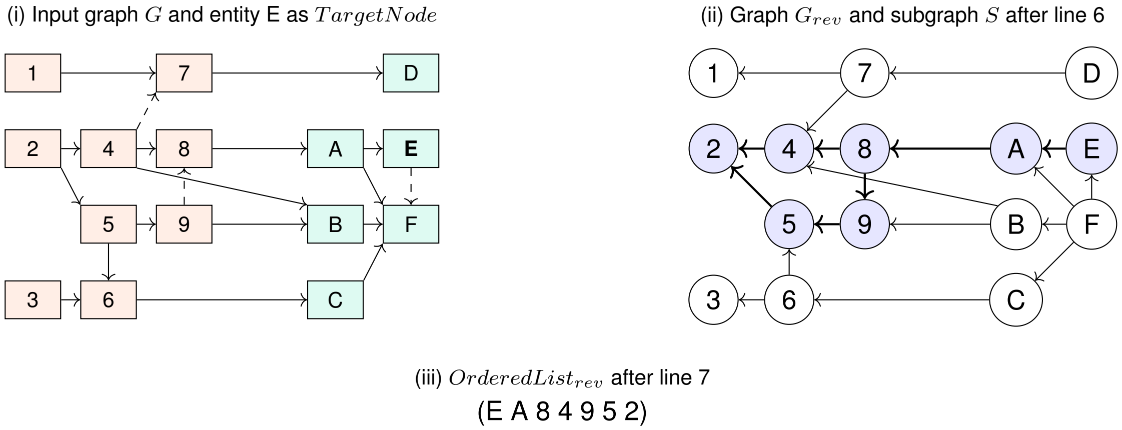
(i) Input graph G and entity E as $TargetNode$
1
7
D
2
4
8
A
E
5
9
B
F
3
6
C
(ii) Graph G_{rev} and subgraph S after line 6
1
7
D
2
4
8
A
E
5
9
B
F
3
6
C
(iii) $OrderedList_{rev}$ after line 7
(E A 8 4 9 5 2)

Fig. 3. An example of the load graph (i): solid lines represent load dependencies and dashed lines foreign key relationships; rectangles represent database entities with numbers indicating EDW level and letters data mart level; the bolded *E* represents the database entity node to be refreshed; based on this, a corresponding load subgraph (ii) is generated; thick edges and blue nodes are included in the subgraph; the topological sorting of the subgraph gives the ordered list (iii) which is a reverse of the final refresh order; the algorithm output and the final refresh order of entities is (2 5 9 4 8 A E); note that also (2 4 5 9 8 A E) and (2 5 4 9 8 A E) are similarly valid results

if the internal ordering is crucial, additional steps will be necessary. We can incorporate the loads themselves as nodes in the load graph and treat them similar to other nodes. Ultimately, our ordered list of nodes will include both entities to be refreshed and loads to be executed. Otherwise, the process remains the same.

D. *Limitations and future work*

There are some limitations and considerations regarding this work. First, we do not provide a way to generate the initial data flow load graph. For this graph to be constructed, we need information on the entity dependencies which can be extracted from the database metadata. The metadata has typically such information as entity names, foreign key constraints, and entity query definitions. The dependency graph can be constructed relatively effortlessly given the amount of theoretical [20] and practical prior work [21]. However, the implementations might be database management system specific due to different ways of storing metadata.

The current version of our algorithm is for selecting an ordered subset of entities based on a single target entity to be refreshed. In practice, there might be need to update multiple entities at once: a typical example being updating all data mart entities. For this, we can do the graph traversal search from all the relevant entities as *TargetNode* and treat the visited nodes as the load subgraph S.

Moreover, in this study we concentrated on reducing the use of computing resources by minimizing unnecessary refreshes. However, load flow optimization also has other aspects like fault tolerance, network latency and overall cost [22].

Finally, our algorithm does not consider the cut-off point from which point on the refreshes are not considered. That is, it is up to the data warehouse environment implementation to decide whether the leaf nodes of the load subgraph (cf. Fig. 3(ii)) reside in the enterprise data warehouse, staging area, or somewhere else. This is a willful abstraction, as different environments function with different needs. In summary, the proposed approach does not account for several possible, implementation-specific intricacies. We have limited the scope by design to account for the challenge we deemed prominent in efficient data loads.

For future work, it should be considered whether it is feasible to further develop the algorithm to account for changes and load dependencies on column-level instead of the level of entities [14]. Especially for wide tables that have a large number of seldom-used column, this approach could further improve the efficiency. Such dynamic maintenance has been proposed for simple views before [23]. Furthermore, we expect to consider how our approach can be integrated with other aspects of efficient entity management, such as optimizing memory [24] and indices [25], [26] for analytics. Another natural further research avenue is concurrent refreshing [27], which we have not discussed in this study.

V. Conclusion

Many data warehouses contain complex relationships between the staging area data structures, enterprise data warehouse data and data marts. In this work, we described a *pull*-type refresh for selecting database entities to be refreshed, when a specific entity is queried. The aim of the approach was to reduce the number of data loads in the system, and load new data only to the data objects where new data are needed. We have aimed to communicate the approach in an accessible way, highlighting the gap between computer science theory and database industry practice, and showing that bridging this gap may not be a particularly arduous undertaking.

Acknowledgements

The authors would like to thank Solita and Tietoevry for the possibility to conduct this research. This study was partly funded by the DigiSus research platform.

References

[1] A. Simitsis, P. Vassiliadis, and T. Sellis, "Optimizing ETL processes in data warehouses," in *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. IEEE, 2005, pp. 564–575.

[2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa, "Efficiently updating materialized views," *ACM SIGMOD Record*, vol. 15, no. 2, pp. 61–71, 1986.

[3] Z. El Akkaoui, E. Zimànyi, J.-N. Mazón, and J. Trujillo, "A model-driven framework for etl process development," in *Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP*. ACM, 2011, pp. 45–52.

[4] S. Dessloch, M. A. Hernández, R. Wisnesky, A. Radwan, and J. Zhou, "Orchid: Integrating schema mapping and ETL," in *IEEE 24th International Conference on Data Engineering (ICDE)*. IEEE, 2008, pp. 1307–1316.

[5] Y. Pang, L. Zou, J. X. Yu, and L. Yang, "Materialized view selection & view-based query planning for regular path queries," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–26, 2024.

[6] H. Gupta and I. S. Mumick, "Selection of views to materialize in a data warehouse," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 1, pp. 24–43, 2005.

[7] Y. Kotidis and N. Roussopoulos, "A case for dynamic view management," *ACM Transactions on Database Systems (TODS)*, vol. 26, no. 4, pp. 388–423, 2001.

[8] U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson, "Data integration flows for business intelligence," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM, 2009, pp. 1–11.

[9] J. Zhou, P.-A. Larson, and H. G. Elmongui, "Lazy maintenance of materialized views," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007, pp. 231–242.

[10] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey, "Algorithms for deferred view maintenance," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, 1996, pp. 469–480.

[11] C. Thomsen, T. B. Pedersen, and W. Lehner, "Rite: Providing on-demand data for right-time data warehousing," in *IEEE 24th International Conference on Data Engineering (ICDE)*. IEEE, 2008, pp. 456–465.

[12] W. Qu and S. Dessloch, "A real-time materialized view approach for analytic flows in hybrid cloud environments," *Datenbank-Spektrum*, vol. 14, pp. 97–106, 2014.

[13] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos, "Modeling etl activities as graphs." in *DMDW*, vol. 58, 2002, pp. 52–61.

[14] Y. Cui and J. Widom, "Lineage tracing for general data warehouse transformations," *The VLDB Journal*, vol. 12, no. 1, pp. 41–58, 2003.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[16] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[17] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics," in *Proceedings of CIDR*, vol. 8, 2021, p. 28.

[18] X. Tang, C. Chai, D. Zhao, H. Ma, Y. Zheng, Z. Fan, X. Wu, J. Zhang, R. Zhang, D. Li, Y. He, K. Huang, G. Meng, Y. Wang, Y. Zhou, T. Tao, L. Jian, J. Shu, Y. Wang, Y. Yuan, G. Wang, and G. Li, "Separation is for better reunion: Data lake storage at huawei," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 5142–5155.

[19] H. Yamada, M. Kitsuregawa, and K. Goda, "Lakeharbor: Making structures first-class citizens in data lakes," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 5583–5592.

[20] M. Arenas, R. Fagin, and A. Nash, "Composition with target constraints," in *Proceedings of the 13th International Conference on Database Theory*. ACM, 2010, pp. 129–142.

[21] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," *The VLDB Journal*, vol. 22, pp. 73–98, 2013.

[22] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal, "Optimizing analytic data flows for multiple execution engines," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2012, pp. 829–840.

[23] A. Gupta, I. S. Mumick *et al.*, "Maintenance of materialized views: Problems, techniques, and applications," *IEEE Data Eng. Bull.*, vol. 18, no. 2, pp. 3–18, 1995.

[24] Z. Li, X. Pi, and Y. Park, "S/c: Speeding up data materialization with bounded memory," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 1981–1994.

[25] P. Boncz, Y. Chronis, J. Finis, S. Halfpap, V. Leis, T. Neumann, A. Nica, C. Sauer, K. Stolze, and M. Zukowski, "Spa: Economical and workload-driven indexing for data analytics in the cloud," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 3740–3746.

[26] W. Zhang and K. A. Ross, "Permutation index: Exploiting data skew for improved query performance," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1982–1985.

[27] E. A. Lee and E. Matsikoudis, "The semantics of dataflow with firing," *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pp. 71–94, 2009.