Extended version: NoSQL Schema Optimization for Time-Dependent Workloads

Yusuke Wakuta
CyberAgent, Inc.
Tokyo, Japan
wakuta_yusuke@cyberagent.co.jp

Teruyoshi Zenmyo
Suntory Wellness Limited
Tokyo, Japan
tzenmyo@gmail.com

Lucas Trapon
Polytech Nantes
Nantes, France
lucas.trapon@etu.univ-nantes.fr

Yuya Sasaki
Osaka University
Osaka, Japan
sasaki@ist.osaka-u.ac.jp

Michael Mior
Rochester Institute of Technology
Rochester, NY, USA
mmior@mail.rit.edu

Makoto Oniziuka
Osaka University
Osaka, Japan
oniduka@ist.osaka-u.ac.jp

ABSTRACT

Predefined or predictable patterns in database workloads are common in automated IoT applications or scientific analysis. These patterns are used to enhance database system performance through adaptive database migrations, optimizing the database for the expected workload at any given time. In this work, we consider the problem of schema optimization for time-dependent workloads in NoSQL databases. Our contributions are threefold. First, we formulate the optimization problem of time series schema design as a single integer linear program (ILP) to minimize the total cost of time-dependent workload execution and database migration. Second, we propose a novel technique for pruning candidate schemas by decomposing the ILP designed for the original time-dependent workload via approximation into a hierarchy of local ILPs for smaller sub-workloads. Finally, we propose an effective technique that reduces migration plan candidates. The experiments confirm that our proposal is superior to static schema optimization methods for typical time-dependent workloads, reducing cumulative latency by up to 40% across all time steps. Furthermore, our pruning technique demonstrates remarkable scalability with the number of time steps, diverging from the NP-complete nature of a naive approach that scales almost quadratically with the number of time steps.

1 INTRODUCTION

Frameworks for big data management are widely used in many applications, such as Web services, IoT applications, and scientific analysis. These applications need to manage petabyte-scale data. In particular, NoSQL databases are an important class of systems used for such large-scale big data management. Historically, many NoSQL databases have roots in systems such as BigTable [7], Amazon Dynamo [11], and Yahoo! PNUTS [9]. Some recent examples of NoSQL databases are Google F1 [27] and Spanner [10]: most of these systems are classified as wide-column stores (or extensible record stores), a type of NoSQL database.

In most of the above applications, workloads follow common time-dependent patterns, such as cycles, workload evolution, and growth and spikes [22]. We focus on predefined or predictable patterns that are common features in automated IoT applications and scientific analysis. Specifically, we describe two such examples.

IoT applications: An electric power company uses an analytical pipeline to collect power usage from 7.5 million smart meters, places them in a distributed database system, aggregates the power usage, computes the total cost, and then notifies electrical power retailers in a timely fashion [26].

Astronomical data analysis: Astronomers use an analytic pipeline to capture images, transform data, calibrate parameters, identify objects, and detect transients and variables. The National Astronomical Observatory of Japan provides a web service to support such pipelines for astronomers [31]. The database stores 436 million objects in a distributed database system [1]. A large number of different queries are executed on the database depending on the type of analysis task. An important feature of these automated analysis pipelines is that they form predictable patterns: workloads are predefined in advance, and they are repeated for a certain time period, such as every day or every week.

Our research goal is to improve the performance of database systems by optimizing time series schema design and adaptively making database migrations leveraging these predictable patterns.

Technical trend and Major issues: Schema design is crucial to achieving high performance for large-scale big data management when using NoSQL databases. Significant research has been conducted on automated schema design for relational databases [3, 4, 19, 32], NoSQL [23, 24, 30], and cloud-scale environments [15, 18]. Since our target is large-scale big data management, we focus on the technical trend of schema design techniques for NoSQL (see Section 6 for relational databases). Most existing work assumes that the workload is static and does not change over time. To support changing workloads, existing approaches may use an average workload aggregated from a dynamic time-dependent workload. However, this approach can be ineffective for time-dependent workloads, since the average of such a workload may not be a good approximation for the workload at any given point in time. Other work considers dynamic workload changes [2, 5, 13, 24, 25], however, to the best of our knowledge, no work simultaneously optimizes the cost of both execution and database migration for time-dependent workloads.

Technical challenges: To tackle the weaknesses of existing techniques, we take an approach to optimizing time series schema by considering both the execution cost of time-dependent workloads and any necessary database migrations. However, if we naively extend existing techniques designed for static workloads to dynamic time-dependent ones, we have three obstacles to finding optimal

answers: 1) the trade-off between the cost of time-dependent workload execution and database migration, 2) the increased number of schema candidates that increases according to the number of time steps, and 3) the number of migration plan candidates (possible approaches to migrate data between schemas) that increases depending on the number of queries and schema candidates. The latter two factors substantially increase the optimization time required for designing a time series schema.

Contributions: We propose efficient techniques to optimize time-dependent workloads by reducing the number of schema candidates and the number of migration plan candidates. The novelty of our proposal is threefold. First, we formulate the optimization problem of time-dependent schema design with a single integer linear program (ILP) to minimize the total cost of time-dependent workload execution and database migration. Second, we propose an efficient schema candidate pruning technique by introducing a novel data structure, which we call a workload summary tree. We decompose the large ILP dervied from the original time-dependent workload via approximation into a hierarchy of local ILPs of smaller sub-workloads. Our approach effectively prunes column families that will never be chosen as ILP solutions at any node in the tree. This technique is scalable as each local ILP works efficiently with a small number of time steps while continuing to capture global features of its parent workload. Finally, we propose an effective technique to reduce the number of migration plan candidates. Note that workload changes can cause changes in optimized schema designs that necessitate a database migration. We can effectively reduce the number of migration plan candidates by restricting them according to how optimized query plans are changed instead of considering the general case of migrating data between two arbitrary schemas.

Paper organization. The rest of this paper is organized as follows. We describe the problem statement and the challenges of the schema design problem for time-dependent workloads (Section 2) and then formulate a solution with a single integer linear program (Section 3). We describe the detail of our approach (Section 4). We validate the effectiveness of our approach through intensive experiments using a variant of the TPC-H benchmark extended to various time-dependent workloads (Section 5). Finally, we position our proposal with respect to the state of the art (Section 6) and give concluding remarks (Section 7).

2 PRELIMINARY

In this section, we describe the schema optimization problem for time-dependent workloads on NoSQL databases, in particular on extensible record stores [6].

2.1 Extensible Record Store

An extensible record store, such as Apache Cassandra [21], is a general type of NoSQL database that achieves high scalability by partitioning the database into multiple distributed servers. Extensible record stores utilize column families (CFs) to express the database schema. We treat CFs as a physical schema, which is derived from a conceptual schema expressed with an entity graph [23] (simplified ER model). CFs contain three types of columns, *partition keys*, *clustering keys*, and *values*. Partition key columns are used for partitioning over multiple servers. Clustering key columns serve as a sorting key used to arrange records within each server. Other columns are treated as values. CF is expressed in the following notation:

$$[\textit{partition keys}][\textit{clustering keys}] \rightarrow [\textit{values}]$$

This notation also implies a functional dependency from the combination of *partition keys* and *clustering keys* to *values*.

2.2 Time-dependent Workloads

We focus on predefined or predictable time-dependent workload patterns, such as cycles, workload evolution, and growth and spikes, which are common patterns in automated IoT applications and scientific analysis [22]. In such workloads, all query and update operations are often known or predictable in advance. A time-dependent workload W denotes a collection of SQL statements, queries Q and update operations U, on a conceptual (relational) schema, and only the frequency of each query/update operation changes over time[1]. Since the frequency of queries and updates changes, we may need database migrations to modify the schema to one that is more efficient to reduce the execution cost of the workload. For example, a more denormalized schema should be chosen when the workload is read-intensive while normalization is preferred for write-intensive workloads. This motivates us to optimize time series physical schema over time-dependent workloads.

2.3 Problem statement

Given a time-dependent workload, a conceptual schema, and a maximum storage size as a constraint, we identify an optimized time series physical schema (set of column families) by minimizing the total cost of workload execution and database migration. As a result of schema optimization, we also output optimized query plans using the optimized column families at each time step and migration plans that migrate data between the column families required at each time step. This function is practically useful since, unlike in a relational database, query execution plans for NoSQL databases are closely linked to the physical schema. This means that 1) new query plans are necessary when the physical schema changes due to schema migration, and 2) developers must provide the specific steps for executing queries (i.e., a query plan), as NoSQL databases typically lack a query planner. Additionally, a migration plan is created from a *migration query* that defines how to collect records for a new column family from existing column families. The migration plan also includes any new columns or column families that must be created along with where the migrated data should be inserted. In addition, new column families are maintained incrementally and the workload continues executing during database migration.

We introduce query plan groups, a collection of query plans that are translated from each SQL query in the workload. We choose a single optimized query plan in a query plan group at every time step by schema optimization. We express a query plan as a sequence of steps where each step represents a `Get` operation, which fetches data from a column family. We call these steps `Get` steps for simplicity[2]. We also express a query plan group using a tree structure in which nodes with the same parent share a prefix of their query plans. We

[1] A new query appears if its frequency is changed from zero to non-zero.

[2] Query plans in prior work [23] not only have `Get` steps but may also have `Order` by steps that execute on the server side additionally with join/filtering/sort steps that are executed on the application side.

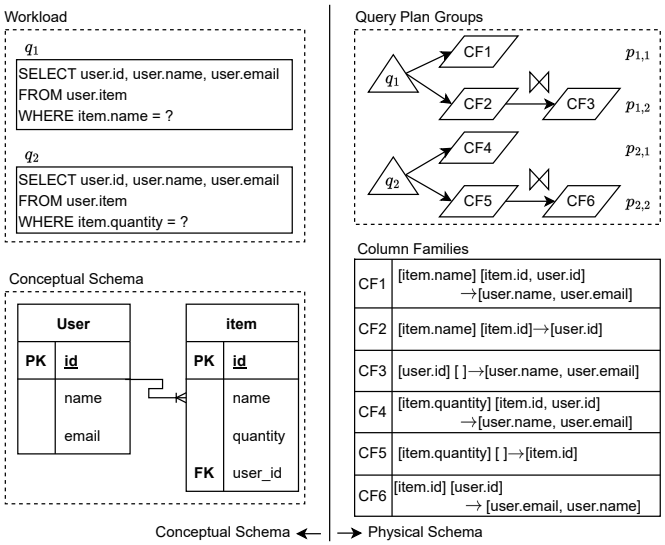


Figure 1: An example schema design obtained from a conceptual schema for two queries, q_1, q_2. A query plan group is generated using the column families that are enumerated from the entities used in the query.

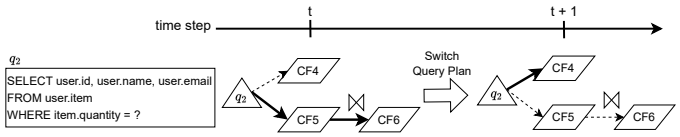


Figure 2: An example of changing query plans (the chosen plans are shown using solid arrows). The workload execution cost is reduced by changing the query plan according to the query frequency changes at each time step.

call a query plan with a single `Get` step a *materialized view (MV) plan* and one with multiple `Get` steps a *join plan*. MV plans are efficient for query processing since they do not need join operations between column families. In contrast, join plans use multiple normalized column families so they are efficient for update processing and saving storage size. However, they require expensive join operations between column families, since extensible record stores do not support join operations on the server side. Instead, join operations are executed at the client and may incur significant overhead for larger column families.

Next, we introduce migration plan groups, a collection of migration plans that are translated from a migration query. A migration plan generates a new column family for the next time step using the schema at the current time step. When there are several sets of column families that can be utilized in the migration plan, there is more than one migration plan in each migration plan group. We choose at most one optimized migration plan for each migration plan group by schema optimization.

2.4 Examples of optimizing time series schema

Figure 1 shows an example of column families obtained from a conceptual schema for two queries, q_1, q_2, in a workload. A query plan group is generated using the column families enumerated from the columns used in each query. $p_{1,1}$ and $p_{2,1}$ are MV plans, and $p_{1,2}$ and $p_{2,2}$ are join plans. For example, query plan $p_{1,2}$ first extracts records from *CF*2 using an equality predicate on $item.name$ in query q_1 and then extracts records from *CF*3 using $user.id$ as the join key between *CF*2 and *CF*3. Since join operations must be executed on the application side, the join plan $p_{1,2}$ is significantly slower than the MV plan $p_{1,1}$. However, the join plan $p_{1,2}$ requires less storage size than the MV plan $p_{1,1}$, since $p_{1,2}$ uses a normalized schema, *CF*2, *CF*3.

Next, Figure 2 shows an example of changing query plans for a time-dependent workload. For this example, we assume that 1) MV plans cannot be chosen for both q_1 and q_2 due to insufficient storage space, and 2) the frequency of q_1 is larger than the frequency of q_2 at time t and they are reversed at time $t + 1$; the frequency of q_1 becomes smaller than q_2's. In this case, a join plan is chosen for q_2 at time t and switched to an MV plan at time $t + 1$ due to a change in query frequency. Such changes in the query plan reduce the workload execution cost. However, they incur the additional cost of database migration. Therefore, we need to choose an optimized time series schema by considering the trade-off between workload execution cost and migration cost.

3 PROPOSED OPTIMIZATION FORMULA

We formulate the problem of time-dependent schema design using a single integer linear program (ILP) and output an optimized time series schema, query plans at every time step, and migration plans between adjacent time steps. The benefit of this approach is that it formulates the total cost of time-dependent workload execution and database migration using a single ILP, so it can handle the trade-off between the cost of time-dependent workload and database migration.

For a given time-dependent workload consisting of queries Q and update operations U, we obtain an optimized time series schema $(S_1, ..., S_T)$ from time step $t = 1$ to T by minimizing the following objective function using three constraints for query plans, migration plans, and storage size.

Objective function: The objective of generating an optimal time series physical schema $(S_1, \ldots, S_T)$ is to minimize the total cost of time-dependent workload execution and database migration.

$$\min_{S_1,\ldots,S_T} \sum_{t=1}^{T} workload(S_t) + \sum_{t=1}^{T-1} migrate(S_t, S_{t+1}) \qquad (1)$$

where $workload(S_t)$ indicates the workload execution cost on schema S_t at time step t, and $migrate(S_t, S_{t+1})$ indicates the migration cost from schema S_t to S_{t+1}. If there is no migration ($S_t = S_{t+1}$), then $migrate(S_t, S_{t+1}) = 0$.

3.1 Workload execution cost

Workload execution cost is defined as the total cost of all query and update operations in the workload; each query/update operation cost is computed as the product of its frequency and its estimated execution cost. In detail, we define the workload execution cost at

time step t as follows:

$$workload(S_t) = \sum_{q_i \in Q} \sum_{cf_j \in CF(P(q_i))} f_i(t) C_{ij} \delta_{ijt} + \sum_{u_u \in U} \sum_{cf_n \in S(t)} f_u(t) C'_{un} \delta_{nt} \quad (2)$$

where $P(q_i)$ is a query plan group enumerated from q_i and $CF(P(q_i))$ is a set of column families enumerated from q_i used in query plan group $P(q_i)$. Intuitively, column families are enumerated by materializing subqueries of q_i. We detail this process in Section 4. The first and second terms on the right-hand side express query cost and update operation cost, respectively. The first term, $f_i(t)$ is the frequency of query i at time step t, and C_{ij} is the coefficient that represents the cost of query i using the column family cf_j. δ_{ijt} is a binary decision variable that expresses whether query q_i uses column family cf_j at time step t. Thus, the query cost is the summation of $f_i(t)C_{ij}\delta_{ijt}$ for all combinations of queries and column families. The second term, $f_u(t)$, is the frequency of the update operation u at time step t and C'_{un} is the coefficient that represents the cost of the update operation u for the column family cf_n. δ_{nt} is a binary decision variable that expresses whether column family n exists in schema S_t at time step t. Thus, the update cost is the sum of $f_u(t)C'_{un}\delta_{nt}$ for all combinations of update operations and column families.

3.2 Migration cost

Remember that we choose a single optimized migration plan among multiple migration plan groups obtained from migration queries. We define the database migration cost from old schema S_t to new schema S_{t+1} using multiple migration queries as follows:

$$migrate(S_t, S_{t+1}) = \sum_{cf_g \in S_{t+1}} \sum_{cf_h \in CF(P(q'_o)), q'_o \in M(cf_g)} C^E_h \delta^E_{goht} + \sum_{cf_g \in S_{t+1}} C^L_g \delta^L_{g(t+1)} + \sum_{u_u \in U} \sum_{cf_g \in S_{t+1}} C^U_{ug} \delta^L_{g(t+1)} \quad (3)$$

where $M(cf_g)$ is migration queries for generating a target column family $cf_g \in S_{t+1}$, migration plan group $P(q'_o)$ and set of column families $CF(P(q'_o))$ enumerated from migration query q'_o are similarly defined in the workload execution cost (Equation (2)).

The first term on the right-hand side expresses the cost of collecting records from the old schema using migration plans. C^E_h is the coefficient that represents the cost of collecting data from each column family cf_h. δ^E_{goht} is a binary decision variable[3] that expresses whether migration query q'_o uses old column family cf_h for generating the target column family cf_g at time step t. Thus, the cost of collecting records from the old schema is the sum of $C^E_h \delta^E_{goht}$ for all combinations of column family cf_g, migration query q'_o, and column family cf_h.

The second term expresses the cost of inserting the collected records into a new schema. C^L_g is the coefficient that represents the cost of inserting the collected records into a new column family cf_g. $\delta^L_{g(t+1)}$ is a binary decision variable that expresses whether a database migration to column family cf_g is required between time steps t and $t+1$. If $\delta^L_{g(t+1)} = 1$ then column family cf_g does not exist at time step t and exists at $t+1$, so we introduce the following constraint (4):

$$\delta_{g(t+1)} - \delta_{gt} \leq \delta^L_{g(t+1)} \quad (4)$$

The third term expresses the cost of maintaining the new schema for ongoing update operations U in the workload. C^U_{ug} is a coefficient that represents the cost of an update operation $u \in U$ for a new column family cf_g during the migration process.

3.3 Constraints

We introduce three constraints for query plans, migration plans, and storage size to choose column families required for optimized query/migration plans and skip generating unused column families.

3.3.1 Constraints for query plans. Constraints for query plans ensure that for each query $q_i \in Q$ at each time step t, 1) we choose a single optimized query plan p_t from a query plan group $P(q_i)$ (constraint (5, 6)), and 2) all column families used in the optimized query plan p_t should exist in schema S_t (constraint (7)). A decision variable is appropriately assigned to each δ_{ijt} and δ_{nt} in Equation (2) using these constraints.

The first constraint (5) ensures that if the column family cf_j is used in the query plan p_t, any other column family cf_l that precedes ($\prec$) cf_j in the same query plan must be chosen.

$$\forall cf_j, cf_l \in CF(P(q_i)).\ cf_l \prec_{p_t} cf_j \rightarrow \delta_{ilt} \geq \delta_{ijt} \quad (5)$$

where δ_{ijt} expresses whether query q_i uses column family cf_j at time step t (introduced in Equation (2)).

The second constraint (6) ensures that we produce a single unique query plan that joins all adjacent entities in a query graph[4] [23].

$$\forall e, e' \in Entity(q_i). \sum_{\{cf_j \in CF(P(q_i)) \mid e, e' \in Entity(cf_j)\}} \delta_{ijt} = 1 \quad (6)$$

where $Entity(q_i)$ is an entity set used in query q_i, e and e' are entities that are adjacent in q_i's query graph, and $Entity(cf_j)$ is an entity set from which a column family cf_j is generated. The constraint (6) ensures that only a single column family cf_j is chosen from $CF(P(q_i))$ for partial columns of each adjacent entity pair e, e' used in q_i at every time step t (specified by $\sum \delta_{ijt} = 1$)[5].

The third constraint (7) ensures that if the query plan p_t is chosen as the optimized plan for q_i at time step t, all column families (cf_j) used in p_t should exist at the same time step.

$$\forall cf_j \in CF(P(q_i)).\ \delta_{jt} \geq \delta_{ijt} \quad (7)$$

That is, if $\delta_{ijt} = 1$ then $\delta_{jt} = 1$ (remember that δ_{jt} was introduced in Equation (2)).

[3] $\delta^E_{goht} = 0$ when cf_g is not newly generated at time step t for all cf_h.

[4] The query graph expresses a partial schema relating to a given query extracted from the entity graph. Our current implementation is restricted to acyclic query graphs as in the implementation of NoSE [23].

[5] We permit generating a single column family for a leaf entity in the query graph.

As an example, we give the following constraints for q_2 at time step t in Figure 2.

$$\delta_{2,5,t} \geq \delta_{2,6,t} \tag{8a}$$

$$\delta_{2,4,t} + \delta_{2,5,t} = 1, \delta_{2,6,t} + \delta_{2,4,t} = 1 \tag{8b}$$

$$\delta_{j,t} \geq \delta_{2,j,t} \quad \forall j \in \{4,5,6\} \tag{8c}$$

where (8a), (8b), and (8c) are instantiated from general constraints (5), (6), and (7), respectively. Constraint (8a) ensures that *CF*5 is chosen whenever *CF*6 is used. The left-hand side of Constraint (8b) specifies choosing either *CF*4 or *CF*5 for partial columns of an adjacent entity pair (*Item* and *User*), and the right-hand side specifies choosing either *CF*6 or *CF*4 for partial columns of a leaf entity (*User*[5]). Finally, the constraint (8c) ensures that S_t contains *CF*4, *CF*5, *CF*6 if they are used in the optimized query plan of q_2.

3.3.2 Constraints for migration plans. Constraints for migration plans ensure that for each target column family $cf_g \in S_{t+1}$ (specified by $\delta^L_{g(t+1)} = 1$), 1) we choose a single migration plan translated from the migration queries for generating the target column family cf_g (constraints (9),(10),(11)) and 2) all column families used in the chosen migration plan should exist in schema S_t (constraint (12)). A decision variable is appropriately assigned to each δ^E_{goht} and $\delta^L_{g(t+1)}$ in Equation (3) using these constraints.

Similar to the constraint (5) for query plans, the first constraint (9) ensures that if the previous column family cf_h is used in a migration plan, any other column family cf_l that precedes ($\prec$) cf_h in the same migration plan must be chosen.

$$\begin{aligned} &\forall q'_o \in M(cf_g),\ \forall cf_h, cf_l \in CF(P(q'_o)). \\ &\quad cf_l \prec cf_h \rightarrow \delta^E_{golt} \geq \delta^E_{goht} \end{aligned} \tag{9}$$

where δ^E_{golt} and δ^E_{goht} express whether old column family cf_l and cf_h exists in schema S_t, respectively.

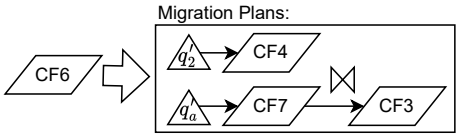
We choose a single migration plan from migration queries for the target column family cf_g in two steps as follows. In the first step, we choose a single migration query q'_o from the migration queries $M(cf_g)$ (specified by $\sum \delta^T_{got} = 1$) when the target column family cf_g is generated at time step $t+1$ (specified by $\delta^L_{g(t+1)} = 1$) using the second constraint (10).

$$\sum_{q'_o \in M(cf_g)} \delta^T_{got} = \delta^L_{g(t+1)} \tag{10}$$

In the second step, we choose a single migration plan for the migration query chosen in the first step. Similarly to the constraint (6) for query plans, the third constraint (11) ensures that only a single column family is chosen (specified by $\sum \delta^E_{gomt} = \delta^T_{got}$) for partial columns of adjacent entity pair e, e' in the migration query graph from each migration plan group transformed from q'_o at every time step t.

$$\begin{aligned} &\forall e, e' \in Entity(cf_g). \\ &\sum_{\{cf_m \in CF(M(cf_g)) \mid e,e' \in Entity(cf_m)\}} \delta^E_{gomt} = \delta^T_{got} \end{aligned} \tag{11}$$

where δ^T_{got} is a binary decision variable that expresses whether an optimized migration plan for the target column family cf_g is chosen from migration plan group P^M_o at time step t.



Migration Queries:

q'_2	SELECT user.name, user.email, user.id, item.id FROM user.item WHERE item.quantity = ?
q'_a	SELECT user.name, user.email, user.id FROM user.item WHERE item.id = ?

CF enumerated for the workload:

CF7	[item.id] [ ] → [user.id]

Figure 3: An example of two migration plans that generate a new column family *CF*6 given in Figure 1. In order to collect data for *CF*6, we generate migration queries q'_2 and q'_a and enumerate their migration plans using our proposed method described in Section 4.5.

Similarly to the constraint (7), the fourth constraint (12) ensures that if a migration plan is chosen as the optimized plan for generating cf_g at time step t, all column families (cf_h) used in the optimized migration plan should exist at the same time step.

$$\forall q'_o \in M(cf_g), \forall cf_h \in CF(P(q'_o)).\ \delta_{ht} \geq \delta^E_{goh(t+1)} \tag{12}$$

Figure 3 shows an example of a migration plan that generates a new column family *CF*6 in Figure 2. The constraints to generate a single migration plan for generating *CF*6 are described below.

$$\delta^E_{6,a,7,t} \geq \delta^E_{6,a,3,t} \tag{13a}$$

$$\delta^T_{6,a,t} + \delta^T_{6,2,t} = \delta^L_{6,t} \tag{13b}$$

$$\delta^E_{6,a,7,t} = \delta^T_{6,a,t},\ \delta^E_{6,a,3,t} = \delta^T_{6,a,t},\ \delta^E_{6,2,4,t} = \delta^T_{6,2,t} \tag{13c}$$

$$\delta_{3,t} \geq \delta^E_{6,a,3,t},\ \delta_{4,t} \geq \delta^E_{6,2,4,t},\ \delta_{7,t} \geq \delta^E_{6,2,7,t} \tag{13d}$$

where (13a), (13b), (13c), (13d), are instantiated from general constraints (9), (10), (11), (12), respectively. Constraint (13a) ensures that if *CF*3 is used in a migration belonging to migration plan group a, then the preceding column family, *CF*7, is also chosen for the schema at time stamp t. Constraint (13b) ensures that at most one migration query is chosen among migration queries (q'_a or q'_2). Constraint (13c) ensures that at most one migration plan is chosen for q'_a and q'_2, respectively. (Note that a migration query and its associated plans will not be required if a migration does not happen for a given time step.) Finally, constraint (13d) ensures that the schema recommended at time step t (S_t) contains all column families that are used in the optimized migration plan.

3.3.3 Constraint for storage size. The storage size constraint (14) ensures that the storage size of all column families should be less than a constant B at each time step t.

$$\forall t \in [1,T].\ \sum_j size(cf_j)\delta_{jt} \leq B \tag{14}$$

where $size(cf)$ is the storage size of column family cf. We ignore the size change of column families even when workload contains update operations, since the change in size is usually quite small compared to the whole database size.

4 PROPOSED SYSTEM

The optimization problem for time series schema described in Section 3 does not scale due to the large number of decision variables (δ_{jt} and δ^{E}_{goht}). For example, the number of variables δ_{jt} is the product of the number of column families and the number of time steps. This is caused by the fact that the number of time series schema candidates increases significantly depending on the number of time steps, and the number of migration plan candidates also increases depending on the number of queries and schema candidates. To overcome these obstacles, we propose a system for optimizing time series schema design by effectively reducing the number of schema (column family) candidates as well as the number of migration plan candidates.

4.1 System Overview

Figure 4 shows an overview of our system. Our system identifies an optimized time series schema, optimized query plans, and migration plans using the following procedures: column family enumeration (Section 4.2), column family pruning (Section 4.3), query plan enumeration (Section 4.4), migration plan enumeration (Section 4.5), cost estimation (Section 4.6), and optimization (Section 4.7).

4.2 Column Family Enumeration

The purpose of this step is to enumerate column family candidates used for the optimization problem of time series schema (Section 3). Since arbitrary query plans can be chosen as optimized plans at any time step, we enumerate all possible column family candidates that can be used for query plans. To this end, we take the same approach used in existing systems for column family enumeration [23]; we decompose each query and materialize the whole query and its subqueries as column families. Thus, we can answer a query with a single column family (query efficient MV plan) or multiple column families by joining them (update efficient join plan). In detail, we employ a query graph introduced in Section 3.3 which is a sub-graph of the entity graph in a conceptual schema and expresses a partial schema referred from a given query. We enumerate column family candidates by recursively decomposing the query graph at each node into two subqueries and materialize them as column families. In addition, in order to increase the utilization of column families for answering queries, we employ *relaxed queries* [23] that are transformed from the original queries in the workload by moving attributes used in `WHERE/ORDER BY` clauses to the `SELECT` clause. We keep at least a single equality predicate in `WHERE` clause in the same way as NoSE to construct a valid `Get` request for the column family. This is common for queries in extensible record stores since table scans are often prohibitively expensive. The column families materialized from relaxed queries can be used to answer more queries, because they can be used for queries with fewer conditions in `WHERE/ORDER BY` clauses. This is because the materialized view for a query defines a partition key for the corresponding column family, which in turn indicates required columns that must be specified to execute a query (i.e. partition key values are required to execute a query). However, the number of column families generated from relaxed queries increases exponentially with the number of edges in the query graph since column families are materialized from subqueries recursively decomposed at every edge. Moreover, it increases in relation to the factorial of the number of attributes used in `WHERE/ORDER BY` clauses, because column family variants are sensitive to attribute order.

To overcome such a significant growth in the number of enumerated column families, we propose two pruning techniques. First, we restrict the number of recursive decompositions of query graphs to one single decomposition. That is, we decompose a query graph into at most two subqueries. The total number of queries (including the original) after decomposition becomes $2k + 1$, which is linear in the number of edges (k) in the original query graph. Second, we reduce the variants of clustering keys in column families by leveraging the features of extensible record stores as follows. Any prefix of the clustering key should contain attributes used in `GROUP BY/ORDER BY` clauses so that `GROUP BY/ORDER BY` operations can be executed on the server side. Notice that we can safely ignore the order of the remainder of a clustering key if the columns appear only in the `SELECT` clause, because those columns do not need to be sorted. Thus, we can reduce the number of column families by treating the remainder of the clustering key to be order-insensitive.

4.3 Column family pruning using workload summary tree

We have observed that the number of schema candidates increases significantly depending on the number of time steps, leading to a substantial increase in the optimization time for time series schema design using an ILP solver. To address this issue, we propose a novel column family pruning technique by introducing a novel hierarchical data structure called a workload summary tree. The workload summary tree approximates the ILP of the original workload as a multi-level workload summary. Our technique effectively creates ILP constraints that prune unpromising column families by leveraging the multi-level workload summary.

4.3.1 Workload summary tree. We design the workload summary tree to have the following features: 1) every child node represents a sub-workload split over time from the workload of its parent node, and 2) every parent node's workload is summarized from its children's sub-workload. Therefore, the root node represents a compact summary of the entire workload with a small number of time steps and each leaf node represents an unsummarized sub-workload split. Each intermediate node represents a summarized sub-workload split. In detail, we design the workload summary tree as a binary tree[6]. Each node manages a sub-workload with three time steps (minimum, median, maximum). The workload managed at each parent node represents a summary of the workloads of its child nodes: the left child manages the left half of the parent workload (between the minimum and median time steps) and the right child manages the right half (between median and maximum time steps).

4.3.2 Creating constraints for column family pruning. By leveraging the workload summary tree, our pruning technique effectively finds promising column families from enumerated column families in Section 4.2 using a multi-level workload summary: the upper levels capture global aspects and lower levels capture more local aspects of the workload. Moreover, in order to take global aspects from upper levels into account at lower levels, we propose a novel algorithm that

[6]To make the discussion simple, we assume the time step size in the workload is 2^n. If not, we can add additional leaf nodes for the remaining time steps.

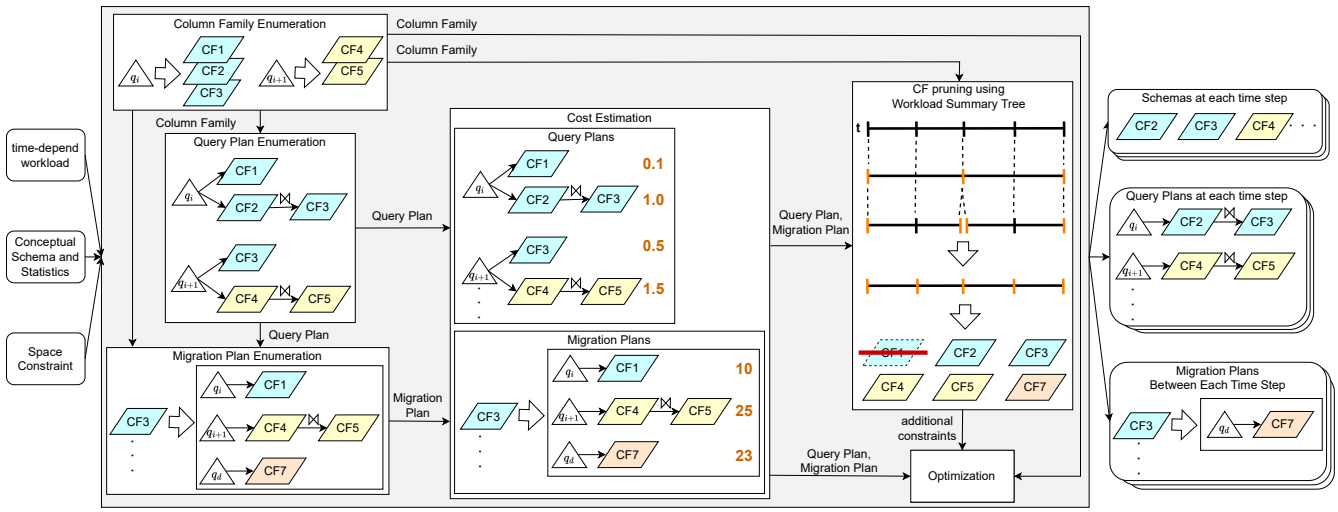


Figure 4: An overview of the proposed system.

recursively constructs an ILP for the summarized workload assigned to each node starting from the root and translates its answer to the ILP constraints of its child sub-workloads. In detail, a constraint enforces that the ILP of child workloads inherits the optimized column families found in the parent workload: a child node solves a local ILP so that the optimized column families at the min/max time steps are identical to the ones found at the same time steps (e.g. min and median for the left child node) in the parent workload. After computing the portion of the ILP for each node throughout the tree, we can effectively find promising column families that were chosen as ILP solutions at least one node in the tree. Our pruning technique finds unpromising column families that are enumerated in Section 4.2 but not included in promising column families given by the workload summary tree. And then, our pruning technique creates ILP constraints that prune these unpromising column families to be used in the optimization in Section 4.7. This approach is efficient because the time step size of the (sub-)workload at every node is limited to only three (minimum/median/maximum), which is significantly smaller than that of the original workload.

The detailed algorithm is given in Algorithm 1. The input parameters, query plans and migration plans, are initialized to the outputs of the steps described in Section 4.4 (Query Plan Enumeration) and Section 4.5 (Migration Plan Enumeration), respectively. Also, the ILP constraints are initialized to an empty set. *solve_ILP* function (line 3) optimizes ILP for each node in the workload summary tree. The column families that are used by either the chosen query plans or migration plans in the ILP solution S are obtained (line 4). *translate_to_constraint* function (line 6) translates the column families chosen in the ILP solution for a current node into the constraint for its child node in order to share the optimized column families found at *min_ts*, *mid_ts*, *max_ts* time steps. *get_child_workload* function (line 7, 9) constructs a sub-workload for a child node for given min/max time steps. We recursively invoke the *get_subtree_cfs* function (line 8, 10) by splitting the current workload into two child sub-workloads. After the recursion (line 8, 10), the identified promising column families are returned.

Algorithm 1: get promising column families

Input : w (workload), cfs (column families), qps (query plans), mps (migration plans), $csrt$ (ILP constraints)
Output: C: promising CF candidates

1 **Function** *get_subtree_cfs(w, cfs, qps, mps, csrt)* **is**
2 **if** $w.min\_ts + 1$ *is* $w.max\_ts$ **then return** $\emptyset$
  // get solution of current workload
3 $S \leftarrow solve\_ILP(w, cfs, qps, mps, csrt)$
  // get CFs used by any query or migration plans in the solution
4 $C \leftarrow S.cfs$
  // get the middle time step
5 $mid\_ts \leftarrow (w.min\_ts + w.max\_ts)/2$
  // translate chosen CFs in the solution to a constraint
6 $csrt \leftarrow csrt \bigcup translate\_to\_constraint(S.cfs)$
7 $l\_child\_w \leftarrow$ $get\_child\_workload(w, w.min\_ts, mid\_ts)$
8 $C \leftarrow C \bigcup get\_subtree\_cfs(l\_child\_w, qps, mps, csrt)$
9 $r\_child\_w \leftarrow$ $get\_child\_workload(w, mid\_ts, w.max\_ts)$
10 $C \leftarrow C \bigcup get\_subtree\_cfs(r\_child\_w, qps, mps, csrt)$
11 **return** C
12 **end**

Algorithm 2 constructs constraints to eliminate unpromising column families utilized for schema optimization, as detailed in Section 3. In line 3, Algorithm 2 invokes the *get_subtree_cfs* function to retrieve promising column families. In line 5, we invoke *gen_cstr_to_exclude* function for each column family enumerated in Section 4.2 (Column Family Enumeration) that is not selected in *get_subtree_cfs* function. The constructed constraints are represented by Equation (15):

$$\forall t \in [1, T], \forall cf_k \in cfs \setminus C.\ \delta_{kt} = 0 \tag{15}$$

and it is integrated in the schema optimization in Section 3, in order to significantly reduce the optimization cost.

Algorithm 2: Create Additional Constraints

Input : w (workload), cfs (column families), qps (query plans), mps (migration plans)
Output: $CSTR$: constraints to exclude unpromising column families

```
1 Function create_additional_cstrs(w, cfs, qps, mps) is
2     CSTR ← ∅
      // get promising CFs using subtree
3     C ← get_subtree_cfs(w, cfs, qps, mps, ∅)
4     for c ∈ cfs − C do
          // generate constraint that excludes the CF
5         CSTR ← CSTR ⋃ gen_cstr_to_exclude(c)
6     end for
7     return CSTR
8 end
```

4.4 Query Plan Enumeration

This step transforms each query in the workload into query plans. We take the same approach [23] as follows. We enumerate query plans that join column family candidates enumerated in the column family pruning step (Section 4.3) from all queries. Here, each query plan corresponds to a reconstruction of the original query graph from its decomposed subqueries. The enumerated query plans consist of three types of operations, 1) `Get`/`Put` operations for column families, 2) `ORDER BY`/`GROUP BY` at the server side, and 3) join/selection/ `ORDER BY`/`GROUP BY` at the application side.

4.5 Migration Plan Enumeration

The purpose of this step is to enumerate the candidates of the migration plan used for the time series schema optimization problem (Section 3). That is, we enumerate the migration plans from schema S_t to S_{t+1} at any time step t. However, the column families in S_t are not decided before schema optimization, so we must enumerate the migration plan candidates that generate each target column family enumerated by the column family enumeration (Section 4.2). We introduce two techniques that effectively reduce the number of migration plan candidates. The first uses query plans as migration plans based on the fact that workload changes cause optimized query plan changes, which necessitates a database migration. The second enumerates additional migration plans using a migration query based on the target column family to complement the first.

4.5.1 Migration plan enumeration reusing query plans. Optimized query plan changes require generating the target column families in S_{t+1} that are used by optimized query plans at time step $t+1$. We observe that if the optimized query plans at time steps t and $t+1$ are produced from the same query, they use column families produced from the same relational entities, so the former plan outputs similar target column families in S_{t+1} as those of the latter plan. Based on this observation, we can utilize query plans from time step t as migration plans from S_t to S_{t+1}. However, since optimized query plans are not decided before schema optimization, we treat all enumerated query plans at time step t as migration plan candidates to allow those query plans to become optimized query plans.

In detail, we identify queries from a set of workload queries Q using each target column family in the enumerated column families (Section 4.2) that we call *source queries*. Then, we rewrite the source queries to migration queries by making the following modifications: 1) we simplify query plans by removing aggregation/`ORDER BY` operations that are not necessary for migration plans, and 2) we adjust the projections of the query plans to include the required columns for the target column family. Finally, we enumerate plans for the migration queries and treat them as migration plans.

The first migration plan in Figure 3 is an example of migration plan enumeration reusing query plans. This migration plan generates a new column family *CF*6 (the target column family). Since *CF*6 is used in the q_2 query plan group, we choose q_2 as the source query for *CF*6. Then, we modify q_2 to migration query q_2' (described in Figure 3) as follows: q_2' shares the same `FROM`/`WHERE` clauses as q_2 but with an extended `SELECT` clause to cover necessary columns for generating *CF*6. Finally, we generate migration plans from q_2'. We obtain $p_{2,1}$ as a migration plan that uses *CF*4.

4.5.2 Complementary migration plan enumeration. The above enumeration technique may not enumerate appropriate migration plans when the number of query plans in each query is small or the query plans do not have some attributes of the target column family. For example, if *CF*4 does not have some attributes of *CF*6, there is no migration plan for *CF*6 with the above technique. To complement this, the second technique enumerates additional migration plans using a simple migration query, which specifies the partition key and columns of the target column family in the `WHERE` clause and `SELECT` clause, respectively.

The second migration plan in Figure 3 is an example of the enumeration of complementary migration plans. q_a' is a simple migration query generated from the target column family,*CF*6 (*item.id* is the partition key and *user.name, user.email, user.name* are the remaining columns as shown in Figure 1). We enumerate its migration plan reusing *CF*7 and *CF*3, which are generated from other queries, such as q_1.

4.6 Cost Estimation

In order to optimize the objective of Equation 1, we need to estimate the coefficients used to calculate the execution cost of the workload (Equation 2) and the migration cost (Equation 3).

Remember that C_{ij} in Equation 2 is defined as the coefficient that represents the cost of processing query i using the column family cf_j. We estimate this cost using linear regression with the function $T(n, w, s)$[7] where n is the number of `Get` operations, w is the query cardinality (the expected number of records in the result), and s is the record size of column family j. n is computed as the sum of `Get` operations in the query plan tree: a single `Get` operation is required for the first step in the tree, and we use the query-cardinality of this node as the required number of `Get` operations for later steps, because we need to invoke a `Get` operation for each record returned from this node. w is computed using the cardinality of attributes used in equality conditions and using the number of records of each entity in the conceptual schema. When query i uses a `GROUP BY` clause, w is computed using the number of expected groups by

[7] We choose the simplest approach of using linear regression. We can utilize more advanced machine learning methods to achieve higher accuracy.

pushing down the GROUP BY clause at the server side. Since the linear regression function $T(n, w, s)$ depends on the instance of extensible record stores and its computing environment, we train $T(n, w, s)$ using performance profiles as training datasets which we collected by changing queries (attributes used in SELECT/WHERE clauses) and the number of records in column families. Next, C'_{un} in Equation 2 is defined as the coefficient that represents the cost of update operation u for the column family cf_n. We estimate by applying linear regression to the function $T'(w)$ where w is the number of expected updated records. Similarly to the above C_{ij} estimation, we train $T'(w)$ using performance profiles.

Regarding the migration cost, we need to estimate the coefficients C^E, C^L, C^U used in Equation 3. First, C^E_h is the coefficient that represents the cost of data collection from each column family cf_h, which depends on the size of cf_h. So, we estimate C^E_h using linear regression with the function $T''(s_h)$ where s_h is the size of cf_h. Second, C^L_g is the coefficient that represents the cost of inserting the collected records into the new column family cf_g, which depends on the size of s_g. So, we estimate C^L_g using linear regression with the function $T'''(s_g)$ where s_g is the size of cf_g. Finally, C^U_{ug} is the coefficient that represents the cost of the update operation $u \in U$ for the new column family cf_g during the migration process. We estimate it using Equation (16):

$$C^U_{ug} = f_{u(t-1)} C'_{ug} \frac{C^L_g}{(interval)} \tag{16}$$

where $C^L_g/(interval)$ is the time ratio of column family g generation to the interval between time steps, and $f_u(t-1)$ is the frequency of update operation u at time step $t-1$.

Similarly to the coefficient estimation for workload cost, we train the above regression models using performance profiles from collecting and inserting records as the training datasets: the profiles are collected by changing the number of records in column families.

4.7 Optimization by ILP solver

Finally, we obtain the optimal design for the time series schema by minimizing the total cost of execution of the time-dependent workload and database migrations for the enumerated column families, query plans, and migration plans. In addition, we additionally minimize the number of column families and their storage size in three steps, as follows.

First, we identify the optimal design for the time series schema using Equation (1) and keep its minimum cost. Second, we additionally minimize the number of column families using (17) while keeping the cost of Equation (1) as the minimum cost.

$$\min \sum_{t=1}^{T} \sum_{j} \delta_{jt} \tag{17}$$

Finally, we also minimize the size of column families using (18) while keeping the number of column families as the minimum.

$$\min \sum_{t=1}^{T} \sum_{j} s_j \delta_{jt} \tag{18}$$

This approach is especially effective when the workload does not have update operations, because the approach ensures the minimality of the number of column families and their size.

5 EXPERIMENTAL STUDY

We conducted experiments to address the following questions. **Q1.** *How effective is our proposal in improving workload latency?* We evaluate the advantages of our proposal by comparing the workload latency to that of NoSE, which optimizes schema design using a static workload approach, such as averaging query frequencies in a time-dependent workload (Section 5.2). **Q2.** *How does the workload latency change when changing the amount of available storage (Section 5.3)?* **Q3.** *How effective is our schema candidate pruning by leveraging workload summary trees?* We evaluate how much the optimization time is reduced (Section 5.4).

All experiments were performed on a server with the following specifications: CPU: Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz (1200 MHz) with 64 logical cores, memory: 1.5TB, operating system: Ubuntu 22.04.2 LTS working with Docker 20.10.12. We deployed Apache Cassandra with five nodes on a single server using Docker containers and measured the workload latency while running a client and database migrator on the same server. We used Gurobi version 10.0.1[8] as the ILP solver.

5.1 Experimental Setting

Workload: We introduce typical query frequency patterns in the prior studies (workload forecasting [22] and forecasting time series [14]):

- **Cycles:** Many applications exhibit cyclic workload patterns because they are designed to interact with humans. For example, OLTP queries tend to be more frequent during the daytime and decrease at night.
- **Evolution and Stagnation[9]:** The second workload pattern involves the evolution or stagnation of database workloads over time. A trend is observed when there is a long-term increase or decrease in the workload.
- **Growth and Spikes:** The last workload pattern involves a sudden increase in query volume during specific time periods. This growth pattern typically occurs after the launch of new services or leading up to deadlines.

Based on the above query frequency patterns, we generate synthetic time-dependent workloads by modifying a widely-used static benchmark, TPC-H. We generated the TPC-H database with a scale factor of 1 for two reasons: 1) the schema optimization time depends on the number of queries and time steps but not on the database size, and 2) the benefits of schema optimization are primarily influenced by query frequency, and less by the database size. Larger databases incur higher migration costs but offer greater advantages for join operations by denormalization. We replicate the queries in the benchmark to have two query groups (the original group and the duplicate group) and assign their time-dependent frequencies as shown in Fig. 5. For example, the time-dependent workload in the top part shows the Cycles patterns of two groups occurring, for example, one in the US and the other in Japan, respectively. Additionally, queries in the duplicate group are referred to as Qn-dup (where $n = 1, \ldots, 22$).

Baseline methods: We compare workload performance using two optimized time series schemas using our proposal (with/without

[8] https://www.gurobi.com/

[9] These are called *trends* in time series forecasting [14].

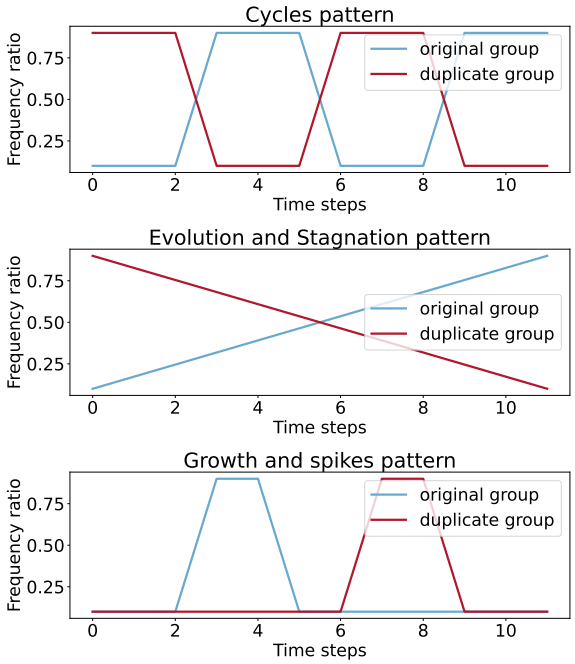


Figure 5: Query frequency ratio (Y-axis) changes of the two query groups over time steps (X-axis) in the three query frequency patterns.

column family pruning[10]) and using three static schemas obtained through NoSE [23] which we call static optimization: the first static schema (average freq.) is obtained using the average query frequencies, and the other two static schemas (first/last time step freq.) are obtained from the query frequencies at the first time step and the last time step, respectively. We also measure workload performance using the time series schema, obtained by optimizing the schema at every time step, without considering migration costs (ideal optimization).

Metrics and parameters: Each query is executed the same number of times at all time steps. Subsequently, we calculate the frequency-weighted average latency by multiplying the actual query latency by the query frequency.

Since TPC-H does not involve update operations, we employ storage size constraints to assess how the schema adapts to a specified storage size limit[11]. We change the storage size as follows. First, we determine the upper limit of storage size by performing static schema optimization using the average query frequencies without specifying a storage size limit. We then use the resulting schema size as the upper limit (100%). Subsequently, we reduce the storage size to below the upper limit.

5.2 Q1. Workload latency

The purpose of Q1 is to verify that our proposal effectively improves workload latency by adaptively making database migrations in response to the workload changes. We used the three typical query frequency patterns described in Section 5.1. Note: database migrations are performed in the background during the experiments, and as such, the workload latency includes the cost of database migrations. For this experiment, we set the storage constraint at 80% of the upper limit.

Figure 6a shows the results of workload latency for the **Cycles** pattern. Compared to the three static optimizations (first/last time step and average frequency), our proposal stably achieves higher performance across all time steps by adaptively changing the schema. The reduction is between 31.8% and 37.7% in cumulative latency across all time steps, indicating its superior performance by adaptively performing database migrations in response to workload changes. That is, the proposed method adaptively changes the schema so that frequent queries are executed in efficient materialized view plans, effectively reducing the frequency-weighted average latency. In contrast, the performance of the three static optimizations fluctuates, sometimes improving, and sometimes declining due to workload changes. We also observe that our proposal is only slightly slower (5.41%) than the ideal optimization in the cumulative latency. These results indicate the superior performance of our proposal in optimizing dynamic time series schema.

We further investigate how query plans change in response to workload changes. Table 1 shows how the query plan changes and at what time steps. For example, join plans are used for Q3, Q5, Q7, Q9 at time step 2 since their frequency is low (see the query frequency changes in the Cycles pattern in Figure 5). Once we move to time step 3, their frequency becomes high, so the schema is optimized so that materialized view plans are used for those queries. In contrast, the queries in the duplicate group (Q3-dup, Q5-dup, Q7-dup, Q9-dup) take opposite plan changes to those in the original group. Indeed, the estimated cost of Q3 is high, 69.62, during low-frequency time steps (0-2, 6-8) and low, 10.46, during high-frequency time steps (3-5, 9-11) in our system.

time	plan change	query
$2 \rightarrow 3$	MV Plan → Join Plan	Q3-dup, Q5-dup, Q7-dup, Q9-dup
	Join Plan → MV Plan	Q3, Q5, Q7, Q9
$5 \rightarrow 6$	MV Plan → Join Plan	Q3, Q5, Q7, Q9
	Join Plan → MV Plan	Q3-dup, Q5-dup, Q7-dup, Q9-dup
$8 \rightarrow 9$	MV Plan → Join Plan	Q3-dup, Q5-dup, Q7-dup, Q9-dup
	Join Plan → MV Plan	Q3, Q5, Q7, Q9

Table 1: How query plan changes at what time steps for the cyclic workload pattern under 80% storage constraint.

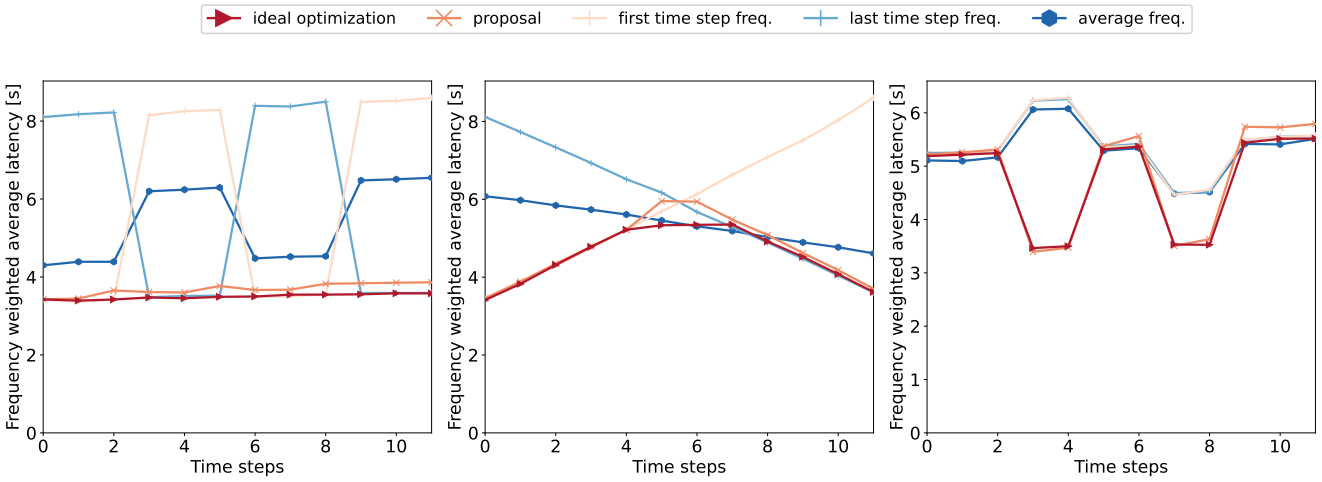
We also conduct the experiments for the other two patterns, **growth and spikes** and **evolution and stagnation**. Their results are shown in Fig. 6b and 6c, respectively. These results demonstrate similar performance characteristics to the **cycles** pattern. That is, our proposal stably achieves higher performance across all time steps by adaptively changing the schema. See Appendix .1 for more details.

5.3 Q2. Changing the size of storage constraint

We evaluate the effectiveness of our proposal by changing the size of the storage constraint. Figure 7 illustrates the results for the Cycles pattern by changing the size of the storage constraint from 100% to

[10] Our proposal without column family pruning does not obtain additional constraints generated in Section 4.3.

[11] Note that storage size constraints have a similar effect to update operations, because a smaller storage size reduces the cost of update operations.



(a) Cycles (b) Evolution and stagnation (c) Growth and spikes

Figure 6: Frequency weighted average latency (Y-axis) over time steps (X-axis) for three different workload patterns.

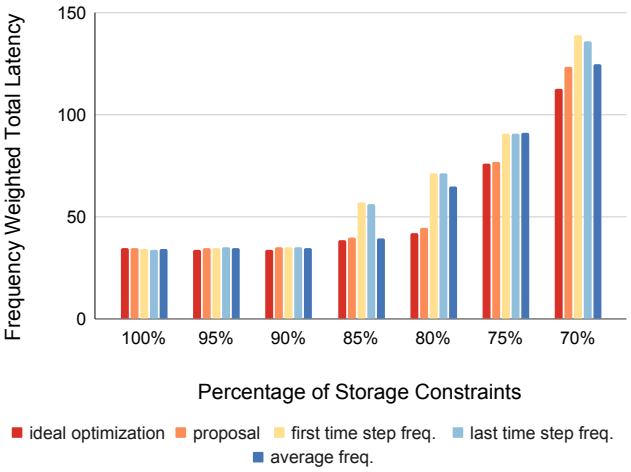


Figure 7: Frequency weighted total latency (Y-axis) by changing the size of storage constraint from 100% to 70% (X-axis).

70% in increments of 5%. We observe that our proposal always performs faster than the static optimizations. In particular, as the storage constraint size becomes smaller, our proposal can adaptively choose an optimized schema, so it performs better the static optimizations. Also, our proposal is almost comparable to the ideal optimization except for the storage constraint of 70%. In general, when the storage constraint becomes tighter, the schema tends to be normalized for all methods, which makes the workload performance slower caused by expensive join plans.

5.4 Q3. Effectiveness of schema pruning

The purpose of Q3 is to verify that our schema candidate pruning effectively improves the schema optimization time by leveraging workload summary trees. Since the number of time series schema candidates increases significantly depending on the number of time steps, we evaluate the optimization time by changing the number of time steps, starting from 4 to 44 with increments of 4 time steps. This allowed us to study a wide range of workload sizes while ensuring that the schema optimization time remained manageable. We report "did not finish" (DNF) when the ILP solver did not output the answer within 24 hours.

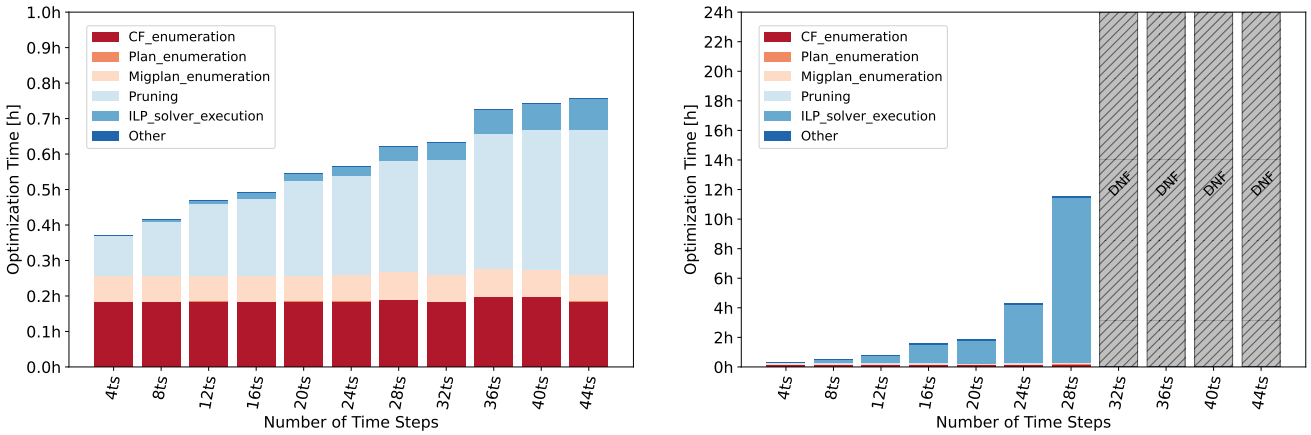
Figure 8 illustrates the results for the Cycles pattern by changing the number of time steps, confirming the efficiency of the schema candidate pruning. We observe that the optimization time using schema candidate pruning (Figure 8a) is significantly faster than without it (Figure 8b). The former is linear in the number of time steps. The latter is almost quadratic in the number of time steps. Also, when the number of time steps exceeds 32, the optimization without using pruning does not finish within 24 hours, making it impractical for real-world scenarios. In contrast, schema candidate pruning consistently keeps the optimization time under one hour, demonstrating its high scalability for the number of time steps.

Next, we consider a detailed breakdown of the optimization time. When using schema candidate pruning, the execution time of the ILP solver (`ILP_solver_execution`) is relatively small, and rather the schema candidate pruning (`Pruning`) occupies a large portion of the optimization time. In contrast, without using schema candidate pruning, the execution time of the ILP solver dominates almost all of the optimization time. This result confirms that our pruning technique effectively reduces the execution time of the ILP solver by significantly reducing the number of column families, since the ILP solver is not scalable to the size of decision variables.

6 RELATED WORK

We provide a summary of research trends in schema design for query workloads in NoSQL databases and relational databases. Due to their distinct characteristics, we categorize existing methods tailored for NoSQL and relational databases. Furthermore, we refer to the related concept of schema evolution management.

Schema design on NoSQL databases. There are also studies on database migration in NoSQL databases. NoSE [23] was proposed as a NoSQL schema design method for static workloads, which estimates the execution cost of executing the workload and optimizes the schema design. However, since it does not support time-dependent



(a) Using column family pruning **(b) Without column family pruning**

Figure 8: Schema optimization time (Y-axis) of our proposal by changing the number of time steps (X-axis). DNF indicates that the ILP solver does not output an answer within 24 hours. `CF_enumeration`**, and** `Plan_enumeration`**, and** `Migplan_enumeration` **indicates the enumeration of column families, query plans, and migration plans, respectively. Also,** `Pruning` **indicates column family pruning,** `ILP_solver_execution` **indicates ILP solver execution, and** `Other` **indicates other processing that is not classified into the aforementioned components.**

workloads, even if the schema is optimized once, performance may deteriorate due to workload changes.

There are studies on database migration for time-dependent workloads [2, 5, 13, 24]. Hillenbrand et al. [13] introduce a method that enumerates multiple database migration patterns based on an input database migration. It employs a rule-based approach to select the optimized database migration method. Notably, this method does not take into account the cost of executing the workload. CONST [24] is an example of a system that heuristically changes the schema design over time in response to changes in workload. However, it ignores the cost of database migration when schema changes so it does not generate optimal schema designs for time-dependent workloads.

Benats et al˙ [5] propose a technique for schema evolution recommendation for polystores which recommends migration between different database management systems (such as from Redis to MongoDB). Tables are then denormalized within a single database management system. However, it does not compute the total cost of executing the workload and database migrations, so users have to choose a suitable schema design.

Google Napa [2] guarantees robust query performance. Clients expect low query latency and low variance in latency regardless of the query/data ingestion load. It also provides users the flexibility to manually tune the system and meet their goals based on data freshness, resource costs, and query performance. In contrast, our approach introduces novelty in automatically optimizing the trade-off between the cost of time-dependent workload execution and database migration.

Schema design on relational databases. Workload-based schema design methods were proposed by several authors [16–18, 20, 25]. BIGSUBS [18] is a schema optimization technique for static workloads.

CloudViews [17] is a computation reuse framework that generates materialized views online. It maximizes the computation reuse by computing view utilities based on compile-time/run-time statistics. However, it does not consider both the view maintenance cost and storage size constraint, so it heuristically expires the materialized views. Pavlo et al. [25] proposed Peloton, a self-driving database framework for in-memory databases. Peloton introduces the receding-horizon control model (RHCM) to predict workload changes and also proposes a method that adaptively changes the schema according to workload changes on demand. This may lead to a local optimum since it does not consider the entire time-dependent workload. Kossmann et al. [20] proposed a dynamic optimization method for self-managing database systems. Their method uses an ILP to determine an efficient order to tune multiple dependent features, such as index selection, compression schemes, and data placement. However, they do not consider the view selection problem corresponding to our physical schema optimization problem.

In addition, there are other works [33, 34] that dynamically change various tuning parameters in database systems. In Wiese et al. [34], the database administrator registers a tuning procedure and the procedure is automatically triggered when a given condition is met, such as when deadlocks occur more frequently than a given threshold. Aken et al. [33] proposed OtterTune, which automatically tunes memory size, cache size, and other parameters by leveraging past experiences: it combines supervised and unsupervised learning methods to choose the most impactful tuning knobs.

Schema evolution management. Schema evolution management is one of the most challenging problems in recent data management. Significant research has been conducted in the area of schema evolution management [12, 28, 29] and its benchmarking [8]. Schema evolution is distinct from the schema optimization problem since the former deals with logical schema evolution, while the latter optimizes the physical schema for a given logical schema and queries. MigCast [12] simulates workload cost (latency) and database migration cost (price) for multiple data migration strategies, such as eager, lazy, and incremental. However, it does not optimize the schema design considering the trade-off between workload latency and database migration cost.

7 CONCLUSION

We proposed new techniques for optimizing schemas for time-dependent workloads by effectively reducing the number of schema candidates and migration plan candidates. First, we formulated the optimization problem of time series schema design with a single integer linear program to minimize the total cost of time-dependent workload execution and database migration. Second, we proposed a novel schema candidate pruning technique by decomposing the ILP of the original time-dependent workload via approximation into a hierarchy of local ILPs for smaller sub-workloads. This technique is scalable by effectively pruning uninteresting schema candidates. Finally, we proposed an effective technique that reduces the number of migration plan candidates by restricting the candidates according to how the optimized query plans are changed. The evaluation confirmed that our proposal achieves consistently higher performance than the existing static schema optimization method.

ACKNOWLEDGMENT

This paper is based on results obtained from a project, JPNP16007, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

REFERENCES

[1] Hyper suprime-cam Subaru strategic program. https://hsc-release.mtk.nao.ac.jp/doc/index.php/database-2, 2021.
[2] A. Agiwal et al. Napa: Powering scalable data warehousing with robust query performance at google. *Proc. VLDB Endow.*, 14(12):2986–2998, 2021.
[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. *VLDB*, (5):496–505, 2000.
[4] R. Ahmed, R. Bello, and A. Witkowski. Automated Generation of Materialized Views in Oracle. *VLDB*, 2020.
[5] P. Benats, L. Meurice, M. Gobert, and A. Cleve. Query-based schema evolution recommendations for hybrid polystores. In *ER Forum/PhD Symposium*, volume 3211 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.
[6] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39:12–27, 2011.
[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.
[8] A. Conrad, M. L. Möller, T. Kreiter, J. Mair, M. Klettke, and U. Störl. Evobench: Benchmarking schema evolution in nosql. In *TPCTC*, volume 13169 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2021.
[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.
[10] J. C. Corbett et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, 2013.
[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220. ACM, 2007.
[12] A. Hillenbrand, M. Levchenko, U. Störl, S. Scherzinger, and M. Klettke. MigCast: Putting a price tag on data model evolution in NoSQL data stores. In *SIGMOD Conference*, pages 1925–1928. ACM, 2019.
[13] A. Hillenbrand, U. Störl, M. Levchenko, S. Nabiyev, and M. Klettke. Towards Self-Adapting Data Migration in the Context of Schema Evolution in NoSQL Databases. In *ICDEW*, pages 133–138, 2020.
[14] R. Hyndman and G. Athanasopoulos. *Forecasting: Principles and Practice*. OTexts, Australia, 2nd edition, 2018.
[15] A. Jindal, H. Patel, A. Roy, S. Qiao, Z. Yin, R. Sen, and S. Krishnan. Peregrine: Workload optimization for cloud query engines. In *SoCC*, pages 416–427, 2019.
[16] A. Jindal, H. Patel, A. Roy, S. Qiao, Z. Yin, R. Sen, and S. Krishnan. Peregrine: Workload optimization for cloud query engines. In *SoCC*, pages 416–427, 2019.
[17] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *SIGMOD*, pages 191–203, 2018.
[18] A. Jindal Konstantinos Karanasos Sriram Rao Hiren Patel Microsoft, A. Jindal, K. Karanasos, S. Rao, H. Patel, and H. P. Microsoft. Selecting Subexpressions to Materialize at Datacenter Scale. *PVLDB*, 11(7):800–812, 2018.
[19] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. Zdonik. Coradd: Correlation aware database designer for materialized views and indexes. *PVLDB*, 3:1103–1113, 09 2010.
[20] J. Kossmann and R. Schlosser. A framework for self-managing database systems. In *ICDEW*, pages 100–106, 2019.
[21] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.
[22] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD*, page 631–645, 2018.
[23] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. NoSE: Schema design for NoSQL applications. *TKDE*, 29(10):2275–2289, 2017.
[24] M. Mozaffari, E. Nazemi, and A. Eftekhari-Moghadam. CONST: Continuous online NoSQL schema tuning. *Software: Practice and Experience*, 2020.
[25] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR*, 2017.
[26] N. Sasaki. Large-scale high-speed processing of smart meter data following the deregulation of electrical power. https://www.global.toshiba/ww/company/digitalsolution/articles/tsoul/22/004.html, Aug 2017.
[27] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, K. Littlefield, and P. Tong. F1: the fault-tolerant distributed RDBMS supporting google's ad business. In *SIGMOD Conference*, pages 777–778. ACM, 2012.
[28] U. Störl and M. Klettke. Darwin: A data platform for schema evolution management and data migration. In *EDBT/ICDT Workshops*, volume 3135 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.
[29] P. Suárez-Otero, M. J. Mior, M. J. S. Cabal, and J. Tuya. Codevo: Column family database evolution using model transformations. *J. Syst. Softw.*, 203:111743, 2023.
[30] K. F. T. Vajk, L. Deak and G. Mezei. Automatic NoSQL schema development: A case study. In *PDCN*, 2013.
[31] T. Takata, H. Furusawa, Y. Okura, Y. Yamada, M. Onizuka, H. Suga, R. Kurosawa, and T. Kambayashi. Toward fast search and real-time inputs of big astronomical catalogs by the new generation relational database. In *Astronomical Society of the Pacific Conference Series*, number 527, page 717, 2020.
[32] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. CrocodileDB in action: Resource-efficient query execution by exploiting time slackness. *Proc. VLDB Endow.*, 13(12):2937–2940, 2020.
[33] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017.
[34] D. Wiese, G. Rabinovitch, M. Reichert, and S. Arenswald. Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning. In *CASCON, 27-30*, page 3, 2008.

.1 Additional Experiments

.1.1 Workload Evolution and Stagnation. Figure 6b shows the workload latencies of the baseline methods using the **Evolution and Stagnation** pattern. The findings reveal several important observations. First, our proposal achieves an impressive latency reduction of 12.2-20.5% compared to the static optimization methods that do not adapt to the time-dependent workload. Furthermore, the static method called `average freq.` is respectively 9.5% and 8.9% more efficient than the other static options `first time step freq.` and `last time step freq.`, making it the best static technique among those tested. Finally, the `ideal` optimization approach, representing the theoretical optimal performance, demonstrates a reduction of 3.4% in latency compared to our method, outperforming all static methods by more than 15%. Our approach not only achieves impressive reductions in latency, but also manages to do so while efficiently computing the schema design, making it a valuable tool for optimizing system performance for time-dependent workloads. This highlights the dual advantage of our approach: reducing latency while maintaining computational efficiency in the schema optimization.

We further investigate how query plans change in response to workload changes. Table 2 shows how the query plan changes and at what time steps. For example, join plans are used for Q3, Q5, Q7, Q9 at time step 5 since their frequency is low (see the query frequency changes in the Evolution and Stagnation pattern in Figure 5). Once we move to time step 6, their frequency becomes high, so the schema is optimized so that materialized view plans are used for those queries. In contrast, the queries in the duplicate group (Q3-dup, Q5-dup, Q7-dup, Q9-dup) take opposite plan changes to those in the original group.

time step	plan change	query
5 → 6	MV Plan → Join Plan	Q3-dup, Q5-dup, Q7-dup, Q9-dup
	Join Plan → MV Plan	Q3, Q5, Q7, Q9

Table 2: How the query plan changes at which time steps for the Evolution and stagnation workload pattern under 80% storage constraint.

.1.2 Growth and Spikes. Fig. 6c shows the workload latencies of the baseline methods using the **Growth and Spikes** pattern. Firstly, our proposed solution stands out because it is only 2% less efficient than the `ideal` method. This underlines its ability to handle sudden spikes in query frequency and optimize system response times efficiently. In contrast, static optimization methods such as first time step freq. and last time step freq. show limited adaptability, with reductions of around 12% in latency compared to our proposal. These methods struggle to cope with the dynamic and unpredictable nature of the spike frequency pattern. The average freq. method, which calculates the average frequency of queries, performs slightly better but is still less efficient than our proposed solution.

.2 Limitation in the SQL specification

Our system does not fully support the SQL specification. First, our system does not handle queries without equality conditions. Second, it does not support several predicates, such as `LIKE` and `BETWEEN`. Third, it does not support nested queries. We have detours for the second and third limitations; rewriting some predicates to equality and range conditions, and decomposing a nested query into multiple flat queries. We use these detours in our experiments when workloads face these limitations. We note that NoSE also has the same limitations. Furthermore, NoSE cannot support aggregate functions such as `SUM` and `AVERAGE`. Thus, our system supports richer queries than NoSE.