Optimizing Time Series Queries with Versions

RUI KANG, Tsinghua University, China
SHAOXU SONG*, Tsinghua University, China

We show that the time-series database for industrial IoT data management exhibits intrinsic demands for integrating an automatic version control system, which introduces advanced data semantics and query optimization. In deployed IoT database instances, IoT data managed by an LSM tree is multi-leveled and multi-versioned due to network issues and erroneous IoT readings. For data semantics, each query merges versioned data according to query expressions or data block levels. For query optimization, we find that existing time-series databases relying on write-ahead-logs suboptimally execute data queries, due to their performance bottlenecks in merging numerous versioned data. In this paper, an algebra consisting of version operators addresses the semantics for time-series applications to evaluate and optimize physical query plans. We propose version reducibility as a key feature of executing consistent plans and evaluate the benefits of putting off data merges. We also show the integration of version queries to existing relational databases by translating them to standard SQL based on relational reducibility. Finally, our extended experiments show the effectiveness of optimizing execution plans over versioned data.

CCS Concepts: • **Information systems → DBMS engine architectures**; *Query operators*; **Spatial-temporal systems**.

Additional Key Words and Phrases: Time-series query; IoT database

ACM Reference Format:
Rui Kang and Shaoxu Song. 2024. Optimizing Time Series Queries with Versions. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 159 (June 2024), 27 pages. https://doi.org/10.1145/3654962

1 INTRODUCTION

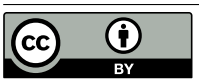
Data generated by the Industrial Internet of Things (IIoT) are typically time series [22]. Each pair of timestamps on a single timeline and series name corresponds to a series value [24]. When receiving data packets from IoT networks, noisy time-series data would occur frequently, challenging current IoT databases.

1.1 IoT Data and LSM Trees

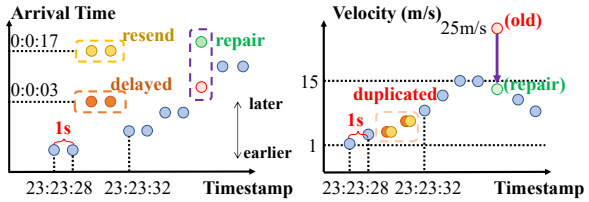
Since each IoT time series is generated with high frequency, such as every millisecond, challenging the persistent performance of B-tree-based databases [24]. IoT data are kept in logs and maintained by a log-structured merging (LSM) tree, for example, the Apache IoTDB [2]. The LSM tree [37] is designed for persisting key-value pairs. Generally, the LSM tree first buffers time-series tuples in a memory table and flushes the buffer to a run sorted by timestamps on secondary storage when it reaches the allocated size. In a time-series database, each sorted run contains the tuples from multiple time series, and the tuples are grouped according to the series name in each run. When IoT data arrives in the same order as it was generated without any updates, the log-structured files

*Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

Authors' addresses: Rui Kang, Tsinghua University, China, kr20@mails.tsinghua.edu.cn; Shaoxu Song, Tsinghua University, China, sxsong@tsinghua.edu.cn.



ACM 2836-6573/2024/6-ART159
https://doi.org/10.1145/3654962



(a) Arrival time of tuples; (b) Repairs of abnormal data

Fig. 1. Versioned time-series data.

do not overlap, and each query is planned as sequential scans of all related runs. This contributes to a trivial LSM tree that does not rely on compaction to update data. However, the IoT data from real-world applications is multi-versioned, and the LSM tree is not as simple as expected. The IoT database relies on the network to collect the tuples, leading to delayed or duplicated data packets.

Example 1. *Figure 1 shows major reasons for a non-trivial LSM tree in an IoT database. Industrial sensors are connected with programmable logic controllers (PLCs), which occasionally sample and upload the states of sensors to the database [11, 42]. Due to network issues, packets to the database may be delayed for reasons such as system recovery [7, 20, 45, 48] and network congestion [31, 43]. Without backward acknowledgment (ACK) messages for delayed packets, i.e., timeout, the communications protocol asks PLCs to resend IoT packets. As network routers consume delayed packets first, we notice a resend of the same packet arriving later than the delayed one. For example, in Figure 1(a), upstream routers resend a delayed packet generated at 23:23:30. This results in unordered and duplicated tuples in the time-series database.*

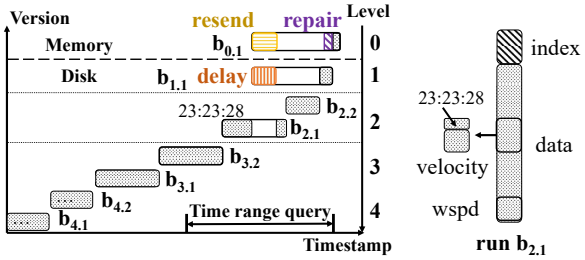
In addition, there could be erroneous readings by IoT devices. Errors are widely observed in time-series data, e.g., abnormal series values [17, 44] and timestamps [18]. Figure 1(b) shows the records generated from a speedometer. We generate a speed constraint to repair data based on ground truths, such as $\frac{|Velocity(t_1)-Velocity(t_2)|}{|t_1-t_2|} \leq 5m/s^2$*, which restricts the speed variances of every second to be less than* $5m/s^2$*. An averaged attribute value of a former and latter tuple repairs the erroneous tuple, marked by the red circle in Figure 1. The database inserts the repaired tuple* $\langle$*23:23:36*, 14$\rangle$*, marked by the green circle, which arrives later than the red one in Figure 1(a).*

Handling messages by middleware like Apache Kafka and RabbitMQ, featuring different use cases, is possible in the cloud. However, IoT databases like Apache IoTDB are often installed in gateways or edge devices, with limited computing resources, high throughput, and unknown routing logic. Given insufficient memory, it is often hard to persist records to fix all delays and correct all records through a middleware or a lower-level layer. Thereby, LSM-tree is employed in product systems such as Apache IoTDB and InfluxDB, to handle the intensive data ingestion from millions of sensors with delayed, duplicated, and repaired data. Our proposal focuses on optimizing the query processing over the LSM-tree data by exploring the versions with branches.

1.2 Optimizing Queries with Versions

Multiple approaches have focused on optimizing queries over the LSM tree, including fast point filters [13, 15, 16, 28], efficient range filters [33, 34, 49, 50],and new leveling policies on I/O tradeoffs [12, 14]. They focus on optimizing LSM trees based on estimated time consumption, rather than optimizing plans of specific workloads over log-structured data.

Example 2. *Figure 2 shows an LSM tree for managing the multi-version time-series data. The LSM tree consists of the buffer in memory and sorted runs in secondary storage, where each run is grouped*



(a) Ranges of the series velocity in each run; (b) Run components

Fig. 2. Multi-version time series are stored in the LSM tree. (a) As the data could be delayed, each run would contain several slices of the same series. (b) The time-series database receives the data from more than one sensor, and each run contains data arriving in the same period, e.g., the velocity and thermal data.

by the series name and sorted by the timestamp. The data from the lower level runs update higher level runs. Figure 2(a) shows the time ranges of the velocity data, with some portions left blank due to delayed network packets, as illustrated by Figure 1. To answer a range query in Figure 2(a), existing databases will merge the overlapped runs throughout the levels from level 0. Loading and updating data requires sufficient disk I/O. However, some updates are unnecessary, like merging delayed packets $b_{1.1}$ with $b_{2.2}$. When wind speed (wspd) is generated at the start of every minute, the natural join on timestamps between velocity and wspd also meets unwanted updates between $b_{0.1}$ and $b_{1.1}$. Thus, some operations from the query plans could be applied before loading and updating data. The operations with selectivity, including time and value filters, and alignment (natural join on timestamps), will reduce the I/O and CPU usage when answering queries.

Existing works or databases execute the queries over an LSM tree sub-optimally. To distinguish the update priority of runs, we introduce versions in query plans as a hint to query engines, where runs created at larger timestamps have higher priority to update, like Figure 2(a).

1.3 Contributions and Overview

Our major contributions are summarized as follows.

(1) Section 2 formalizes the versioned data model and discusses versioned data semantics.
(2) Section 3 shows the versioned time series algebra to construct execution plans. Notice that updating the whole series is time-consuming, version reducibility exchanges the execution of updates and other operators with consistent execution results, concluded as optimization rules.
(3) Section 4 discusses the practical details of managing versioned time-series data based on the Apache IoTDB. Further in Section 5, we translate version queries to SQL based on relational reducibility for relational databases like PostgreSQL.
(4) Section 6 compares our proposals integrating optimization rules with other baselines on real-world version queries on IoT data. It indicates the effectiveness of our optimization methods.

2 VERSIONED TIME-SERIES DATABASE

In this section, we introduce the versioned time series data for handling updates through versions by formalizing the data model in Section 2.1 and semantics in Section 2.2. We also show practical version management through branches for IoT data in Section 2.3.

Table 1. Definitions of formal notations

Notation	Description
$S = \{T, \mathbf{V}, \mathbf{R}\}$	Model of schema, timestamps, version and series values
$T, A \in \mathbf{R}, v \in \mathbf{V}$	Timestamps, series attributes and versions
$\langle t, \mathbf{v}, \mathbf{r}\rangle \in I$	A tuple from versioned series instance I
D	A versioned time-series database
$ts(D)$	A time-series database
φ, q	Tuple calculus formula, query by semantic rules
e, Q	Expression (execution plan), query by expression
$\alpha_\theta; \alpha_\Delta$	Filter selectivity; Ratio of updates and duplicates
B	Number of blocks in IoT data store
α_T	Ratio of data blocks without time range overlaps

2.1 Versioned Time Series Data

Motivated by Example 2, we extend existing data models of time series [19, 26, 39] to a versioned time series model $S = \{T, \mathbf{V}, \mathbf{R}\}$ for timestamp, version variables, and attributes, respectively. We use version numbers to distinguish updates and the original record of each timestamp. For example, $\mathbf{R} = \{wspd, velocity\}$ is the relational part of series *wspdCollect*. We consider the versions variables $v_1, v_2, \ldots$ with their values $v_i \in \mathbb{N}$. Each attribute $A \in \mathbf{R}$ has a version number, namely $v_A \in \mathbf{V}$.

Example 3 (Version for updates). *Consider a simple case of versioned single-dimensional time series $I = \{\langle t, v, a\rangle, \langle t, v', a'\rangle\}$ of schema $S = \{T, v_A, A\}$. When $v' > v$, the tuple $\langle t, v, a\rangle \in I$ is updated by $\langle t, v', a'\rangle$. When persisting them in different LSM blocks, the timestamps of writing two blocks could be the version of each tuple. In Figure 2, the LSM tree will write $b_{1.1}$ earlier than $b_{0.1}$. Tuples from block $b_{0.1}$ can update the tuples of the same timestamp in block $b_{1.1}$.*

When each tuple has one version attribute, it is easy to distinguish an updated tuple from a deprecated one. However, time-series alignment (natural join on timestamps) leads to more than one version dimension. We need to compare versions on multiple dimensions and find updated attribute values. We define the series update $\Delta(I)$ by updating each attribute.

Definition 1 (Version update). *Let I denote a version instance of schema $\{T, \mathbf{V}, \mathbf{R}\}$. Consider an attribute $A \in R$. For each tuple $\langle t, \mathbf{v}, \mathbf{r}\rangle$, the version update operator modifies the value of attribute A by another tuple $\langle t', \mathbf{v}', \mathbf{r}'\rangle$ satisfying, $t' = t, \mathbf{v}[A] < \mathbf{v}'[A]$. Formally, the version update is applied to an instance I with $\Delta_A(I) = \{s \mid s[A] := s'[A], s, s' \in I, s'[v_A] = \max_{s_x \in I, s[T]=s_x[T]} s_x[v_A]\}$. Given $\mathbf{R} = \{A_1 \ldots A_n\}$, it could be applied to the whole relation, $\Delta(I) = \Delta_{A_1}(\ldots(\Delta_{A_n}(I)))$.*

Example 4. *We use this example to illustrate the version update operator. Consider the timestamps of writing $b_{0.1}$ and $b_{1.1}$ as 2 and 1. We observe two single-variate tuples $\{\langle t_1, a_1\rangle, \langle t_1, b_1\rangle\}$ stored in $b_{1.1}$, which are updated by $\{\langle t_1, a_2\rangle, \langle t_1, b_2\rangle\}$ from $b_{0.1}$. Before the arrival of $b_{0.1}$, an execution aligns two tuples by timestamps as $\langle t_1, a_1, b_1\rangle$, having the version number $\langle v_A, v_B\rangle$ as $\langle 1, 1\rangle$. The versioned aligned tuple is further updated by $\langle t_1, a_2, b_2\rangle$ with version $\langle 2, 2\rangle$. Each application of the version update operator modifies an attribute value from the aligned tuple, leading to an updated tuple.*

Example 4 indicates that version numbers of all time-series attributes can identify an updated instance and so as an updated versioned time-series database. The version instances and the correspondent updated databases construct an abstract versioned time-series database. Since the version values are discrete, we have $\mathrm{adom}(v)$ as the finite active domain of any version number in the versioned database D, and we consider $\mathrm{adom}^{\mathrm{full}}(v) = \mathrm{adom}(v) \cup \{\mathrm{now}\}$ as the finite active

domain of versions. A version tuple $\mathbf{v}$ of schema $\mathbf{V}$ is taken from $\prod_{\nu \in \mathbf{V}} \mathsf{adom}^{\mathsf{full}}(\nu) = [\mathsf{adom}^{\mathsf{full}}(\nu)]^{\mathbf{V}}$. The abstract versioned database is a finite set of databases, $\mathbb{D} = \{D_{\mathbf{v}_1}, D_{\mathbf{v}_2}, \ldots, D_{\mathbf{v}^{\mathsf{now}}}\}$.

2.2 Semantics of Abstract Versioned Database

Version views help us to evaluate a query on an updated versioned TSDB. We formalize the semantics as follows to determine whether a query could be executed within the versioned TSDBs.

Definition 2 (Semantics). *Let $\{TS_1, TS_2, \ldots\}$ be a time-series schema. With the set of versions, $[\mathsf{adom}^{\mathsf{full}}(\nu)]^{\mathbb{V}(D)}$, an abstract versioned time-series database $\mathbb{D}$ is indexed as a finite set of database states $\mathbb{D} = \{D_{\mathbf{v}_1}, \ldots, D_{\mathbf{v}^{\mathsf{now}}}\}$. Each database state $D_{\mathbf{v}}$ collects the time series ts for each time-series schema TS. The semantics are defined by logic formulas below, where we introduce the symbol of satisfaction relationship $\vDash$ for the time series, relating true logic formulas to the abstract versioned time-series database D w.r.t. a set of versions $\mathbf{v}$ and valuation assignment Γ that maps each free variable to a value.*

$-D, \Gamma, \mathbf{v} \vDash TS(T, A_1, \ldots)$ iff *TS is atomic with tuple* $\langle \Gamma(T), \Gamma(A_1), \ldots \rangle \in$ *ts of schema TS in TSDB* $D_{\mathbf{v}}$;

$-D, \Gamma, \mathbf{v} \vDash (T \phi T')$ iff $\Gamma(T) \phi \Gamma(T')$;

$-D, \Gamma, \mathbf{v} \vDash (T \phi t)$ iff $\Gamma(T) \phi t$;

$-D, \Gamma, \mathbf{v} \vDash (A \phi A')$ iff $\Gamma(A) \phi \Gamma(A')$;

$-D, \Gamma, \mathbf{v} \vDash (A \phi a)$ iff $\Gamma(A) \phi a$;

$-D, \Gamma, \mathbf{v} \vDash \xi \wedge \psi$ iff $D, \Gamma, \mathbf{v} \vDash \xi$ *and* $D, \Gamma, \mathbf{v} \vDash \psi$;

$-D, \Gamma, \mathbf{v} \vDash \xi \vee \psi$ iff ξ *or* ψ *is not an atomic term, meanwhile* $D, \Gamma, \mathbf{v} \vDash \xi$ *or* $D, \Gamma, \mathbf{v} \vDash \psi$;

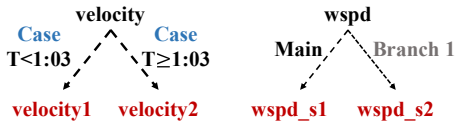
$-D, \Gamma, \mathbf{v} \vDash (\exists X)\xi$ iff *there exists a constant x s.t.* $D, \Gamma[X \to x], \mathbf{v} \vDash \xi$,

where $TS(T, A_1, \ldots)$ *is an atomic term if and only if the schema TS is atomic, and* $\Gamma[X \to x]$ *is the valuation* Γ *given the assignment of variable X by constant x.*

Here, X is a variable of either timestamps T or attributes A. The constants are represented as t, a, x. For the disjunction of two atomic formulas, we introduce branching in the versioned time-series database to check and merge time series in Section 2.3. Let ξ be any first-order formula, the answer to a versioned time-series query is written as, $q(D, \mathbf{v}, \xi) = \{\Gamma \mid D, \Gamma, \mathbf{v} \vDash \xi\}$.

Example 5 (First-order Query). *An example first-order time-series query aligns two instances, $\{ts_1, ts_2\}$. Provided ts_1 of schema $TS_1 = \{T_1, A_1\}$ and ts_2 of $TS_2 = \{T_2, A_2, A_3\}$, we have calculus $q(D, \mathbf{v}^{\mathsf{now}}, \xi)$, $q = \{T, A_1, A_2, A_3 \mid (\mathbf{v} = \mathbf{v}^{\mathsf{now}}) \wedge TS_1(T, A_1) \wedge TS_2(T, A_2, A_3) \wedge (T \geq t) \wedge (A_1 > a)\}$, where the tuple calculus is applied over the database state D indexed by $(\mathbf{v} = \mathbf{v}^{\mathsf{now}})$, and ξ is the conjunction of predicates over timestamp T and relational attributes $A \in \mathbf{R}$. In the above calculus, the time-series query consists of time alignment $(TS_1(T, A_1) \wedge TS_2(T, A_2, A_3))$, range query $(T \geq t)$ and value filtering $(A_1 > a)$.*

Besides first-order queries on time series, aggregation queries to evaluate time slices are also important for IoT applications [22, 30],e.g., by second-order formulas. Consider the calculus $q(D, \mathbf{v}, \xi)$ that finds a query answer based on the version view $D_{\mathbf{v}}$, the aggregation takes a function $f : Inst(TS) \to \mathbb{Q}$ to map each time-series sub-sequence to a rational number. The semantic rule for aggregation in time-series query is defined as, $D, \Gamma, \mathbf{v} \vDash \Psi : (\exists A)(A = f(q(D, \mathbf{v}, \xi)))$, which restricts that both query calculus q and semantic rules are built on the same version view of the database, and we use Ψ when at least one of the formulas is not first-order, e.g., $\xi \wedge \Psi$ is also marked as Ψ. The above semantic rule aims to find the statistics by f over a first-order query.



(a) Series collaboration by time ranges

(b) Series collaboration by specified priority

Fig. 3. Version control through branching: collaboration by ranges and priority.

Table 2. Semantics of versioned time series algebra

Operator	Definition
$\sigma_{\theta(TS)}(I)$	$\{s \mid s \in I, s \vDash \theta(TS)\}$
$\pi_{\mathbf{R}'}(I)$	$\{s \mid s \in I, s.\mathbf{r} = \mathbf{r}[\mathbf{R}']\}$
$\cup[I_1, \dots]$	$\{s \mid \forall t, s[A] = I_x(t).A, x : \arg\min_i(I_i(t).A \neq \text{null})\}$
$I_1 \bowtie_T I_2$	$\{s_1 \cup s_2 \mid (s_1, s_2) \in (I_1, I_2), t_1[T] = t_2[T]\}$
$\rho_{T',\mathbf{R}'}(I)$	$\{s' \mid s \in I, s'[T', \mathbf{V}', \mathbf{R}'] = s[T, \mathbf{V}, \mathbf{R}]\}$
$G_{T_g:f}(I)$	$\{s \mid s \in \Delta(I), s.t = t_g, s[A_f] = f(\sigma_{T=t_g}(\Delta(I))\}$

Example 6 (Aggregation Query). *To find out the average of the attribute values A_1 over sliding windows on a minute length, the aggregation is written by* $q_{\text{avg}}(D, \mathbf{v}^{\text{now}}, \Psi)$, $q_{\text{avg}} = \{T, A \mid TS_1(T, A_1) \wedge (\exists A)(A = \text{avg}(\{T', A_1' \mid TS_1(T', A_1') \wedge (T' \geq T - 1\text{min}) \wedge (T' < T)\}))\}$, *where we consider the version view on* $\mathbf{v}^{\text{now}}$ *as default, and the predicate* $(\mathbf{v} = \mathbf{v}^{\text{now}})$ *is not added. Here, the variable T_1 is referred to and treated as a constant in the range query inside the function* avg, *which finds the average value of the attribute A_1.*

2.3 Collaboration by Branches

Similar to existing version control database systems [41], we introduce branching to identify different contributions to the same instance, e.g., a single-variate or multivariate time series. Series collaboration is helpful for data analysis under unreliable IoT sensors through merging series branches with priority. We could insert data tuples into different branches when the database accepts arrival data packets. Each branch maintains a set of updates. For example, Figure 3 demonstrates two applications of using collaboration branches. When the time ranges from the two branches do not overlap, it is the first case in Figure 3(a). Otherwise, we use the priority of branches to decide the update order. The multi-leveled series collaboration can support the series merging on varied priority combinations. It describes the version merge order, which fits for managing runs in LSM trees. For example, the main branch of *wspd* contains data from $b_{1.1}$ and treats the incoming data $b_{0.1}$ as the update branch of $b_{1.1}$.

3 VTSA: VERSIONED TIME-SERIES ALGEBRA

Based on semantic analysis of version data, we formalize the required operators to compose an execution plan for version query processing engines. Here, we first introduce the operators, named Versioned Time Series Algebra. We also show optimization chances based on version reducibility to change the execution order while guaranteeing a consistent executing result.

3.1 Syntax and Operator Semantics

In Section 2.2, we formalize the semantic rules of version time series. The operators in Table 2 form an expression e executable by a version query engine, i.e.,

$$e ::= I \mid \sigma_\theta(e) \mid \pi_{\mathbf{R}'}(e) \mid (\cup[e_1, \dots]) \mid (e_1 \bowtie_T e_2) \mid G_{T_g,f}(e) \mid \rho_{T',\mathbf{R}'}(e),$$

named (time/value) range filtering, projection, (union) branch merging, alignment, aggregation by time groups, and renaming. An expression is either a versioned time series I from the database D or a syntactic composition of expressions. We also allow the basic operators Δ in an expression e. For the operator $\sigma_{\theta(TS)}(e)$, condition $\theta(TS)$ is defined as, $\theta(TS) = (C_1(TS) \vee C_2(TS) \cdots \vee C_n(TS))$, with each conjunction $C_i(TS) = p_{i,1}(TS) \wedge \cdots \wedge p_{i,m}(TS)$ collecting a set of predicates over the schema $TS = S \setminus \mathbf{V} = \{T, \mathbf{R}\}$. Each predicate $p(TS)$ is either in the form of $p(T) : T_i \phi T_j, T_i \phi t$ or $p(\mathbf{R}) : A_i \phi A_j, A_i \phi a$, where $T_i, T_j \in T$, $A_i, A_j \in \mathbf{R}$, t, a are constants, and $\phi \in \{>, \geq, =, <, \leq\}$. Given an instance I with its relational part $\mathbf{R}$, the projection $\pi_{\mathbf{R}'}$ has a constraint, $\mathbf{R}' \subseteq \mathbf{R}$. Branch merging takes an ordered list as input, with the order representing the priority of choosing instances or branches. For example, we update series *wspd* by merging the main branch and the branch 1 in Figure 3(b) by expression $wspd = \cup[wspd_{b1}, wspd_{main}]$. The aggregation by time operator ($G_{T_g:f}$) takes two parameters, the group-by dimensions of time T, and the aggregation mapping function $f : Inst(TS) \to \mathbb{Q}$ for a rational number.

3.2 Version Reducibility

Given the execution plan composed of version operators in VTSA, the query performance benefits from pushing down the operators, e.g., time range filters, as noticed in Example 2. However, unlike relational algebra, versioned updates are tuple-dependent in execution plans. Without looking up other tuples, changing the execution orders leads to inconsistent execution results.

Example 7 (Filtering Tuple-dependent Data). *Consider an instance I_{main} of schema $\{T, wspd\}$ containing $\langle$1:03, 1.5$\rangle$, which is updated by an instance from branch $I_b : \{\langle$1:03, 2.5$\rangle\}$ with higher priority. We express a value filtering query by $\sigma_{wspd<2}(\Delta(I))$. After changing the execution order of value filtering and versioned updates, we have to remove the record from branch b, which updates the tuple from the main branch. Thus, we have $\sigma_{wspd<2}(\Delta(I)) \neq \Delta(\sigma_{wspd<2}(I))$.*

Motivated by this example, we need constraints when changing the execution orders to ensure the equivalence before and after the execution order changes, formalized as version reducibility.

Definition 3 (Version Reducibility). *Let $q(D, \mathbf{v}, \varphi)$ denote any time-series query with φ as a SQL or tuple calculus query over the time-series database $ts(D_\mathbf{v})$. A time-series query q is version reducible concerning an algebraic expression e' if and only if, $ts(\Delta(Q(D, e'))) = q(D, \mathbf{v}, \varphi)$, where $Q(D, e')$ represents the result of executing the expression e' over the instances in the versioned database D. The version reducibility consists of translation and algebraic version reducibility. Translation introduces an algebraic expression e equivalent to the time-series query φ under database state $D_\mathbf{v}$. Specifically, translation finds an algebraic expression e, satisfying $ts(Q(\Delta(D), e)) = q(D, \mathbf{v}, \varphi)$. Finally, an expression e is algebraic version reducible concerning an expression e', if and only if, $ts(\Delta(Q(D, e'))) = ts(Q(\Delta(D), e))$. Figure 4 illustrates the relationships of version reducibility.*

Motivated by version reducibility, we analyze whether changing the execution order of version update and other operators in VTSA leads to consistent execution results. When an expression consists of version-reducible operators, the operators could be pushed down ahead of the version update. For example, consider version instance I of schema $\{T, A\}$. The expression $e : \sigma_{T>t_1 \wedge T<t_2}(\Delta(I))$ uses a time range to filter the updated instance. Since filtering tuples on time will keep the deprecated and updates simultaneously, we could push down the filter, resulting in an equivalent

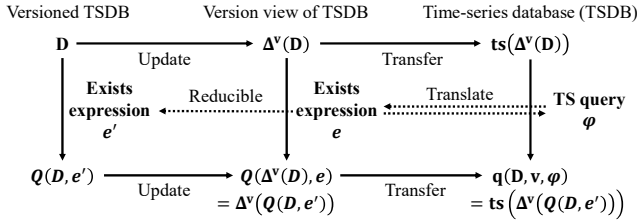


Fig. 4. Version reducibility: for consistent query execution directly over versioned data.

expression $e' : \Delta(\sigma_{T>t_1 \wedge T<t_2}(I))$. The expression e' avoids loading each tuple from the data store and saves execution resources. To advance, assume we can update instances by merging branches. we show consistent operators that stand for the operators that naturally satisfy version reducibility, including branch merging (union) and aggregation. They are either used to update instances or work on updated ones. We have $\Delta(\cup[I_1, \dots]) = \cup[\Delta(I_1), \dots]$ and $\Delta(G_{T_{g:f}}(I)) = G_{T_{g:f}}(\Delta(I))$. Since they rely on timestamps to update, find differences, and find groups, we also optimize them with non-overlapping components.

In the following sections, we show optimization rules based on the version reducibility of each operator and analyze the reduction of execution cost, consisting of disk I/O and CPU usage.

3.3 Simple Version Reducible Operators

We show the class of simple reducible operators that guarantee consistent execution by simply exchanging the operators.

Corollary 1. *Time range filtering, projection, and renaming are version-reducible, satisfying* $op(\Delta(I)) = \Delta(op(I)), op \in \{\sigma_{\theta(T)}, \pi, \rho\}$.

The reducibility of time range filtering and projection indicates potential optimization chances by executing them ahead of updates. IoT data is accumulated over time, meaning that the tuples of a time series will be stored separately in log files. Each file contains tuples that arrive simultaneously, as shown in Figure 2. Thus, time range filtering could avoid loading unnecessary data blocks from the disk. Assume that there are total B blocks for all files and each time series is distributed evenly in these files. In addition, we use α_Δ to represent the ratio of updated and duplicated data. In other words, after branch merging, we have $(1 - \alpha_\Delta)B$ blocks left.

Proposition 2 (Time range optimizing analysis). *Let* $0 \le \alpha_\theta = \frac{|\sigma_\theta(I)|}{|I|} \le 1$ *denote the selectivity ratio of time range* $\theta(TS)$. *Pushing down the time range filter ahead of update executions saves the disk I/O of* $O((1 - \alpha_\theta)B)$. *It reduces a CPU consumption of,*

$$O((1 - \alpha_\Delta - \alpha_\theta)B(|\mathbf{R}| + 2)T_{filter} + (1 - \alpha_\theta)BT_{merge}),$$

where T_{merge} *and* T_{filter} *denote the cost of merging two blocks and filtering a block with a predicate, respectively.*

In Proposition 2, the time range filter only selects $O(\alpha_\theta B)$ data blocks to load into memory. Since the log file of LSM may contain tuples from unrelated time series, the CPU usage consists of (1) finding related tuples from version files, (2) range filtering on these tuples, and (3) branch merging. Since the projection is version-reducible, we could extract the related attributes from versioned log files before updating them. Thus, for reduced expression, the first two parts consume $O(\alpha_\theta B(|\mathbf{R}| + 2)T_{filter})$. Here, $(|\mathbf{R}| + 2)T_{filter}$ is the CPU usage of finding related time-series tuples and applying two predicates of a time range filter, such as $T > t_1 \wedge T < t_2$. When files are loaded

in time order, we could merge and update the time series by blocks at the cost of $O(\alpha_\theta B \cdot T_{merge})$. Without version reduction, block merging costs $O(B \cdot T_{merge})$ and results in $O((1-\alpha_\Delta)B)$ blocks to filter. The reduced CPU usage equals the estimated cost difference.

3.4 Reducing Value Filter with Branch Merge

Given the analysis of the non-reducible value filter in Example 7, we show that certain conditions could make value filters reducible. The condition specifies a finite list of instances to update each related versioned time series.

Corollary 3 (Value filter reducibility). *Let $[I_1 \dots I_k]$ denote a finite list of all branches or blocks to merge with priority. The value filter is reducible to non-overlapping instances according to the time ranges of merging instances. That is,*

$$ts(\sigma_\theta(\cup[I_1, \dots I_k])) = ts(\cup[\sigma_\theta(I_1), \dots \sigma_\theta(\cup[I_\ell, \dots]), \dots]),$$

where each pair of components such as $\langle I_1, \cup[I_\ell, \dots]\rangle$ does not overlap in the time range and the reduced expression at RHS follows the same instance order as LHS.

The conclusion shows that the value filter is reducible when the merged instances do not share timestamps. Otherwise, we need to merge all overlapped branches before applying the filter.

Example 8. *Example 7 shows the value filtering of updated I_{main} according to I_b, which overlaps on timestamp 1:03. When introducing another branch $I_{b'}$ with timestamp 1:05, we have,*

$$\sigma_{A<2}(\cup[I_{b'}, I_b, I_{main}]) = \cup([\sigma_{A<2}(I_{b'}), \sigma_{A<2}(\cup[I_b, I_{main}])]).$$

Proposition 4 (Value range optimizing analysis). *Let α_θ denote the selectivity of a value filter and $\alpha_T \cdot B$ denote the number of non-overlapping components, $\alpha_T < 1$. The pushing down reduces CPU consumption of $O(\alpha_T B(T_{merge} - \alpha_\Delta(|\mathbf{R}|+2)T_{filter}))$. When each storage file saves value ranges of content time series in its header, the push-down of value filters saves disk I/O of $O((1-\alpha_\theta)\alpha_T B)$.*

We count the non-overlapping blocks $\alpha_T B$ based on time ranges. In Example 8, α_T is 1/3 as only one block $I_{b'}$ does not require merging. The expression $\Delta(\sigma_\theta(I))$ reads B blocks from disk and consumes a CPU time of $O(BT_{merge} + (1-\alpha_\Delta)B(|\mathbf{R}|+2)T_{filter})$. Similar to Proposition 2, $(|\mathbf{R}|+2)T_{filter}$ denotes the CPU cost of finding tuples and applying two range predicates. The filter applies to $(1-\alpha_\Delta)B$ blocks because block merging consumes updating records and eliminates duplicates.

After applying Corollary 3, we execute the value filter directly on $\alpha_T B$ blocks and merge $(1-\alpha_T)B$ blocks. Then, we apply the value filter to merged blocks $(1-\alpha_\Delta)(1-\alpha_T)B$. The filtering process consumes $O((1-\alpha_\Delta)(1-\alpha_T)B(|\mathbf{R}|+2)T_{filter})$. We combine block merging and filtering time estimations to get the saved CPU time.

In addition, we show avoided disk I/O when IoT storage files keep both value and time ranges. The file value ranges help avoid loading unnecessary data blocks, similar to the time ranges. Value filter pushing down only applies to $O(\alpha_T B)$ blocks and saves the disk I/O of $O((1-\alpha_\theta)\alpha_T B)$.

3.5 Reducing Alignment with Branch Merge

In addition to value filters, the alignment is also reducible when branch merging can represent version updates.

Corollary 5 (Version alignment reducibility). *Consider the same settings as Corollary 3. We write the update of I_1 as $\Delta(I_1) = \cup[I_{1.1}, \dots I_{1.k}]$ and similar for I_2. Assuming $\mathbf{R}_1 \cap \mathbf{R}_2 = \emptyset$, the alignment is version reducible, satisfying $ts(\Delta(I_1) \bowtie_T \Delta(I_2)) = ts(\cup[I_{1.1} \bowtie_T (\cup[I_{2.1}, \dots, I_{2.\ell}]), (\cup[I_{1.2}, \dots]) \bowtie_T I_{2.\ell+1}, \dots])$, where each component aligns two (merged) branches with overlapped time ranges from $\langle I_1, I_2\rangle$. In addition, every two components are not overlapped on timestamps, like $I_{1.1}$ and $\cup[I_{1.2}, \dots]$.*

The reducibility of alignment is also conditional, which relies on the time ranges of each branch.

Example 9 (Continue Example 8). *Let I_1 denote the branch merge result $\cup[I_{b'}, I_b, I_{main}]$ and I_2 be the result of $\cup[I_{2.1}, I_{2.main}]$. Without the time range of branches, the push-down of alignment requires a pair-wise execution before branch merging. That is, $I_1 \bowtie_T I_2 = \cup[I_{b'} \bowtie_T I_{2.1}, I_{b'} \bowtie_T I_{2.main}, \dots]$. Each pair-wise alignment $I_{LHS} \bowtie_T I_{RHS}$ requires $O(|I_{LHS}| + |I_{RHS}|)$ based on sorted instances. Thus, the pair-wise alignment consumes even more than the original plan. Instead, given the instance time ranges, we only execute the pairs with overlapped ranges. Since alignment is distributive w.r.t. branch merging, we merge branches that share the same operating variable, by Corollary 5, $\cup[I_{b'} \bowtie_T I_{2.1}, I_b \bowtie_T I_{2.1}] = \cup[\cup[I_{b'}, I_b] \bowtie_T I_{2.1}]$.*

Proposition 6 (Alignment optimizing analysis). *Under the same settings as Corollary 5, let T_{align} denote the time for aligning two blocks. Also, α_{θ_1} and α_{θ_2} represents the selectivity of time series I_1 and I_2 in these data blocks. Only applying the version reducibility optimization rule of alignment saves the CPU consumption,*

$$O(\alpha_T B(2 - \alpha_{\theta_1} - \alpha_{\theta_2})(T_{merge} - \alpha_\Delta T_{align}) + B(\alpha_{\theta_1} + \alpha_{\theta_2} - 1)T_{merge} - \alpha_\Delta B(|\mathbf{R}_1| + |\mathbf{R}_2|)T_{filter}).$$

Without pushing down rules, we merge the updates and find series tuples of attributes $\mathbf{R}_1 \cup \mathbf{R}_2$. This process costs $O(B \cdot T_{merge} + (1 - \alpha_\Delta)B(|\mathbf{R}_1| + |\mathbf{R}_2|)T_{filter})$ because $\mathbf{R}_1 \cap \mathbf{R}_2 \neq \emptyset$. For I_1, we have $(1 - \alpha_\Delta)B(1 - \alpha_{\theta_1})$ blocks to align. It is similar for I_2. The CPU usage is the sum of update merging, tuple finding, and aligning, where alignment costs $O(((1 - \alpha_\Delta)B(1 - \alpha_{\theta_1}) + (1 - \alpha_\Delta)B(1 - \alpha_{\theta_2}))T_{align})$.

When pushing down, we merge $(1 - \alpha_T)B$ blocks. For non-overlapping components and merged blocks, we filter them and find the alignment of two instances. The former works on $(1-\alpha_T)B$ blocks, while the latter runs on $(1-\alpha_\Delta)(1-\alpha_T)B$ blocks. Each part consumes similarly to alignment analysis without pushing down, and the difference between the two executions is shown in Proposition 6.

3.6 Extension to Multi-variate Time Series

In industries, PLCs may connect to more than one sensor, leading to multiple synchronized sensor readings. IoT databases accept data packets with each tuple composed of multiple sensor readings. When repairing, we modify some attributes and insert a single-variate update into a branch for each attribute. For example, instance I of schema $\{T, A, B\}$ stores multivariate tuples in the main branch and opens a branch $I_{1.A}$ for updates of attribute A. When querying, the branches of the same priority are placed and merged such as $\cup[I_{1.A}, I_{1.B}, I_{1.C}, I_{main}]$. Querying over multivariate time series shows potential optimizations based on projection.

When a query relates to all attributes in the schema, we apply optimization rules in Section 3.3-3.5. For example, we optimize a time range query by Corollary 1. We have,

$$\sigma_\theta(\Delta(I)) = \cup[\sigma_\theta(I_{1.A}), \sigma_\theta(I_{1.B}), \sigma_\theta(I_{1.C}), \sigma_\theta(I_{main})].$$

Otherwise, executing projections ahead of updates is beneficial. Corollary 1 concludes the version reducibility of projections. For example, when θ only applies to A, expression $\pi_A\sigma_{A>a}(\Delta(I))$ is equal to $\sigma_{A>a}(\cup[I_{1.A}, \pi_A(I_{main})])$. We show the benefits of updating by branches and pushing down projections.

Proposition 7. *Let $\mathbf{R}' \subseteq \mathbf{R}$ denote the queried attributes as the subset of attributes $\mathbf{R}$. Assume that there are B blocks of data for instance I. Pushing down the projection $\pi_{\mathbf{R}'}$ ahead of updates will reduce the disk I/O of $O(B \cdot \frac{\sum_{A\in\mathbf{R}\setminus\mathbf{R}'} \alpha_{\Delta.A}}{1-\sum_{A\in\mathbf{R}'} \alpha_{\Delta.A}})$ and the CPU consumption of $O(BT_{merge} \cdot \frac{\sum_{A\in\mathbf{R}\setminus\mathbf{R}'} \alpha_{\Delta.A}}{1-\sum_{A\in\mathbf{R}'} \alpha_{\Delta.A}})$, where $\alpha_{\Delta.A}$ is the ratio of updates for attribute A, and T_{merge} is the cost of merging two data blocks.*

Compared to saving the whole tuple on each update, opening branches for each attribute could reduce the disk I/O. Based on the update ratio, the updated data will occupy $B \cdot (1 - \sum_{A\in\mathbf{R}} \alpha_{\Delta.A})$.

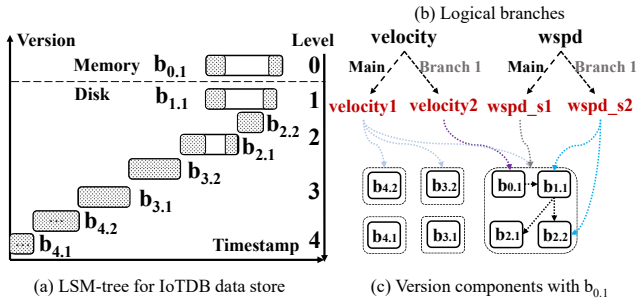


Fig. 5. Combining LSM-tree with branches.

Thus, when we load updates on attributes $\mathbf{R}'$, we need a disk I/O of $O(B \cdot \frac{1-\sum_{A\in\mathbf{R}} \alpha_{\Lambda.A}}{1-\sum_{A\in\mathbf{R}'} \alpha_{\Lambda.A}})$. Similarly, since we only need to merge these blocks, we have reduced CPU usage as concluded above.

4 SYSTEM IMPLEMENTATION

Based on the Apache IoTDB, we show implementation details of **IoTDB-v**, a versioned time-series database. (1) For language, based on an outer join of branches, we integrate branch priority into a data analysis query. (2) For combining branches with LSM, we record the reference linkages from a version branch to data blocks in the LSM tree, shown in Figure 5. (3) For data storage, we analyze the effects of two factors on using HDD and SSD, including LSM space amplification and branch-supported parallel random access. (4) For concurrent queries, we discuss sharing updated subqueries among different workloads. (5) For distributed processing, we show workload separation based on branches and the LSM tree.

4.1 Version Query Language Extensions

IoTDB-v supports a view of aligning branches under the same node by all timestamps (outer join). For example, in Figure 3(b), we assume that $I_{main} = \{\langle t_1, a_1\rangle, \langle t_2, a_2\rangle\}$ and $I_b = \{\langle t_2, a_3\rangle\}$. When using *wspd* in the FROM-clause of SQL, each returned tuple has a schema of $\{T, A_{main}, A_b\}$ and the returned tuple set consists of $\langle t_1, a_1, \text{null}\rangle$ and $\langle t_2, a_2, a_3\rangle$. A data analysis query could easily integrate the priority selection of branches to manage versions, such as the time range query (**Q1**),

```
SELECT coalesce(b, main) FROM wspd WHERE T > t1 AND T < t2;
```

updates data in branch *main* by valid data points of the same timestamp in branch *b*. Similarly, a downsampling query (**Q4**) could integrate the branch selection,

```
SELECT AVG(A)
FROM (SELECT coalesce(b, main) A FROM wspd)
GROUP BY SLIDING WINDOW 1s;
```

It updates according to data repairs and aggregates tuples by a 1-second sliding window.

4.2 Combining LSM-tree with Branches

Figure 5 demonstrates the relationship among the LSM storage tree, version branches, and version components. In Figure 5(a), the IoTDB adopts an LSM tree to manage all inserted data and uses labels to identify each run like $b_{0.1}$. Optimization rules such as value filtering in Corollary 3 of Section 3.4 merge the blocks that are overlapping on time ranges before applying value filters. Figure 5(b) contains version branches of *wspd* and *velocity* that are created to manage updates. Figure 5(c) uses a graph of components to organize the non-overlapping components, each highlighted by the

dashed line squares and containing at least one data block label. Every pair of components does not share common time ranges. In a component, we connect the blocks by arrows (direct edges) to denote updates. For example, when adding $b_{0.1}$, we notice that $b_{0.1}$ overlaps and updates $b_{1.1}$. We find the component of $b_{1.1}$ and add an edge from $b_{0.1}$ to $b_{1.1}$.

When we add an update branch, we use logical branches to track updates and index data. For example, when repairing *wspd*, we could add a tuple named *wspd_s2* and Branch 1 will add a pointer to the related block $b_{1.1}$ or the whole component. The LSM tree compaction only works on the prefix of record names, e.g., *wspd*. Thus, branching will not affect LSM in writing and compacting files. Since the LSM tree of IoTDB has fewer levels and compaction always results in blocks like $b_{4.1}$, managing the pointer will not lead to high overhead to the system.

When executing a plan like $\cup([\sigma_{A<2}(I_{b2}), \sigma_{A<2}(\cup[I_{b1}, I_{main}])])$ for value filters, we first find the blocks of each branch and rewrite the plan with labels. Consider I_{b1} as *wspd_s2*. $\sigma_{A<2}(\cup[I_{b1}, I_{main}])$ is equivalent to $\sigma_{A<2}(\cup[b_{0.1}, \cup[b_{1.1}, b_{2.2}]_{s2,s1}, b_{2.1}])$. It means that when merging $b_{1.1}, b_{2.2}$, we find records of *wspd.A* and use records with suffix *s2* to replace *s1* on the same timestamp.

4.3 HDD vs. SSD Implementation

As mentioned in [32], block leveling in LSM-tree, adopted by *IoTDB* and **IoTDB-v**, leads to large space amplification. Thus, it is not friendly to the more expensive solid-state storage devices (SSDs) and Flash drives, compared to hard disk drives (HDDs). In addition, while SSDs support efficient random and parallel reads, the database based on an LSM tree could waste I/O bandwidths due to its sequential reads and writes [32, 47]. Thus, block leveling is crucial for SSDs and Flash Drives. Fortunately, our proposals could use better the SSD attribute of inner parallelism. Consider a value range filtering query by branch merging, $\sigma_\theta(\cup[I_{1.1}, I_{2.1}, I_{3.2}, I_{4.2}])$. Existing systems like IoTDB [46] find these runs to read and merge before applying the value filter, involving sequential reads of all runs. In **IoTDB-v**, version reducibility allows the filter to push down ahead of branch merge (Corollary 3) based on non-overlapping components on time ranges. Since non-overlapping components will not update each other, we execute the subqueries like $\sigma_\theta(I_{3.2})$ independently by different threads. For the LSM data store, processing by component would increase the reading parallelism of SSDs according to the number of components.

4.4 Concurrent Version Query Sharing

IoTDB also works concurrently to answer queries. For example, downsampling queries

$$e_1 : G_{T:\text{avg}(wspd)}(I), e_2 : G_{T:\text{avg}(velocity)}(I)$$

apply to different attributes of instance I. In addition, different queries may ask for (1) related updated range queries like $e_3 : \sigma_{T>23:00}(\Delta(I))$, $e_4 : \sigma_{T>23:00 \wedge T<23:59}(\Delta(I))$, (2) the same instance update such as $e_5 : \Delta(I_1) \bowtie_T \Delta(I_2), e_6 : \Delta(I_1) \bowtie_T \Delta(I_3)$, and (3) common updates, e.g., $e_7 : \cup[I_{1.b0}, I_{1.b1}, I_{1.main}]$, $e_8 : \cup[I_{1.b0}, I_{1.main}]$.

We notice that pushing down operators ahead of the update operator may cost more due to eliminating common expressions. For example, when executing alignment ahead of updates, we can not share instance update $\Delta(I_1)$ between e_5 and e_6. Existing approaches focus on sharing identical subqueries [25, 36] that rely on updated tables in version databases. Similar to the introduction of optimization rules, we discuss the subquery sharing of each operator with version updates and analyze the CPU usage and disk I/O.

We first show how to share based on common updates. Consider expressions e_7 and e_8. e_7 additionally include branch $I_{1.b1}$ as the secondary choice prior to main branch. When $I_{1.b1}$ is a component, i.e., not overlapping with other branches, e_8 is a subquery of e_7 such that $e_7 = \cup[e_8, I_{1.b1}]$. Otherwise, the execution result of e_8 is not shared. To conclude, version updates by branch can

Table 3. Relational reducibility of version query operators.

Ver. Ops	Relational expression (in relational algebra)
$\Delta^{v=b}(I)$	$G_{T:\text{coal}(b_i, main)}(I_{main} \rtimes_T I_b)$
$\sigma_\theta(I)$	$\sigma_\theta(I)$
$\pi_{\mathbf{R}'}(I)$	$\pi_{\mathbf{R}'}(I)$
$\cup[I_1 \dots I_m]$	$G_{T:\text{coal}(I_1.A,\dots,I_m.A)\dots}((I_1 \rtimes\!\ltimes_T I_2) \cdots \rtimes\!\ltimes_T I_m)$
$I_1 \bowtie_T I_2$	$I_1 \bowtie_T I_2$
$\rho_{T',\mathbf{R}'}$	$\rho_{T',\mathbf{R}'}$
$G_{T_g:f}(I)$	$G_{T_g:f(\text{coal}(b_i, main))}(I_{main} \rtimes_{T_g} I_b)$

share a common sub-list of components and other components not shared should be placed in order. For example, given $e_i : \cup[\cup[I_{k.1}, I_{k.2}, I_{k.3}], I_{k.4}]$ and $e_j : \cup[I_{k.2}, I_{k.3}]$ merged by components, we share e_j such that $e_i : \cup[\cup[I_{k.1}, e_j], I_{k.4}]$. The benefits depend on the shared subexpression, e.g., saving the disk I/O of $O(B_2 + B_3)$ and CPU usage of $O((B_2 + B_3)T_{merge})$ for the above example.

For time or value ranges, the subsumption relationship between two conditions decides whether a query could be shared as a subquery of another. For example, e_3 is a subquery of e_4 by existing studies [40], with $T > 23{:}00 \wedge T < 23{:}59$ always satisfying $T > 23{:}00$. Based on version control, when $T > 23{:}59$ corresponds to new components, e_4 is a subquery of e_3 to share the update of I within $(23{:}00, 23{:}59)$, different from existing approaches. The benefits also come from shared expressions without additional overhead, e.g., the disk I/O and CPU usage of executing e_4.

Since aggregation relies on updated input, we move different functions on the same instance together and execute them by updating once, e.g., $G_{T:\text{avg}(wspd),\text{avg}(velocity)}(I)$ for e_1 and e_2. Finally, when expressions involve aligning the same updated instance, we compare the reduction from pushing down alignment (Proposition 6) and sharing common expressions. For example, sharing I_1 leads to a disk I/O reduction of $O(B_1)$ and the total CPU usage is $O((B_1 + B_2 + B_3)(T_{merge} + (1 - \alpha_\Delta)T_{align}))$. Based on the estimated selectivity of alignment, we choose the plan with fewer costs.

4.5 Distributed Version Query Processing

IoT databases installed in gateways and cloud servers form a network of distributed nodes. The task of an expression such as a time range filter could be separated into multiple tasks by distributed nodes. This section assumes IoT data are synchronized and discusses the component-based task separation. The component-based separation considers each version update $\Delta(I)$ as a branch merging list $\cup[I_{1.1}, \cup[I_{1.2}, I_{1.3}], \dots]$, consisting of non-overlapping components on time like $I_{1.1}$. Based on version reducibility, the operators can execute ahead of branch merging. After pushing down operators, we separate the task for each node based on the expression in the merging list. For example, Example 8 shows a reduced result of $\sigma_{A<2}(\Delta(I))$ by $\cup([\sigma_{A<2}(I_{b'}), \sigma_{A<2}(\cup[I_b, I_{main}])])$. We map each element like $\sigma_{A<2}(I_{b'})$ to a distributed node. Then, we concatenate fetched results by time order as the result.

5 TRANSLATION TO RELATIONAL QUERY

Existing relational databases like PostgreSQL [38] have made themselves portable and efficient for IoT data. Managing time series directly by PostgreSQL saves the effort to maintain and optimize a new database system. This section shows the reducibility of translating version queries to relational SQL for relational databases.

5.1 Relational Reducibility

Definition 4 (relational reducibility). *Let e denote an expression in versioned time-series algebra. The expression e is relational reducible to an expression e^{RA} by relational operators if and only if, $ts(Q(\Delta^{\mathbf{v}}(D), e)) = ts(Q(D, e^{\mathsf{RA}}))$.*

The relational reducibility finds a derived relational expression e^{RA} from a version query expression e under the constraint that $Q(\Delta(D), e) = Q(D, e^{\mathsf{RA}})$. We summarize the reduction rules in Table 3 for an equivalent relational expression.

Theorem 8. *Version query expression e can be translated into a relational expression e^{RA} by Table 3. The expressions e and e^{RA} satisfy relational reducibility.*

Example 10. *Consider a time range query **Q1** and alignment query **Q3** from Example 5. **Q1** finds an updated time-series slice by range filter $T > t_1 \wedge T < t_2$. **Q3** aligns two updated instances according to equal timestamps. We have the version expression for **Q1**, $e_{\boldsymbol{Q1}} = \sigma_{T>t_1 \wedge T<t_2}(\Delta(I))$, and **Q3**, $e_{\boldsymbol{Q3}} = \Delta(I_1) \bowtie_T \Delta(I_2)$. Assuming that we need to merge only one branch named b for each version instance and instance I is a single-variate time series. Refer to Table 3, we have equivalent relational expressions,*

$$e^{\mathsf{RA}}_{\boldsymbol{Q1}} = \sigma_{T>t_1 \wedge T<t_2}(G_{T:\mathrm{coal}(b,main)}(I_{main} \rtimes_T I_b)), \text{ and}$$

$$e^{\mathsf{RA}}_{\boldsymbol{Q3}} = G_{T:A=\mathrm{coal}(b,main)}(I_{1.m} \rtimes_T I_{1.b}) \bowtie_T G_{T:B=\mathrm{coal}(b,main)}(I_{2.m} \rtimes_T I_{2.b}).$$

Here, the left outer join refers to timestamps in I_{main} and updates the value in the main branch by branch b. The left join keeps all the original tuples in the main instance I_{main} and aligns the updates in I_b, where the tuples in main without updates have null *values in the result column b. The function* coal(A, B) *stands for coalesce, which takes A when A is not* null *and otherwise uses B. The simple pattern-based replacement expresses and executes version data analysis in a relational database, such as PostgreSQL.*

5.2 Relational Integration of Version Controls

Based on relational reducibility, we integrate version control into PostgreSQL as *Postgres* and **Postgres-v**, in Table 4, which summarizes the integration of version and relational reducibility. For SQL-based baseline *Postgres*, we translate the relational expression satisfying relational reducibility to SQL as the input query. For SQL-based proposal **Postgres-v** with optimization rules, we successively apply version reducibility to the version query expression, relational reducibility to version reduced expression, and translation to relational expression to get the SQL.

Example 11. *First, we illustrate the translation from a relational reduced expression to benchmark SQL for baseline Postgres. Consider query **Q1** with converted relational expression $e^{\mathsf{RA}}_{\mathbf{Q1}}$. We write the subquery $G_{T:\mathrm{coal}(b,main)}(I_{main} \rtimes_T I_b)$ for updated instance*

```
SELECT coalesce(b, main)
FROM I.main LEFT JOIN I.b ON I.main.T = I.b.T;
```

*The SQL of version time range filter (**Q1**) refers to the subquery of updated instance as,*

```
WITH R AS
  (SELECT T, coalesce(b, main) A
   FROM I.main LEFT JOIN I.b ON I.main.T = I.b.T)
SELECT * FROM R WHERE T > t1 AND T < t2;
```

In addition, based on version reducibility, version expression $e_{\boldsymbol{Q1}}$ is reduced into $\Delta(\sigma_{T>t_1 \wedge T<t_2}(I))$. The version-reduced expression could also be converted to a relational expression, with

$$G_{T:\mathrm{coal}(b,main)}(\sigma_{T>t_1 \wedge T<t_2}(I_{main}) \rtimes_T \sigma_{T>t_1 \wedge T<t_2}(I_b)).$$

Table 4. Integrated techniques of proposals and baselines

System	Category	Version red.	Relation red.	Branching
IoTDB [46]	TSDBMS	Not enabled	Not needed	Supported
IoTDB-v	TSDBMS(V)	All rules	Not needed	Supported
InfluxDB [4]	TSDBMS	Corollary 1	Not needed	Supported
Postgres [38]	RDBMS	None	Applied	New relation
Postgres-v	RDBMS(V)	All rules	Applied	New relation
TardisDB [41]	RDBMS(V)	None	Not needed	Supported

Table 5. Dataset information

Name	Data Type	#Attribute	#Points
Bitcoin (Open) [3]	Float	7	21M
Climate (IoT)	Float	4	34M
Ship (IoT)	Float	210	162M
White noise (Gen)	Float	4	1B

Similarly, we translate it to SQL by referring to the pushed-down expressions as subqueries.

```
WITH R1 AS (SELECT * FROM I.main
            WHERE I.main.T > t1 AND I.main.T < t2)
     R2 AS (SELECT * FROM I.b
            WHERE I.b.T    > t1 AND I.b.T    < t2)
SELECT T, coalesce(b, main)
      FROM R1 LEFT JOIN R2 ON R1.T = R2.T;
```

*After converting version query expressions by version and relational reducibility, we can also execute the optimized version queries as proposal **Postgres-v** in relational databases.*

6 EXPERIMENTS

The experiments study the effectiveness of the proposed optimization rules in our version control query engine for IoT applications.

6.1 Settings and Applied Queries

6.1.1 Datasets. Table 5 shows the dataset statistics. The datasets are mainly collected from the real world, and white noise is used to increase data scales. To simulate the arrival of IoT data, we insert single-variate tuples and repair a 10% amount of data points as default. We also generate 5% delayed and duplicated points according to real-world statistics [21]. Figure 16 of Section 6.4.5 applies a white noise data stream to benchmark the performance of writing version data on varied data scales. The stream generates and inserts tuples into compared databases with a speed based on their ability to persist arrival data [9] with 100 million tuples on IoTDB.

6.1.2 Benchmark Queries. We use a set of queries based on the needs of real-world IoT applications to benchmark the version query performance of databases. We already introduced them in

previous examples and sections, such as time range query (**Q1**), value range query (**Q2**), alignment (**Q3**), and downsampling (**Q4**). We also include queries of more operators, including downsampling on alignment $G_{T:\text{avg}(A)}(\Delta(I_1) \bowtie_T \Delta(I_2))$ (**Q5**), and downsampling on merged branches $G_{T:\text{avg}(A)}(\cup[I_{b1}, I_{main}])$ (**Q6**). In addition, since the equality of series values is also important to IoT data analysis, we include **Q7** as an alignment on both timestamps and series values within a time range query $\sigma_{T>t_1 \wedge T<t_2}(\Delta(I_1) \bowtie_{T,V} \Delta(I_2))$. Also, we align all branches to see historical data $I_{main} \bowtie_T I_{b1} \bowtie_T \cup[I_{b1}, I_{main}]$ by **Q8**. Based on Example 10 and 11, we translate and optimize these queries into SQL for relational databases (RDBMS).

6.1.3 Integrated Systems and Baselines. In Table 4, we summarize the integrated techniques of the proposed systems, including **IoTDB-v** and **Postgres-v**, both of which adopt version reducibility to delay time-consuming updates. We compare them with the baselines, i.e., IoTDB [2], Postgres [38], InfluxDB [4], and TardisDB [41].

6.1.4 Runtime Characteristics. We use a machine with a 3TB 7200 RPM HDD, a 1TB NVMe SSD, 48GB DDR4 memory, 2.90GHz Intel i7 cores with 16 physical threads, and 16 MB L3 cache, running on Windows NTFS file system. The database is configured by default with allocated memory of 2GB, a page size of 65536 bytes, a buffer of 2MB, a disk block size of 512 bytes, and sorted runs of size 8MB.

6.2 Query Performance on Version Datasets

This section compares the proposals and baselines over real-world and large-scale datasets. Section 6.2.1 reports the performance under datasets and Section 6.2.2 shows the scalability of different systems on white noise dataset with 1 billion data points in total.

6.2.1 Version Query on Various Datasets. Figure 6 compares our proposed systems **IoTDB-v** and **Postgres-v** to other baselines on various datasets. For *IoTDB* and **IoTDB-v**, we find **IoTDB-v** could reduce the execution time significantly on selective operators such as range query and alignment. Unlike TardisDB, **IoTDB-v** can apply a value range filter on some blocks directly without updates. The difference is mainly in query optimization over the version control storage. IoTDB integrates data analysis with version control and optimizes operator execution order with fewer costs, while TardisDB merges branches before data analysis.Intuitively, storing all records for each version in a relation can serve as a version control database. However, it is not affordable. Instead, the baselines are more advanced to store updates only in LSM-tree or new relations as branches, the same as a version control database.

For version queries and optimizations, both IoTDB and TardisDB integrate version control by branches to store tuple updates. The difference is mainly in query optimization over the version control storage. IoTDB integrates data analysis with version control and optimizes operator execution order with fewer costs, while TardisDB simply merges branches before data analysis. Note that *Postgres* and **Postgres-v** perform better on alignments (**Q3** and **Q8**) and are more scalable on the Ship dataset, having more time series. This is because PostgreSQL adopts B-tree to find related data points efficiently, while **IoTDB-v** requires reading and merging related blocks, which is time-consuming. InfluxDB optimizes IoT queries better on range queries (**Q1** and **Q2**) by pushing down time filters, but requires more resources on other queries.For **Q7**, the join on values is more costly than that on timestamps (**Q3**), because of the need to check each pair of records rather than relying on the order of timestamps in each LSM-tree run.

6.2.2 Version Query on Various Scales. We report scalability experiments on the Noise dataset sized 1 billion points in Figure 7. Compared to *IoTDB*, **IoTDB-v** optimizes query plans by rules, which show clear saving time as scale increases such as Proposition 2. *Postgres* and **Postgres-v** use a

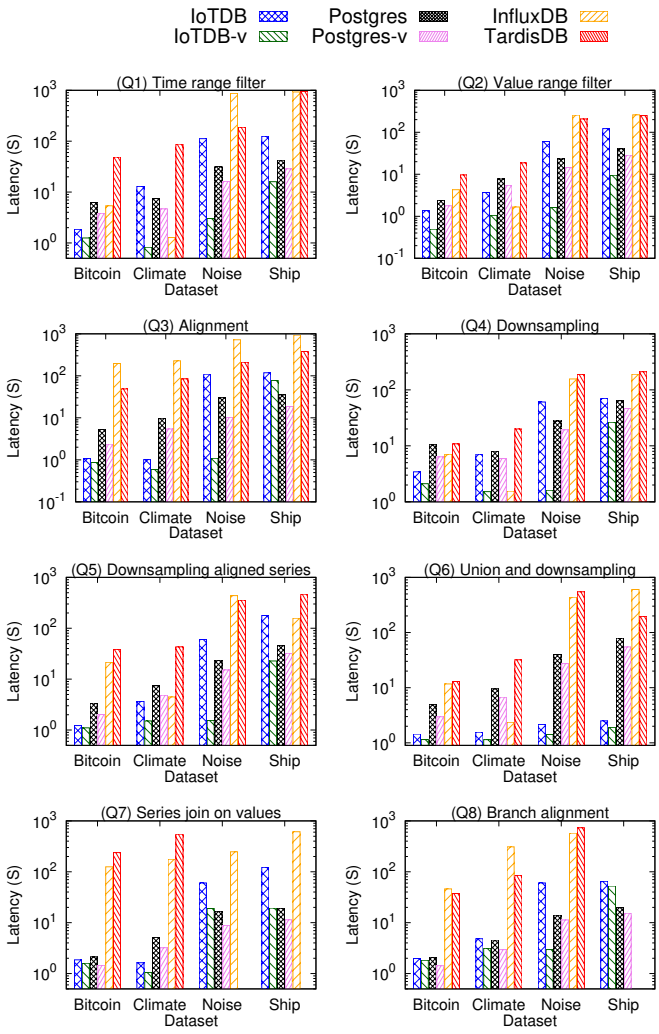


Fig. 6. Performance of queries over various datasets. InfluxDB and TardisDB exceed 10^3 seconds and thus are omitted.

B-tree to locate data, which is efficient for querying over large scales. *IoTDB* needs to search related data from its data stores, which is the drawback of using an LSM tree. InfluxDB and TardisDB execute range filters efficiently but meet hardness on large data scales for the same reason as analyzed in Section 6.2.1. Regarding InfluxDB, consider alignment (**Q3**). InfluxDB does not apply version reducibility to optimize alignment, leading to the overhead of reading and buffering data points with tags in the memory to update, which is time-consuming.

6.3 Performance of Version Optimizations

6.3.1 Performance of Concurrent Queries. Section 4.4 displays our motivations and techniques to execute version queries concurrently in IoTDB. Figure 8 validates our proposals by comparing the concurrent effectiveness of **IoTDB-v** to other baselines on each query. Here, **Postgres-v** uses

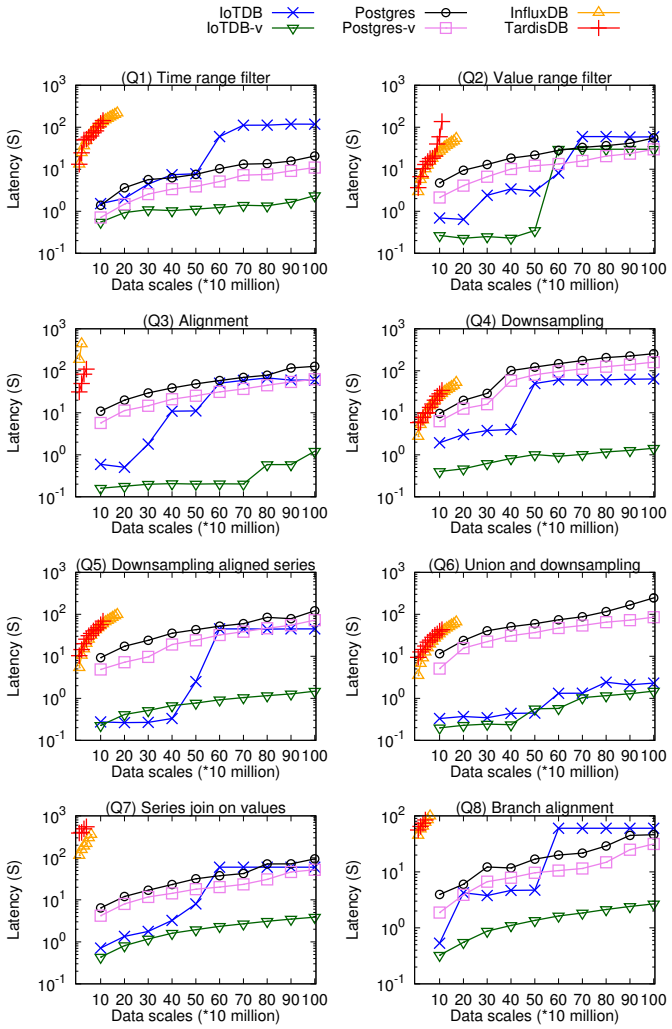


Fig. 7. Performance of queries (Q1-Q8) on varied data scales with up to 1 Billion tuples.

reducibility to optimize but relies on PostgreSQL to execute concurrently. We notice that updated subquery sharing leads to better average latency, as the lines of **IoTDB-v** decrease over concurrency. In addition, **Postgres-v** also shows improvement over *Postgres*. This is because the complex pattern by outer join and aggregation in the SQL for *Postgres* is hard to share by optimization rules in existing databases.

6.3.2 Performance of Distributed Queries. Based on our proposed technique in Section 4.5, we benchmark and compare the performance of various distribution nodes. Since the distribution task separation rule is based on components for each operator, we study the distribution acceleration of each query. We compare the IoTDB to open-source distributed *PostgreSQL* [10] in Figure 9, where the distributed release of InfluxDB is currently commercial and TardisDB does not support distributed nodes through networks. We notice that PostgreSQL could improve its performance

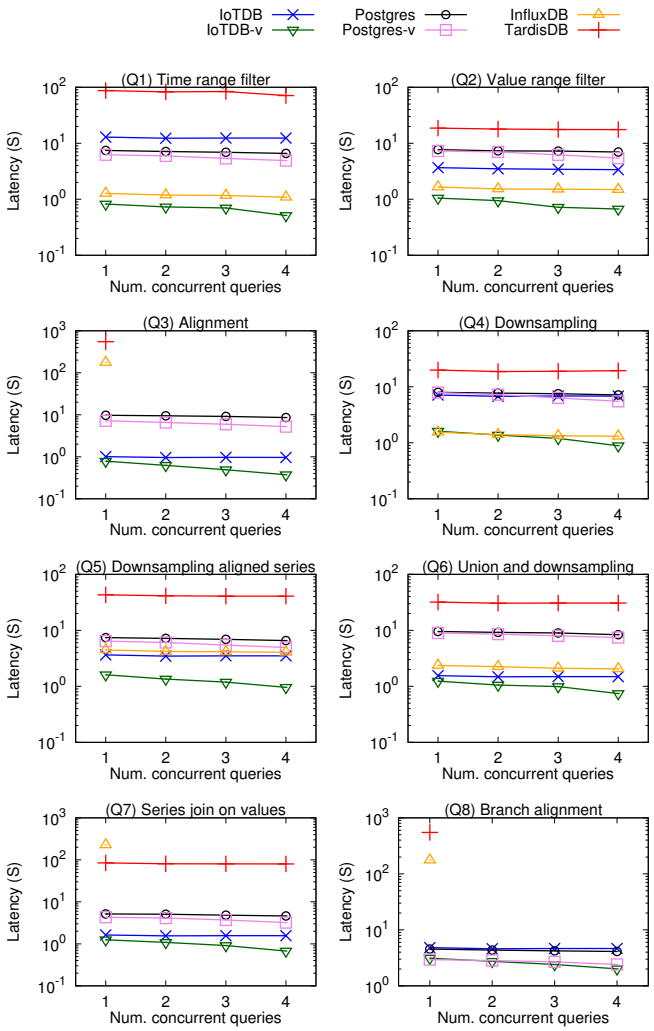


Fig. 8. Averaged CPU time of concurrent version queries. TardisDB and InfluxDB exceeding 10^3 seconds are omitted.

better on more nodes. Our approaches benefit more from optimization rules than more nodes because it is restricted by large components, which decide total execution time.

6.3.3 Performance of Storage Devices. Section 4.3 introduces the application of our scheme onto the storage devices. Figure 10(a) compares data store sizes in each database to the size of the plain data file. It evaluates the space amplification of compared systems to see the high overhead brought by an LSM tree. Figure 10(b) evaluates the performance gain by our component-based processing scheme, which shows an improvement in latency by changing the data store from HDDs to SDDs.

6.3.4 Multivariate Time Series Optimization Performance. Section 3.6 discusses version update management of multivariate time series with dependent attributes, which concludes the benefits of

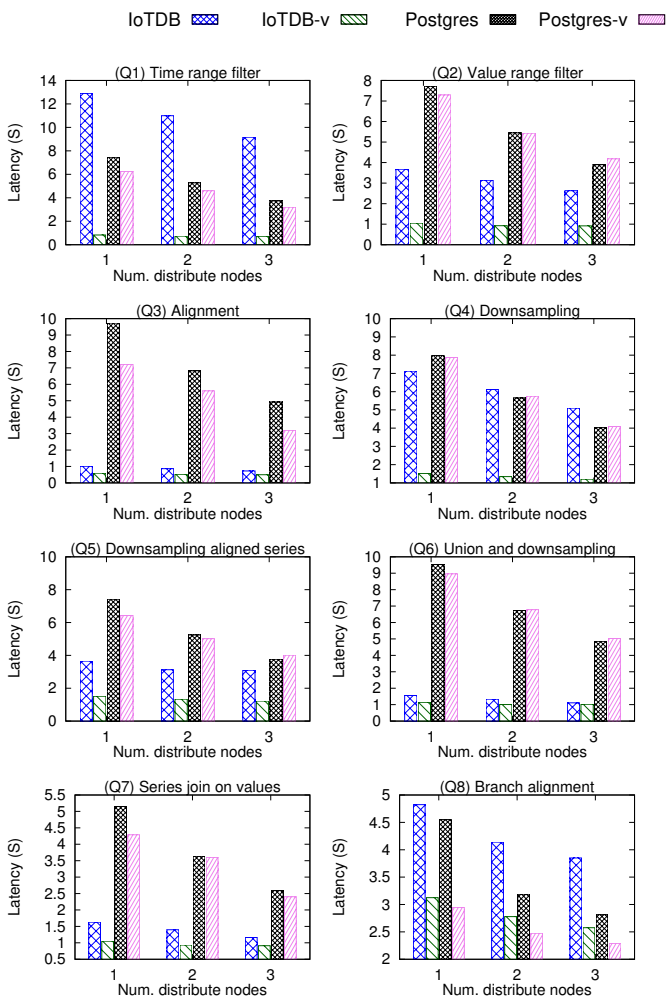


Fig. 9. Performance of distributed version query (TardisDB does not support, InfluxDB is commercial).

updating only the queried attributes. We use a group of branches for all attributes and evaluate different querying attributes in Figure 11. We observe that the time costs of our proposals **IoTDB-v** and **Postgres-v** grow almost proportionally to the queried attributes, confirming Proposition 7. The baselines *IoTDB* and *Postgres* still need to update the entire time series at constant costs due to dependent data collection. Similar to Section 6.2, due to time constraints, InfluxDB and TardisDB are not reported on the Ship dataset.

6.4 Parameter Studies

6.4.1 Effects of Varied Series Number. Figure 6 reports the effects by changing the number of aligned instances in Q3 (alignment) and merged instances in Q6 (union). We report the performance in Figure 12 for joining and merging more than two instances. It shows our proposals outperform stably when aligning and merging more time series.

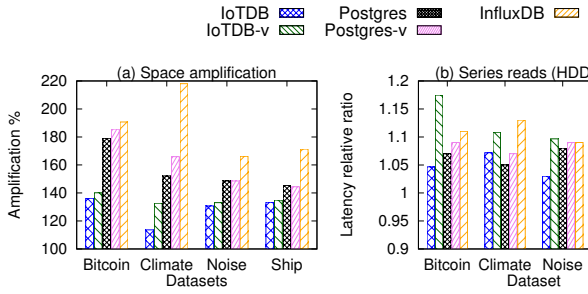


Fig. 10. Storage overhead and efficiency improvement of SSD to HDD. Memory-resident TardisDB is omitted.

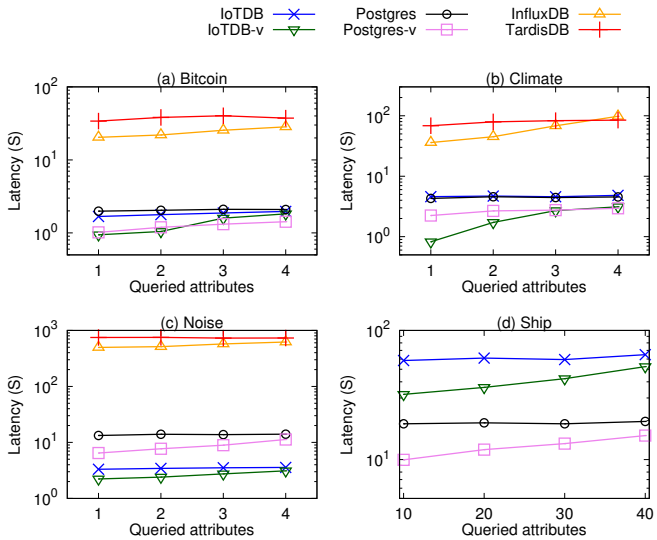


Fig. 11. Multivariate alignment (Q3) performance. TardisDB and InfluxDB exceeding 10^3 seconds are omitted.

6.4.2 Effects of Merged Branches. To verify the efficiency of applying branch merging optimization rules, in Propositions 4 for value range filter and 6 for alignment, we report the performance on different branches. The experiment shows the scalability of proposed systems by merging large branches, by separating them into components.

6.4.3 Effects of Varied Sampling Frequencies. Since aligning and filtering time series are sensitive to related data blocks, queries read data evenly from data blocks when using different sampling ratios. Thus, varied frequency controls disk I/O for the databases on the LSM tree, while changing the selectivity of related tuples from each block. In Figure 14, we report the performance of Q1-Q3 on various frequencies. It proves the effectiveness of version reducibility and validates the cost estimations in Propositions 2, 4, and 6 on CPU usage estimations. Moreover, a larger data block leads to reads of unnecessary data, while smaller blocks require more block seeking in disks. Thereby, we evaluate the performance under varied block sizes in Figure 14, which proves the existence

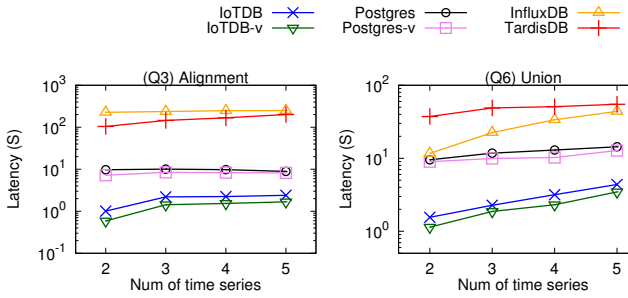


Fig. 12. Performance of query optimization on various numbers of time series in union and join queries.

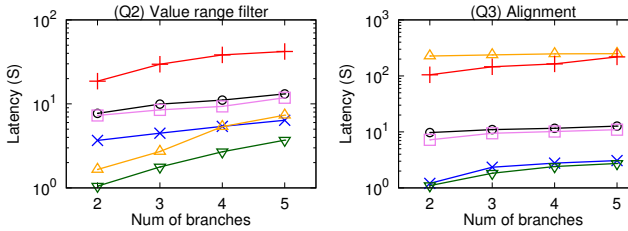


Fig. 13. Performance of query optimization on various numbers of merged branches.

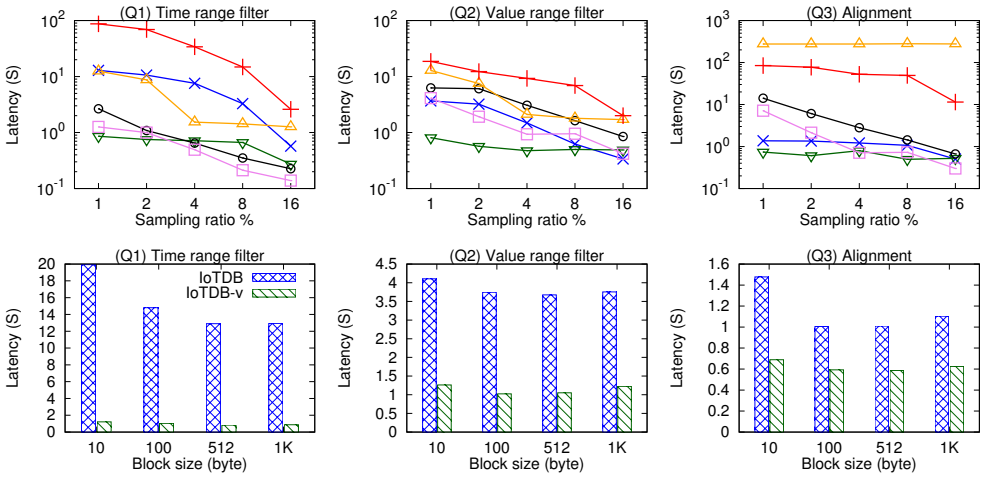


Fig. 14. Query optimization performance on varied data sampling frequency and block size.

of a better block size. The results of Q1 and Q2 are generally similar to Q3 of alignment because **IoTDB-v** needs similar effort to read related data blocks.

6.4.4 Effects of Varied Selectivity. Because Propositions 2 and 4 show that selectivity affects the reduced disk I/O and CPU usage on filtering, we validate our optimization rules by varying the selectivity of filters in Figure 15. It examines that the reduced querying latency is nearly proportional to the decreased selectivity.

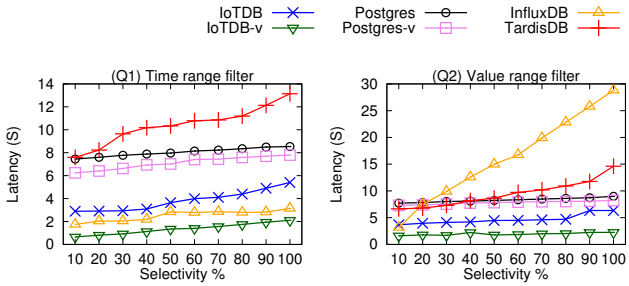


Fig. 15. Performance of filtering on varied data selectivity.

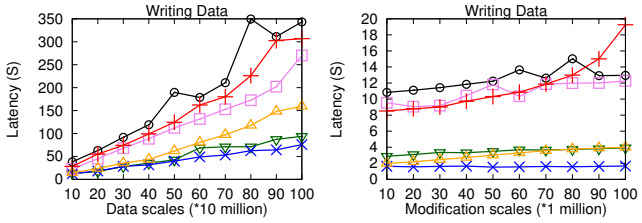


Fig. 16. Performance of writing data on varied data scales and modification rates.

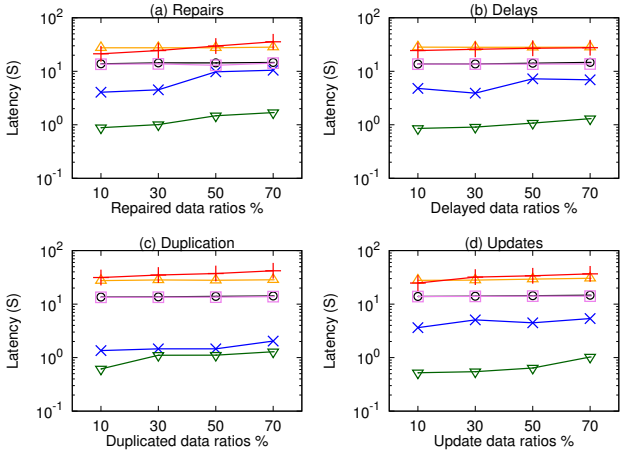


Fig. 17. Time range query (Q1) on various update types.

6.4.5 Writing Efficiency. Databases store data versions differently, leading to varied insertion and update costs. For example, IoTDB/**IoTDB-v** and InfluxDB use LSM-tree, while **Postgres-v** and Postgres use B-tree with buffers. We use Figure 16 to evaluate the speed of persisting arrival data points. It proves that databases based on LSM-tree have better writing performance of IoT data.

6.4.6 Effects of Varied Versioning Types. To verify our motivations for delayed, duplicated, and erroneous records, we report the performance under different update ratios in Figure 17. It shows our approach could handle various IoT data types efficiently and effectively.

7 RELATED WORKS

We review our related works on version controls and time-series query language for TSDBs.

Version control database. Version control databases are constructed for different data types, where versioning could use branches with data updates. Using a relational database, Decibel [35] studies versioning and branching on the whole dataset, and TardisDB [41] extends the standard SQL for version control. We also focus on storing updates in branches. We advance existing studies by formalizing versioned time series algebra for a query engine with optimizations. This work proposes version reducibility to optimize queries. Our advances also show in implementing version control and optimizing executable SQL by relational reducibility in relational databases.

Time-series queries. The query language of time series has been studied by the query on arrays [27, 29, 51] and also relates to the query on sequential relations, e.g., SRQL [6, 39] defines its semantics over sorted tables. The query languages for arrays serve effectively for scientific cases that involve matrix operations [29], which are hard to handle version controls. Real usage cases on version controls motivate us to extend the languages further.

Time-series database and query optimization. We focus on the advance of Apache IoTDB [2] by introducing a multi-versioned series. Consider the time-series databases [23] and review the optimization of time-series queries over log files. Existing time-series databases can be classified based on the data models and data storage, e.g., TimescaleDB [5] built over PostgreSQL [38], and InfluxDB over an LSM-alike storage system [4]. Generally, the TSDBs on the top of NoSQL storage contain multi-versioned data [1, 2, 4]. For these databases, given a query, the database will first merge-sort related data, involving file reading from disk and data updates [8, 14, 34], before executing an optimized plan based on relational or sequential algebra. Recent approaches to accelerate LSM query mainly focus on building indexes on data blocks [33, 34, 49], while TSDBs focus on optimizing the LSM structures, e.g., InfluxDB [4]. Again, as Figure 1 shows, merge-sorting is a time-consuming process. Without the support of versioned time-series algebra, it is challenging for existing TSDBs to push down and optimize these processes.

8 CONCLUSIONS

Motivated by IoT data versions in deployed database instances, in this paper, we formalize the version data model along with its semantics and show optimizing chances of versioned query plans. We study consistent query executions of an optimized plan by proposing version reducibility of version algebraic operators and relational reducibility of relational operators to integrate versioned query models into existing systems. As IoT databases, such as the Apache IoTDB, typically adopt an LSM tree to manage persisted data, the implemented system can improve the performance of concurrent and distributed query plans based on LSM indexes and hardware like SSD. The extensive experiments over real datasets show that our proposed optimization rules based on data versions could improve the query performance on real workloads.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (62232005, 62072265, 62021002, 92267203), the National Key Research and Development Plan (2021YFB3300500), and Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song is the corresponding author (https://sxsong.github.io/).

REFERENCES

[1] Apache Cassandra. (Apache Cassandra). https://cassandra.apache.org/
[2] Apache IoTDB. (Apache IoTDB). https://iotdb.apache.org/
[3] Bitcoin Dataset. (Bitcoin Dataset). https://archive.ics.uci.edu/ml/datasets/BitcoinHeistRansomwareAddressDataset

[4] InfluxDB. (InfluxDB). https://www.influxdata.com/

[5] TimescaleDB. (TimescaleDB). https://www.timescale.com/

[6] Heba Aamer, Jan Hidders, Jan Paredaens, and Jan Van den Bussche. 2021. Expressiveness within Sequence Datalog. In *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021*, Leonid Libkin, Reinhard Pichler, and Paolo Guagliardo (Eds.). ACM, 70–81. https://doi.org/10.1145/3452021.3458327

[7] Ahmed Awad, Matthias Weidlich, and Sherif Sakr. 2020. Process Mining over Unordered Event Streams. In *2nd International Conference on Process Mining, ICPM 2020, Padua, Italy, October 4-9, 2020*, Boudewijn F. van Dongen, Marco Montali, and Moe Thandar Wynn (Eds.). IEEE, 81–88. https://doi.org/10.1109/ICPM49681.2020.00022

[8] Haitian Chen, Ziwei Wang, Yunchuan Li, Ruixin Yang, Yan Zhao, Rui Zhou, and Kai Zheng. 2023. Deep Learning-Based Bloom Filter for Efficient Multi-key Membership Testing. *Data Sci. Eng.* 8, 3 (2023), 234–246. https://doi.org/10.1007/S41019-023-00224-9

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[10] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2490–2502. https://doi.org/10.1145/3448016.3457551

[11] Fan Dang, Xikai Sun, Kebin Liu, Yi-Fan Xu, and Yun-Hao Liu. 2023. A Survey on Clock Synchronization in the Industrial Internet. *J. Comput. Sci. Technol.* 38, 1 (2023), 146–165. https://doi.org/10.1007/S11390-023-2908-4

[12] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 79–94. https://doi.org/10.1145/3035918.3064054

[13] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Trans. Database Syst.* 43, 4 (2018), 16:1–16:48. https://doi.org/10.1145/3276980

[14] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 505–520. https://doi.org/10.1145/3183713.3196927

[15] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 365–378. https://doi.org/10.1145/3448016.3457273

[16] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *CoRR* abs/2103.02515 (2021). arXiv:2103.02515 https://arxiv.org/abs/2103.02515

[17] Xiaoou Ding, Hongzhi Wang, Jiaxuan Su, Zijue Li, Jianzhong Li, and Hong Gao. 2019. Cleanits: A Data Cleaning System for Industrial Time Series. *Proc. VLDB Endow.* 12, 12 (2019), 1786–1789. https://doi.org/10.14778/3352063.3352066

[18] Chenguang Fang, Shaoxu Song, and Yinan Mei. 2022. On Repairing Timestamps for Regular Interval Time Series. *Proc. VLDB Endow.* 15, 9 (2022), 1848–1860. https://www.vldb.org/pvldb/vol15/p1848-song.pdf

[19] Like Gao and X. Sean Wang. 2018. Time Series Query. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_428

[20] Philipp M. Grulich, Jonas Traub, Sebastian Breß, Asterios Katsifodimos, Volker Markl, and Tilmann Rabl. 2019. Generating Reproducible Out-of-Order Data Streams. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019*. ACM, 256–257. https://doi.org/10.1145/3328905.3332511

[21] Stefan Herrnleben, Rudy Ailabouni, Johannes Grohmann, Thomas Prantl, Christian Krupitzer, and Samuel Kounev. 2020. An IoT Network Emulator for Analyzing the Influence of Varying Network Quality. In *Simulation Tools and Techniques - 12th EAI International Conference, SIMUtools 2020, Guiyang, China, August 28-29, 2020, Proceedings, Part II (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering)*, Houbing Song and Dingde Jiang (Eds.), Vol. 370. Springer, 580–599. https://doi.org/10.1007/978-3-030-72795-6_47

[22] Mostafa Jalal, Sara Wehbi, Suren Chilingaryan, and Andreas Kopmann. 2022. SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial Internet of Things. In *SSDBM 2022: 34th International Conference on Scientific and Statistical Database Management, Copenhagen, Denmark, July 6 - 8, 2022*, Elaheh Pourabbas, Yongluan Zhou, Yuchen Li, and Bin Yang (Eds.). ACM, 12:1–12:11. https://doi.org/10.1145/3538712.3538723

[23] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Trans. Knowl. Data Eng.* 29, 11 (2017), 2581–2600. https://doi.org/10.1109/TKDE.2017.2740932

[24] Yuyuan Kang, Xiangdong Huang, Shaoxu Song, Lingzhe Zhang, Jialin Qiao, Chen Wang, Jianmin Wang, and Julian Feinauer. 2022. Separation or Not: On Handing Out-of-Order Time-Series Data in Leveled LSM-Tree. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 3340–3352. https://doi.org/10.1109/ICDE53745.2022.00315

[25] Tarun Kathuria and S. Sudarshan. 2017. Efficient and Provable Multi-Query Optimization. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts (Eds.). ACM, 53–67. https://doi.org/10.1145/3034786.3034792

[26] Lingkun Kong and Konstantinos Mamouras. 2020. StreamQL: a query language for processing streaming time series. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 183:1–183:32. https://doi.org/10.1145/3428251

[27] Alberto Lerner and Dennis E. Shasha. 2003. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer (Eds.). Morgan Kaufmann, 345–356. https://doi.org/10.1016/B978-012722442-8/50038-0

[28] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 739–752. https://www.usenix.org/conference/atc19/presentation/li-yongkun

[29] Leonid Libkin, Rona Machlin, and Limsoon Wong. 1996. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 228–239. https://doi.org/10.1145/233269.233335

[30] Rui Liu and Jun Yuan. 2019. Benchmarking time series databases with IoTDB-benchmark for IoT scenarios. *arXiv preprint arXiv:1901.08304* (2019).

[31] S.H. Low, F. Paganini, and J.C. Doyle. 2002. Internet congestion control. *IEEE Control Systems Magazine* 22, 1 (2002), 28–43. https://doi.org/10.1109/37.980245

[32] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1 (2017), 5:1–5:28. https://doi.org/10.1145/3033273

[33] Chen Luo and Michael J. Carey. 2019. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *Proc. VLDB Endow.* 12, 5 (2019), 531–543. https://doi.org/10.14778/3303753.3303759

[34] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2071–2086. https://doi.org/10.1145/3318464.3389731

[35] Michael Maddox, David Goehring, Aaron J. Elmore, Samuel Madden, Aditya G. Parameswaran, and Amol Deshpande. 2016. Decibel: The Relational Dataset Branching System. *Proc. VLDB Endow.* 9, 9 (2016), 624–635. https://doi.org/10.14778/2947618.2947619

[36] Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. *Proc. ACM Manag. Data* 1, 4 (2023), 247:1–247:26. https://doi.org/10.1145/3626734

[37] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[38] Behandelt PostgreSQL. 1996. PostgreSQL. (1996).

[39] Raghu Ramakrishnan, Donko Donjerkovic, Arvind Ranganathan, Kevin S. Beyer, and Muralidhar Krishnaprasad. 1998. SRQL: Sorted Relational Query Language. In *10th International Conference on Scientific and Statistical Database Management, Proceedings, Capri, Italy, July 1-3, 1998*, Maurizio Rafanelli and Matthias Jarke (Eds.). IEEE Computer Society, 84–95. https://doi.org/10.1109/SSDM.1998.688114

[40] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 249–260. https://doi.org/10.1145/342009.335419

[41] Maximilian E. Schüle, Josef Schmeißer, Thomas Blum, Alfons Kemper, and Thomas Neumann. 2021. TardisDB: Extending SQL to Support Versioning. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2775–2778. https://doi.org/10.1145/3448016.3452767

[42] Joshua E. Siegel, Dylan C. Erb, and Sanjay E. Sarma. 2018. A Survey of the Connected Vehicle Landscape - Architectures, Enabling Technologies, Applications, and Development Areas. *IEEE Trans. Intell. Transp. Syst.* 19, 8 (2018), 2391–2406. https://doi.org/10.1109/TITS.2017.2749459

[43] Somayeh Sojoudi, Steven H. Low, and John C. Doyle. 2014. Buffering Dynamics and Stability of Internet Congestion Controllers. *IEEE/ACM Trans. Netw.* 22, 6 (2014), 1808–1818. https://doi.org/10.1109/TNET.2013.2287198

[44] Shaoxu Song, Fei Gao, Aoqian Zhang, Jianmin Wang, and Philip S. Yu. 2021. Stream Data Cleaning under Speed and Acceleration Constraints. *ACM Trans. Database Syst.* 46, 3 (2021), 10:1–10:44. https://doi.org/10.1145/3465740

[45] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2019. Optimal and General Out-of-Order Sliding-Window Aggregation. *Proc. VLDB Endow.* 12, 10 (2019), 1167–1180. https://doi.org/10.14778/3339490.3339499

[46] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proc. ACM Manag. Data* 1, 2 (2023), 195:1–195:27. https://doi.org/10.1145/3589775

[47] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel (Eds.). ACM, 16:1–16:14. https://doi.org/10.1145/2592798.2592804

[48] Wolfgang Weiss, Víctor Juan Expósito Jiménez, and Herwig Zeiner. 2020. Dynamic Buffer Sizing for Out-of-order Event Compensation for Time-sensitive Applications. *ACM Trans. Sens. Networks* 17, 1 (2020), 1:1–1:23. https://doi.org/10.1145/3410403

[49] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 323–336. https://doi.org/10.1145/3183713.3196931

[50] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct Range Filters. *ACM Trans. Database Syst.* 45, 2 (2020), 5:1–5:31. https://doi.org/10.1145/3375660

[51] Ying Zhang, Martin L. Kersten, Milena Ivanova, and Niels Nes. 2011. SciQL: bridging the gap between science and relational DBMS. In *15th International Database Engineering and Applications Symposium (IDEAS 2011), September 21 - 27, 2011, Lisbon, Portugal*. ACM, 124–133. https://doi.org/10.1145/2076623.2076639

Received October 2023; revised January 2024; accepted February 2024