December 2021

Practical Target-Based Synchronization Strategies for Immutable Time-Series Data Tables

Bennett Andrew Meares
Clemson University, bennett.meares@gmail.com

Practical Target-Based Synchronization Strategies for Immutable Time-Series Data Tables

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

---

by
Bennett Meares
December 2021

Accepted by:
Dr. Amy Apon, Committee Chair
Professor Mitch Shue
Dr. David White

Abstract

As the Internet of Things and industrial monitoring of utilities grow, efficiently synchronizing immutable time-series data streams between databases becomes a pressing issue. Extracting data from critical production databases demands careful consideration of the stress imposed on the machines, so synchronization strategies are required to minimize the transfer of duplicate data and the load imposed on remote sources.

Literature on the synchronization problem is generalized to arbitrary tables and does not consider the characteristics of time-series data streams, so research was required to investigate methods to quickly synchronize source and target time-series data tables. This thesis examines immutable time-series scenarios and synchronization strategies to answer the following question: given several scenarios, which target-based immutable time-series synchronization strategies best optimize run-time, bandwidth, and accuracy?

The strategies explored in this research are implemented into the Meerschaum system, a project intended to leverage these time-series concepts for production deployments. As a practical demonstration, these strategies are used to continuously cache Clemson University's utilities data.

Acknowledgements

This thesis would not have been possible without support from many people in my life, and for this encouragement I am extremely grateful. I would like to extend my thanks to the following people in particular:

- **Professor Mitch Shue** and **Dr. Amy Apon** for their diligence and valuable insight which helped me organize my ideas into this academic paper.
- **Dr. David White**, **Snowil Lopes**, and **Tim Howard** for taking a chance on me in CEVAC and giving me space to grow and test my ideas.
- **Zach Smith**, **Drew Emery**, and **Harrison Hall** for their unending encouragement and strong friendship. Special thanks to Zach for lending me his mathematical prowess when comprehending *CPISync*.
- **Summer Robinson** for patiently listening to me ramble about synchronization strategies as we drove the van across the continent.
- **The FOSS community** which continues to inspire me to write the best software I can.

Table of Contents

List of Figures

Nomenclature

Pipe

Meerschaum representation of time-series data streams. A pipe corresponds to a target table and the metadata required to synchronize it.

Sample

Rows fetched from a source or target table used in the synchronization procedure.

ART **Approximation Reconciliation Tree**

A data structure which combines properties of the Patricia Trie, Merkle Tree, and Bloom Filter to quickly approximate the difference between sets.

BTI **Backtrack Interval**

The amount of time to backtrack from the *RT* to create the *ST*.

CPI **Characteristic Polynomial Interpolation**

A method of set reconciliation via rational interpolation of two sets' characteristic polynomials, functions whose zeros are the elements of the corresponding sets.

ET **End Time**

The newest boundary timestamp for selecting new rows.

ETL **Extract, Transform, Load**

A common data engineering procedure for copying and manipulating data from multiple sources and inserting into a single store.

IBF **Invertible Bloom Filter**

A variation of the Bloom Filter which allows set items to be retrieved and the difference between sets calculated.

IBLT **Invertible Bloom Lookup Table**

A variation of the Invertible Bloom Filter which allows for key-value pairs.

RT **Reference Time**

The reference timestamp for the synchronization algorithm. If the *RT* cannot be determined, then no optimizations may be made.

SQL **Structured Query Language**

The standard language for modifying and retrieving data from relational databases. The PostgreSQL dialect is employed in this thesis.

ST **Start Time**

The oldest boundary timestamp for selecting sample rows.

TBDS **Target-Based Database Synchronization**

An algorithm for determining rows missing from a target table without altering the schema of the source database.

Chapter 1

Introduction

Many real-world data streaming applications generate immutable time-series data streams. Particularly in the growing IoT industry, numerous commercial and open-source time-series data management systems has emerged, such as the studies published by Jensen et al. [2017], Pelkonen et al. [2015], Wang et al. [2020], Yang et al. [2019], and Rhea et al. [2017]. A persistent problem when working with time-series data is how to regularly and efficiently synchronize historical data between databases, such as when copying records from a production server to an analytical database.

Simple strategies may work well for a few, small tables, but as the number of data streams grows, so too does the need to reduce the processing and bandwidth requirements per transaction. For example, copying a table of a few thousand rows might only take a second, but updating several growing tables with millions of rows each requires careful planning to rapidly synchronize changes.

1.1 Synchronization in Practice

Many situations rely on the efficient synchronization of time-series data, and this section addresses case studies in the areas of industry, science, and finance.

1.1.1 Smart Buildings

Many institutions retrofit their facilities into "smart" buildings to monitor utilities usage and reduce carbon emissions [Lazarova-Molnar and Mohamed, 2019]. Oftentimes, legacy systems simply insert sensor readings into large database tables and therefore were not initially intended for frequent access. Due to the critical nature of these systems, external resources must be allocated for analysis, and sub-streams of data must be regularly synchronized from the source database.

One practical example is Clemson University's utilities data analytics system, which routinely syncs data from several production databases into an analytical cache database, on which further ETL steps are taken. One of the production databases is a 2014 Microsoft SQL Server which contains a table of five billion rows that grows by tens of thousands of rows per day (with sensors reporting at frequencies ranging from once per minute to hourly), resulting in a table size of roughly 300 GB. Although the link between the databases is fairly large (approximately one gigabit of bandwidth), queries on the production database and data transferred over the network must be kept to a minimum to avoid long delays or overloading mission-critical infrastructure. Therefore a set of strategies was required to readily fetch new data for analysis without abusing the sensitive remote servers.

1.1.2 Environmental Observations

Observational environmental data inherently contain a time dimension, and given a high temporal frequency, the volume of data grows rapidly. Inserting readings from environmental sensors into the source database is an append-only procedure because the new readings are guaranteed to be unique, but as data are replicated and the overall system grows in complexity, opportunities to save resources arise.

Consider the large-scale environmental monitoring that the National Oceanic and Atmospheric Administration (NOAA) undertakes. NOAA operates several satellite programs to capture environmental data, which are archived and distributed through the Comprehensive Large Array-Data Stewardship System (CLASS). One of these programs whose data are available in CLASS is the Joint Polar Satellite System (JPSS), a collaborative program between NOAA and NASA. To save bandwidth, data from the JPSS satellites are captured, processed, validated, and aggregated by the JPSS Ground System before being ingested by CLASS:

> The Data Processing Node (DPN) processes mission data into raw, sensor and environmental data products. Currently NOAA has JPSS-provided DPN implementations to minimize WAN communications utilization. [Vyas, 2019]

The transmission and storage of time-series data within JPSS and other NOAA programs are optimized internally, but external users must choose how to extract new data from CLASS. Users may specify datetime boundaries when requesting data, and given this read-only client-server model, synchronization strategies may be employed to minimize bandwidth and the demand imposed on NOAA's public servers.

Another notable example is the public weather API provided by NOAA and the National Weather Service (NWS). The API provides access to atmospheric readings (temperature, humidity, cloud coverage, etc.) from weather stations across the United States, mostly in regional airports. The records are reported hourly and are usually available within three hours, and the most recent week of data may be accessed.

Due to the scale of the operation, records are occasionally backlogged or modified, and outages are not uncommon. The API is designed for interactive applications, but these characteristics make building a historical data set complicated. To efficiently and accurately build a historical data set, a synchronization strategy is required. The API is integrated into Meerschaum through the `noaa` plugin.

1.1.3 Financial Transactions

Another area where synchronizing time-series data is important is the financial sector. Financial transactions are, for the most part, event-based data streams which share many similarities with the time-series data streams mentioned above. A key distinction between event streams and "regular" time-series data streams is frequency; whereas sensor-based data streams produce regular readings, event streams generate irregular intervals. This characteristic introduces more hurdles when synchronizing:

> *Is a gap of missing rows due to an outage or just a feature of the data stream?*
>
> *How can backlogged rows be detected and accounted for?*

One practical example of a company wrangling with the synchronization problem is M1 Finance's fraud-detection system [Onica, 2020]. M1 regularly parses several data streams (e.g. login attempts and transactions) for detecting suspicious activity and for internal analysis. To synchronize data for the analysts, M1 streams login

attempts with AWS Data Migration Service into their S3 warehouse bucket as a collection of Parquet files. The data warehouse, which is roughly 400 GB in size, grows by one to five GB per day with frequencies ranging between hundreds to thousands of events per second. The analysts execute queries on the warehouse using Amazon Redshift Spectrum, a federated SQL query execution engine which bypasses traditional ETL and allows queries to be run on a Redshift cluster without completely loading the source data into tables. This architecture can dynamically create compute resources, thereby shifting the analytical demand away from production servers. Additional caching is done by their BI tools to further reduce processing and network demand.

Because M1 Finance controls the source database, steps were able to be taken to offload the analytical demand away from the production machines. When an architecture is designed where changes from the source database are pushed to an intermediate store (in this case the S3 bucket), then demand from resolving problems like backlogging are pushed "down the chain" to the analytical layer. The approach to use Amazon Redshift Spectrum and S3 saves disk space but still requires careful consideration of data retention and handling changed records.

Another example to consider is the transactions API provided by Apex Clearing Corporation, the clearing house behind M1 Finance. Like the CLASS interface mentioned in subsection 1.1.2, Apex Clearing offers a tool to extract transaction data over a given interval. The API which powers the tool may be accessed with a web driver, and the extracted data are an example of the kind of read-only access that demands a synchronization strategy to update a historical data set. The API is adapted to the Meerschaum system via the `apex` plugin.

1.2 Related Works

Earlier database synchronization studies have produced many efficient algorithms, and awareness of existing work is necessary when creating novel strategies. These set reconciliation algorithms are generalized and as such are limited in capabilities. This thesis is intended to demonstrate how the inclusion of common properties like a datetime axis greatly extends design possibilities and optimization opportunities.

1.2.1 Hashing Partitions

Target-Based Database Synchronization

The constraints of target-based synchronization are addressed by Ahluwalia et al. [2010]. To detect changes in a read-only source table, Ahluwalia et al. propose an algorithm similar to `rsync` [Tridgell and Mackerras, 1996] called Target-Based Database Synchronization (*TBDS*) in which tables are partitioned and hashed, and target partitions with different hashes from the source are replaced. The framework for this approach is later implemented in *Iterative Syncs* (section 3.2) with the primary distinction that *TBDS* uses hashing to detect mutability but *Iterative Syncs* rely on row-counts because tables are assumed to be immutable.

Synchronization Algorithms based on Message Digest

Specifically in the realm of mobile devices, Choi et al. [2010] propose Synchronization Algorithms based on Message Digest (*SAMD*). *SAMD* takes a multi-staged approach during which tables of hash values are stored and compared to detect which rows must be synchronized. This algorithm requires that the source and target databases must maintain a message digest table and therefore allow write access on

the source database, thereby breaking the target-based and read-only constraints outlined by Ahluwalia et al. [2010]. The core principle of comparing hashes nonetheless proves valuable when designing target-based synchronization strategies.

1.2.2 Characteristic Polynomial Interpolation

CPISync

Minsky and Trachtenberg [2001] propose a set reconciliation algorithm using the properties of polynomials, first implemented as *CPISync* by Trachtenberg et al. [2002] as a novel approach for synchronizing PDAs between PCs.

CPISync optimizes network load by only transmitting the rational values from evaluating the data sets' characteristic polynomials on each host. The overall performance of *CPISync* is tied to a number of samples ($\bar{m}$) necessary to solve the system of equations to recover the differences between the sets. Therefore, *CPISync* is best used when little data have changed between hosts and conserving bandwidth is paramount. Read below to better understand *CPISync*.

1. A characteristic polynomial of a set $S = \{x_1, x_2, ..., x_n\}$ has the following form, such that the zeroes of the function are the elements of S.

$$\chi_S(Z) = \prod_{i=1}^{n}(Z - x_i)$$

2. The ratio between characteristic polynomials of two sets is equal to the ratio of

the characteristic polynomials of the differences between the sets.

$$\Delta_A = S_A - S_B$$

$$\Delta_B = S_B - S_A$$

$$\frac{\chi_A(Z)}{\chi_B(Z)} = \frac{\chi_{\Delta_A}(Z)}{\chi_{\Delta_B}(Z)}$$

3. A large enough value for $\bar{m}$ must be determined to account for the number of differences between the sets ($\bar{m} = m_{\Delta_A} + m_{\Delta_B}$), and a predetermined set of values (of size $\bar{m}$) is evaluated for the data sets' characteristic polynomials on each host. The ratio between the evaluations is equal to the ratio of the difference sets.

   The following calculations are performed over $\mathbb{F}_{13}$.

$$\bar{m} = 4$$

$$k = [1 \,..\, \bar{m}]$$

$$S_Z = -k = \{-1, -2, -3, -4\}$$

$$S_A = \{1, 3, 5, 7, 9\}$$

$$S_B = \{1, 3, 5, 7\}$$

$$\chi_{S_A}(Z) = (Z-1)(Z-3)(Z-5)(Z-7)(Z-9)$$

$$\chi_{S_B}(Z) = (Z-1)(Z-3)(Z-5)(Z-7)$$

Z_k	$\chi_A(Z)$	$\chi_B(Z)$	$\chi_A(Z) \,/\, \chi_B(Z)$
-1	8	7	3
-2	5	9	2
-3	9	9	1
-4	0	7	0

4. The rational function $\chi_{\Delta_A}(Z) \,/\, \chi_{\Delta_B}(Z)$ can be recovered through rational function interpolation.

$$R(Z) = \frac{P(Z)}{Q(Z)} = \frac{\chi_A(Z)}{\chi_B(Z)} = \frac{\chi_{\Delta_A}(Z)}{\chi_{\Delta_B}(Z)}$$

In cases such as those addressed in chapter 2 where S_B is a subset of S_A, Δ_B will always be the null set, and the recovered polynomial will only contain the elements of Δ_A.

$S_A = \{1, 3, 5, 7, 9\}$

$S_B = \{1, 3, 5, 7\}$

$\Delta_B = \emptyset$

$$\therefore \frac{\chi_A(Z)}{\chi_B(Z)} = \frac{\chi_{\Delta_A}(Z)}{1} = \chi_{\Delta_A}(Z) = P(Z) = R(Z)$$

The polynomial $R(Z)$ can be represented as a the sum of a series of coefficients multiplied by growing powers of Z.

$$R(Z_k) = \chi_{\Delta_A}(Z_k) = \sum_{i=0}^{\bar{m}} p_i Z_k^i = p_0 + p_1 Z_k + p_2 Z_k^2 + \ \ldots \ + p_{\bar{m}} Z_k^{\bar{m}}$$

The coefficients of $R(Z)$ can determined by solving a system of linear equations

from the points $(Z_k, \frac{\chi_A(Z_k)}{\chi_B(Z_k)})$ collected from the hosts.

$$\begin{bmatrix} Z_1^0 & Z_1^1 & Z_1^2 & Z_1^3 \\ Z_2^0 & Z_2^1 & Z_2^2 & Z_2^3 \\ Z_3^0 & Z_3^1 & Z_3^2 & Z_3^3 \\ Z_4^0 & Z_4^1 & Z_4^2 & Z_4^3 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix}$$

$$p = \{4, 1, 0, 0\}$$

$$R(Z) = Z + 4$$

5. The zeroes of the recovered interpolated function $R(Z)$ correspond to Δ_A (calculations performed over $\mathbb{F}_{13}$).

$$S_A = \{1, 3, 5, 7, 9\}$$

$$S_B = \{1, 3, 5, 7\}$$

$$\Delta_A = \{9\}$$

$$R(9) = 9 + 4 = 0$$

Partitioned-CPISync

As part of the e-Triage project [Greiner and Donner, 2010] (sponsored by the German Federal Ministry of Education and Research), Tang et al. [2010] expand upon the work of Trachtenberg et al. [2002] and compare several synchronization strategies in the context of satellite communications where conserving bandwidth is crucial and latency is especially high. Tang et al. [2010] compare three different synchronization mechanisms: *Slow Sync* as a baseline, *Maatkit* and the algorithms provided by the toolkit, and a modified version of *CPISync* dubbed *Partitioned-CPISync*. *Partitioned-*

CPISync is designed to account for scenarios where $\bar{m}$ is unknown. The algorithm recursively partitions the data sets and executes *CPISync* (referred to as *Basic-CPISync* in the paper) when the size of a partition is below the threshold ($m_p < \bar{m}$).

Priority CPISync

Jin et al. [2012] also take the work of Trachtenberg et al. [2002] to a more practical level with *Priority CPISync* (*P-CPI*). Like *Partitioned-CPISync*, *P-CPI* partitions the data sets and calls *CPISync* on individual partitions. The primary distinctions for *P-CPI* is the paper's probability analysis which demonstrates the number of *CPISync* invocations to be $O(\eta m \log(\eta m))$ with high probability of at least $1 - \frac{1}{\eta m}$ (with a worst-case of $O(mb)$ and best case of $O(m \log(m))$.

Efficient Synchronization over Broadcast Networks

Like Tang et al. [2010], Muhammad et al. [2013] explore *CPISync* for satellite communications. Muhammad et al. consider *CPISync*'s role in the distributed databases context and explore several network topologies and techniques for obtaining differences across a broadcast medium. Specifically, the authors consider a mesh network (in which each node syncs with its peers), a star network where the master node broadcasts all of the packets and nodes accept or drop packets, and a star network with network coding on packets which have not been correctly received on all nodes. The paper demonstrated how the number of packets transmitted scales as functions of the number of users and differences between nodes. Despite introducing processing delays, the network coding star topology was found to minimize the transmitted packets in most cases.

1.2.3 Space-Efficient Approximate Synchronization

Bloom Filters

Bloom Filters have been used to determine set membership since the 1970s [Bloom, 1970]. Algorithms which utilize Bloom Filters benefit from their space-efficiency and unique capability to quickly query tables where no clear index can be constructed, such as text-based information retrieval. Bloom Filters are well-researched and are often implemented into popular database systems [Byali et al., 2020].

The original Bloom Filter is a probabilistic and space-efficient structure for determining set membership with a low false-positive rate. It consists of a bit array of length m and k hash functions which map set items to indices — integers in the range $[0, m)$. The filter supports two operations: $add()$ and $query()$. To add elements to the Bloom Filter, k indices are calculated by hash functions $h_k(x)$, and the bits in the array at the corresponding indices are set to 1. When querying an item's membership, its indices are calculated and checked in the bit array. If all bits at the indices are 1, the item is *probably* in the set, but if any of the bits are 0, the item is *definitely not* in the set.

Approximation Reconciliation Trees

Due to the significant computation required for exact synchronization techniques (such as *CPISync* per Trachtenberg et al. [2002]), Byers et al. [2002] instead offer an approximate solution with greatly reduced computational complexity called an Approximation Reconciliation Tree (ART) which combines properties of existing approximation structures, namely Bloom Filters, Patricia Tries [Knuth, 1973], and Merkle Trees [Merkle, 1980].

An *ART* represents a set as a Patricia Trie to structure searches and uses a Merkle Tree to make searching the Patricia Trie feasible. Merkle Trees represent sets as a tree of hash values such that the hash of a node is dependent on the hashes of its children. Finally, the *ART* summarizes the Patricia Trie and Merkle Tree construction as a Bloom Filter, which is the message transmitted during the synchronization process. Byers et al. note that the use of the summary Bloom Filter nearly eliminates collisions in the Merkle Tree and avoids complications from collapse operations when comparing across Patricia Tries. The authors conduct experiments to evaluate the accuracy and speed performances of *ART* and conclude that the speed of *ART* is inverse to the number of corrections and outperforms standard Bloom Filters for set differences less than 2% with two correction passes or differences less than 30% for no correction passes.

Invertible Bloom Filters

Eppstein and Goodrich [2010] introduce the Invertible Bloom Filter (*IBF*), a variation of the standard Bloom Filter (and extension of the counting Bloom Filter) which allows set items to be retrieved and differences calculated. Rather than a simple bit array, the array of the *IBF* contains three fields: *count*, *idSum*, and *hashSum*. When adding an element into an *IBF*, the same k indices are produced by hash functions $h_k(x)$. The *count* fields of the cells at the generated indices are incremented, the value of the element is added to the *idSum* fields, and the resulting hash value of an additional hash function $g(x)$ is added to the *hashSum* fields. Finally, an additional fallback Bloom Filter with the same number of cells and two randomized hash functions $f_1(x)$ and $f_2(x)$ is used for elements which are difficult to recover from the primary Bloom Filter.

The *IBF* yields a major advantage over a regular Bloom Filter: set items

may be retrieved from the filter and therefore the difference between two sets may be calculated from two filters. Like *CPISync*, two filters may be independently evaluated on separate hosts and the difference between the sets derived by subtracting the contents of the filters (though not directly subtracting the filters, which was later added in a later revision of the *IBF* [Eppstein et al., 2011]).

Invertible Bloom Lookup Tables

Goodrich and Mitzenmacher [2011] extend the *IBF* to include key-value pairs and name the variation the Invertible Bloom Lookup Table (*IBLT*). The authors describe the changes made to the *IBF* as "deceptively simple" and explain in-depth the application of the *IBLT* in the database reconciliation space.

Like the *IBF*, the *IBLT* includes three fields: *count*, *keySum*, and *valueSum*. Similar to the fields of the *IBF*, these fields contain the sums of the values mapped to the cell, but rather than deriving a dedicated hash value (the purpose of $g(x)$ in the *IBF*), the provided keys and values are used to update the fields.

Difference Digest

Eppstein et al. [2011] put the *IBF* to use as a component of the Difference Digest, a structure for set reconciliation. Because the efficiency of the *IBF* (and Bloom Filters in general) depends on its size, Eppstein et al. include a *Strata Estimator* to gauge the size of the difference between the sets ($\bar{m}$ in the case of *CPISync*). Additionally, to allow for filter subtraction, the authors tweak the manner in which $idSum$ and $hashSum$ fields are updated: rather than simple addition and subtraction, the fields are updated with the XOR of the elements. This has the effect of producing negative counts which did not appear in the original implementation of the *IBF*.

To evaluate the efficacy of the Difference Digest, Eppstein et al. compare its

performance against three strategies: (1) Approximate Reconciliation Trees (*ART*), (2) *CPISync*, and (3) a naïve approach of trading a sorted list of the target table's keys (referred to as *List*). The authors note that without precomputation, Difference Digest is significantly slower than *List*, but with precomputation and small set differences (less than 15%), Difference Digest can outperform the naïve approach by an order of magnitude. Additionally, the authors demonstrate how the bandwidth performance of *List* and *ART* improves as the set difference grows because the two approaches encode the target set.

As demonstrated in subsection 4.1.3, the authors note that *CPISync* is an ideal choice for preserving bandwidth and achieving an accurate result at a steep cost of expensive computation. For small set differences, both Difference Digests and *CPISync* require significantly less bandwidth than *ART* and *List*. The authors conclude that precomputed Difference Digests are superior in situations of constrained computation where the difference between the sets is small.

Key-Value Storage System Synchronization in Peer-to-Peer Environments

Similar to this thesis, Pham [2014] discusses several set reconciliation algorithms in the context of peer-to-peer key-value synchronizations between mobile devices. Among these algorithms are *ART*, *CPISync*, and *IBFSync* as well as log-based, timestamp-based, and naïve approaches. Pham categorizes the algorithms according to their behaviors: (1) communication rounds (bounded versus unbounded), (2) accuracy (exact versus approximate), and (3) awareness of prior context (e.g. log-based approaches that require metadata or context-free strategies like *CPISync*). Pham then proposes a novel algorithm named *ASync* which consists of first completing an approximate synchronization via a regular Bloom Filter then conducting an exact synchronization via an *IBF* to capture the remaining differences.

To demonstrate the performance of *ASync*, Pham evaluates the communication costs, processing time, and synchronization time of two of the discussed algorithms (along with a naïve approach): *IBFSync* and *ASync*. The two algorithms are compared against several degrees of changes of the source table, and the author concludes that the two-phase architecture of *ASync* leads to considerable improvements over *IBFSync* in terms of communications cost, processing time, and synchronization time.

Cuckoo Filters

Fan et al. [2014] note the space efficiency and limitations of standard Bloom Filters [Pagh et al., 2005] and consider variations of the Bloom Filter designed to address these limitations (such as the false-positive rate and deletion support), namely Counting Bloom Filters [Fan et al., 2000], Blocked Bloom Filters [Putze et al., 2010], d-left Counting Bloom Filters [Bonomi et al., 2006], and Quotient Filters [Bender et al., 2012]. Although these variations accommodate the shortcomings of the standard Bloom Filter, the improvements come at the cost of reduced space efficiency. Fan et al. instead propose a replacement for the Bloom Filter called the Cuckoo Filter which offers support for adding and removing items while outperforming Bloom Filters. Additionally, the authors experimentally demonstrate that Cuckoo Filters outperform the stated Bloom Filter variations.

Similar in structure to the standard Bloom Filter, the underlying structure and hashing scheme of the Cuckoo Filter are based on Cuckoo Hash Tables [Pagh and Rodler, 2004]. A basic Cuckoo Hash Table consists of an array of "buckets" and hash functions $h_1(x)$ and $h_2(x)$ which determine the indices of candidate cells for items. An unoccupied candidate cells is selected if available, otherwise an occupied cell is chosen, and the existing occupant is displaced (hence the name *cuckoo hashing*).

The Cuckoo Filter employs Cuckoo Hash Tables for set reconciliation by means of fingerprinting and partial key cuckoo hashing. When inserting items, a fingerprint and candidate index are calculated, and the index of the alternate cell is derived by an XOR of the first hash function. Because buckets may contain duplicate fingerprints, the fingerprint size may be kept small to reduce size requirements of the entire structure.

Variations of the Cuckoo Filter include the Conditional Cuckoo Filter [Ting and Cole, 2021] and Adaptive Cuckoo Filter [Mitzenmacher et al., 2020] which extend Cuckoo Filters by allowing for duplicate keys and significantly reduce the false positive rate, respectively.

XOR Filters

Noting the efficiency of Bloom and Cuckoo Filters, Graf and Lemire [2020] implement an approach called the Bloomier Filter [Chazelle et al., 2004] and name the implementation the XOR Filter. The underlying structure consists of an array slightly larger than the cardinality of the set. Three hash functions $h_1(x)$, $h_2(x)$, and $h_3(x)$ generate indices of items in each third of the array. This is intended to maintain an aggregated XOR value of the three array locations that is equal to the item's fingerprint.

The authors discuss in depth the benchmark results of several variations of the Bloom Filter, Cuckoo Filter, Blocked Bloom Filter, and the XOR Filter. The XOR Filter outperforms each structure in speed and space requirements with the exception of the Blocked Bloom Filter in speed but not space. They conclude that although the construction of the XOR Filter is roughly twice as slow as a regular Bloom Filter, this cost is amortized over many queries, and memory requirements are reduced by approximately 15%.

1.2.4 Summary

From Bloom Filters in the 1970s through *CPISync* in the 2000s and Cuckoo Filters in the 2020s, advancements continue to be made in the database reconciliation space. This thesis aims to address an overlooked aspect of the set reconciliation problem: synchronizing immutable time-series data streams. The inclusion of a datetime axis expands the tools available when designing synchronization algorithms, and the following sections focus on taking advantage of this property when determining which data to fetch from the source database. The works discussed may be leveraged in the $filter()$ stage of the synchronization procedure, so to further research for immutable time-series situations, the rest of this thesis is dedicated to exploring novel strategies for fetching source and target samples.

1.3 Overview of the Algorithm

The basic stages of the synchronization algorithm are **fetch** (*Extract* and *Transform*), **filter**, and **insert** (*Load*). These steps make up a kind of ETL process specifically tuned for time-series data. The implementations of $fetch()$, $filter()$, and $insert()$ are mostly language- and protocol-independent and may be further optimized in production (for example, the $filter()$ function may precede $fetch()$ per *CPISync*; see subsection 3.2.4).

1.3.1 The *synchronize*() Procedure

The basic algorithm introduced below may be referred to as *Simple Backtrack Sync* (subsection 3.1.2).

1. **Determine the "reference time" (RT) datetime.**

   If none is provided, use the latest timestamp value from the target table as the RT. If no value is found, no optimizations may be made and the entire source table must be fetched.

   The RT will be the reference point for the synchronization.

2. **Determine the "backtrack interval" (BTI).**

   If none is provided, use a default value of one minute.

   The BTI "walks back" the RT to catch rows that were backlogged during the last synchronization.

3. **Derive the "start time" (ST) timestamp by subtracting the BTI from the RT.**

   The ST will also be $None$ if the RT could not be determined.

4. **Fetch a source and target sample, each with rows greater than the ST.**

   If the ST is $None$, the target sample does not need to be fetched.

5. **Filter out rows of the target sample from the source sample.**

   This may be skipped if the target sample was not fetched.

6. **Insert the filtered sample into the target table.**

Therefore, the pseudocode for this algorithm would be the following:

```
procedure SYNCHRONIZE(Source, Target, ReferenceTime, BacktrackInterval)
    if ReferenceTime is None then
        ReferenceTime ← LATEST(Target)
    StartTime ← ReferenceTime − BacktrackInterval
    ▷ StartTime will be None if ReferenceTime is still None. ◁
    SourceSample ← FETCH(Source, StartTime)
    if StartTime is None then
        FilteredSample ← SourceSample
    else
        TargetSample ← FETCH(Target, StartTime)
        FilteredSample ← FILTER(SourceSample, TargetSample)
    INSERT(Target, FilteredSample)
```

The figure below illustrates a scenario where new data are fetched, filtered, and inserted into the target table.

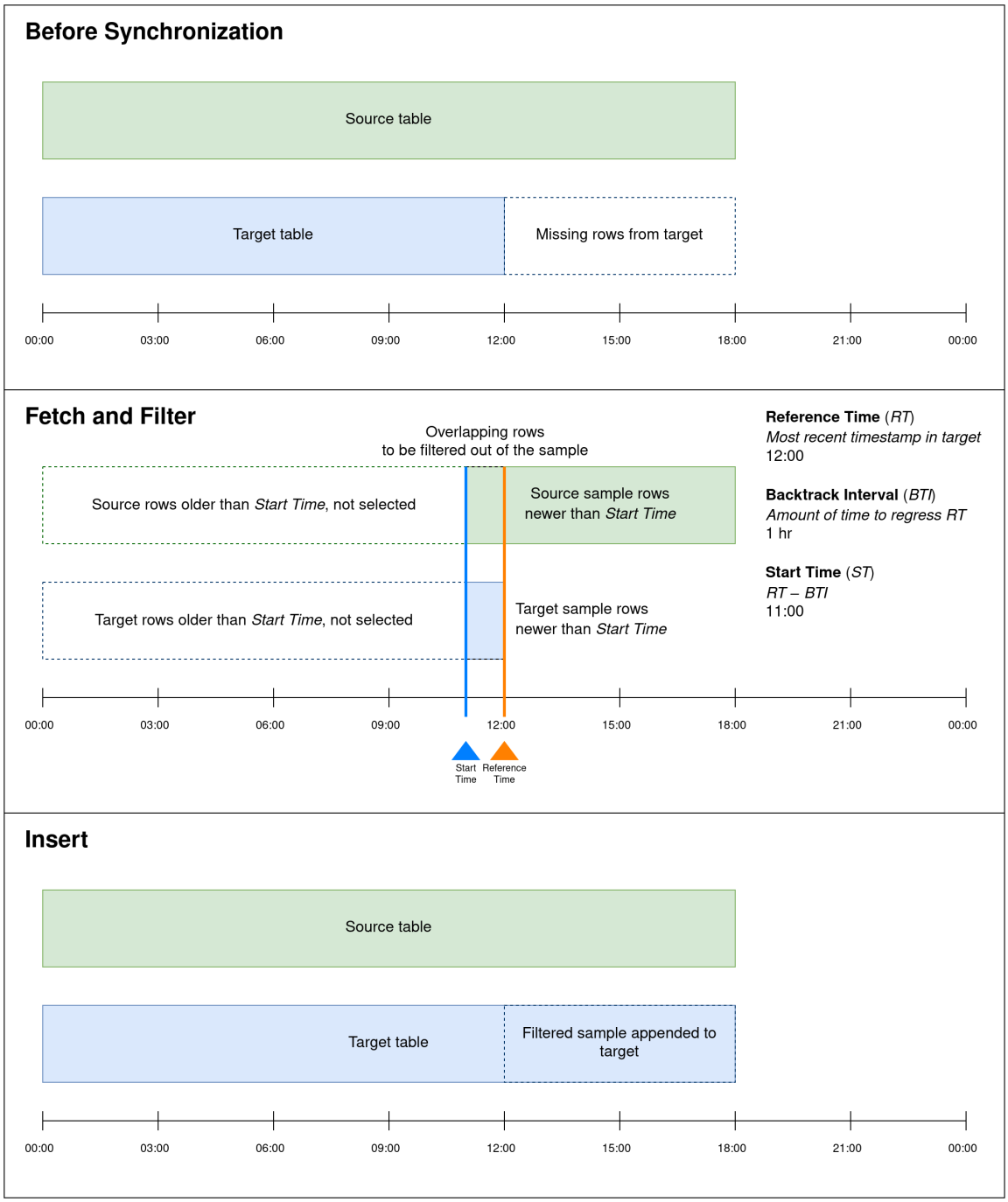

Figure 1.1: An example of *Simple Backtrack Sync*

1.3.2 The *fetch*() Function

The $fetch()$ function retrieves sample rows from a table, and because the source database may be a sensitive production server, one optimization goal during the fetch stage is to design a straightforward query which minimizes data sent over a network link. An example of $fetch()$ per *Simple Sync* in the form of a SQL query would look like the following:

```
SELECT *
FROM source
WHERE datetime > '2021-01-01 00:00:00'::TIMESTAMP
```

1.3.3 The *filter*() Function

The idea behind the $filter()$ function is to remove rows found in the target sample from the source sample. Below is a simple (although inefficient) pseudocode representation of $filter()$.

function FILTER($SourceSample$, $TargetSample$)
    if $TargetSample$ is $None$ **then**
        return $SourceSample$
    $FilteredSample \leftarrow$ TABLE()
    for all $row \in SourceSample$ **do**
        if $row \notin TargetSample$ **then**
            INSERT($FilteredSample, row$)
    return $FilteredSample$

Assuming the source table has `datetime` and `id` indices, an approximate SQL query for the $filter()$ function would look like the following:

```
SELECT source_sample.*
FROM source_sample
LEFT JOIN target_sample ON (
  source_sample.id = target_sample.id
    AND source_sample.datetime = target_sample.datetime
)
WHERE target_sample.datetime IS NULL
```

The above query performs as expected for immutable data. The following implementation in the popular Python data science library `pandas` considers all of the columns, not just the indices, so modified rows would be included in the filtered sample.

```
filtered_sample = source_sample[
  ~source_sample.fillna(custom_nan).apply(tuple, 1).isin(
    target_sample.fillna(custom_nan).apply(tuple, 1)
  )
].reset_index(drop=True)
```

One critical aspect of the Python implementation of the *filter*() function is that the order of columns must be retained. If the source table suddenly were to change the order of its columns, then all of the samples would appear to be new. Therefore, in the practical Meerschaum implementation, the order of columns in the source sample is enforced to be the same as the target.

1.3.4 The *insert*() Function

Once rows are filtered, the last step is to update the target table. For immutable data streams, the data are inserted into the target. The specific implementation depends on the environment (e.g. database flavor, protocol, implementation), but for the most part a series of `INSERT` statements may be generated, such as the default behavior of `DataFrame.to_sql()` in `pandas`. Additional implementations may be written to take advantage of database flavors such as the PostgreSQL's `COPY FROM STDIN WITH CSV` functionality [Meares, 2021a].

Chapter 2

Scenarios

Generating and appending data from sensors is a mostly straightforward task: take a reading, submit to an endpoint, and insert into the store. Complications arise when tables need to be synced between disconnected databases. The simple approach would be to drop target tables and continuously copy the source (i.e. *Naïve Sync*, see subsection 4.1.1), but characteristics of the scenario may be exploited to more efficiently synchronize the tables.

Despite the restrictions of immutability and a datetime axis, there exists substantial variability between time-series data streams. The frequency, temporal resolution, number of IDs, and prevalence of backlogging are among properties of scenarios which influence the performance and design of synchronization strategies. The follow sections detail factors of certain scenarios and ways to use these aspects when designing strategies.

2.1 Append-Only Data Streams

2.1.1 A Single, Simple ID

The first scenario will consist of a single sensor that regularly emits a reading (e.g. hourly) and each time appends a record into a table. A basic record consists of three columns: an eight-byte timestamp, four-byte integer ID, and an eight-byte float (double precision) value, which would translate to the following SQL query:

```
INSERT INTO source (
  datetime, id, value
) VALUES (
  '2021-01-01 00:00:00'::TIMESTAMP, 1, 1.0
);
```

This growing table will be the source table. The target table will reside on another database, which has no direct connection to the source database. Connecting the target and source databases will be our syncing service.

There are several aspects of this scenario that we can use to design our syncing strategy. First, time only moves in one direction. We can assume that no records will be backlogged into the source table. Second, there is only a single stream of data (one sensor). This allows us to use the datetime index as the primary method of determining which rows have already been accounted for.

For this scenario, the ideal syncing procedure is a simplified version of the overview introduced in section 1.3 called *Simple Sync* (subsection 3.1.1) and is as follows:

1. Determine the most recent datetime from the target to be the "start time" (ST).

   This will be a lightweight operation because the datetime column is indexed.

2. Fetch data from the source that is newer than the ST.

The efficiency of this step depends on whether the source is indexed by datetime.

3. Insert the fetched data into the target table.

A SQL query for fetching new rows may be akin to the following:

```
SELECT *
FROM source
WHERE datetime > '2021-01-01 00:00:00'::TIMESTAMP
```

2.1.2 Multiple Simple IDs

The second scenario is similar to the first; the only difference is that two sensors will be reporting to the source table. In this case, we assume that the sensors report within the same minute interval. Several attributes of this scenario allow us to design an appropriate strategy:

1. The data stream has datetime and ID indices.
2. Rows are immutable.
3. New rows always have later datetime values than existing rows.
4. Both sensors report within a known interval of each other.

Given these characteristics, the simple algorithm described in section 1.3 applies to this scenario. The notable difference from the previous scenario is that a backtrack interval (BTI) — one minute in this case — is subtracted from the most recent datetime (ST) so that new rows from both IDs are selected from the source. A SQL query for this approach may look like the following:

```
SELECT *
FROM source
WHERE datetime > (
  '2021-01-01 01:00:00'::TIMESTAMP
    - INTERVAL '1 minute'
)
```

Consider the figure below which illustrates why the basic algorithm works in this scenario. The key factor is that the sensors "walk in-step," i.e. after applying the known BTI, the table may be synchronized like a single data stream.

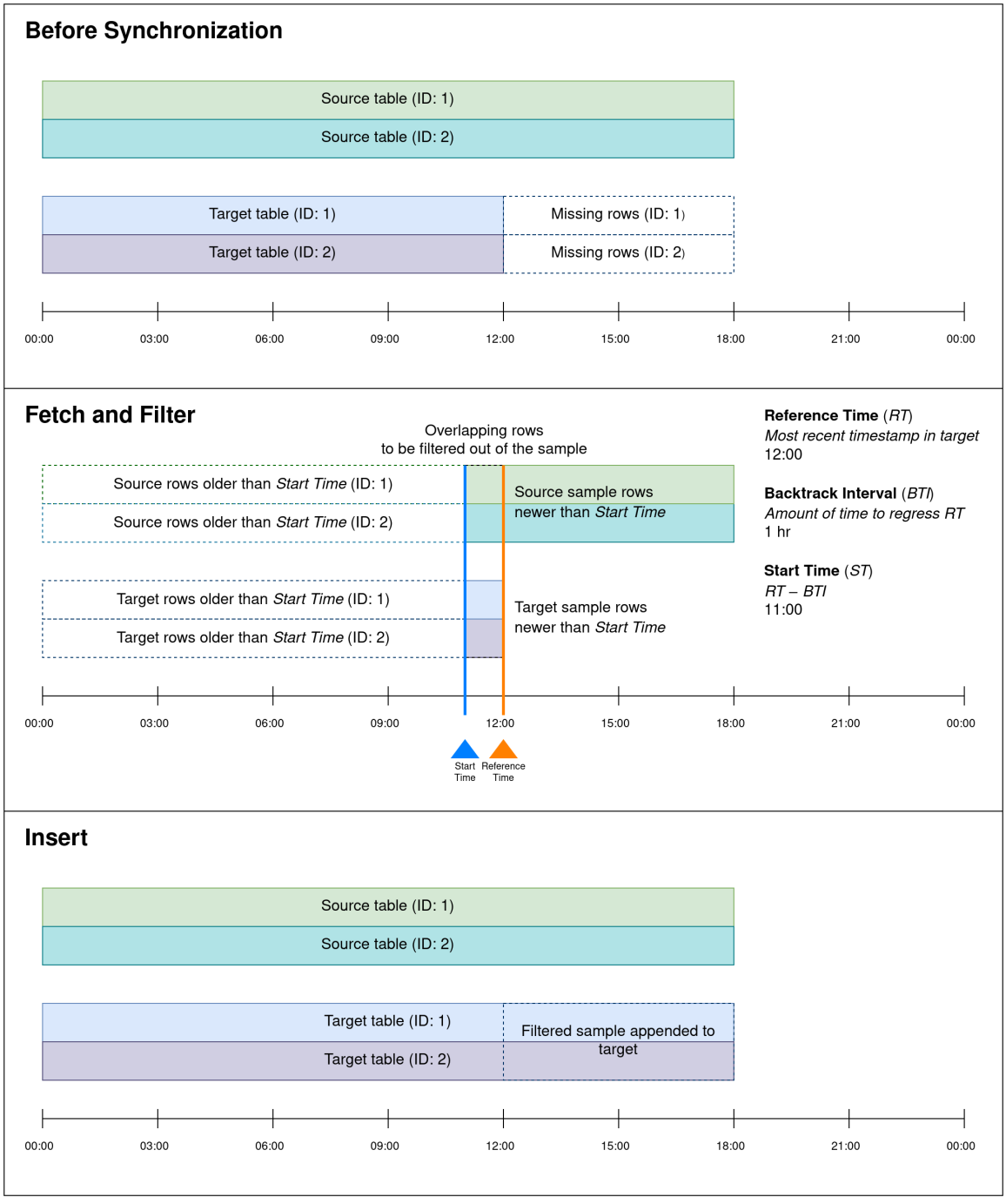

Figure 2.1: A known BTI can synchronize multiple sensors which "walk in-step" within a known interval.

2.2 Backlogged Data

2.2.1 A Single ID with Known Backlogged Data

The above strategies rely on the assumption that no rows will be backlogged into the source table. The BTI can catch small amounts of recently backlogged data, but the size of the interval window is the inversely proportional to the performance of the synchronization process.

Consider the single, append-only situation from subsection 2.1.1. If an old row is inserted into the source table three days after the most recent rows — perhaps due to an outage — then the aforementioned strategy *Simple Backtrack Sync* will fail to select this row and insert it into the target table. A new strategy must be designed in response to the possibility of backlogged data. For this scenario, the attributes of the data stream would be the following:

1. The data stream has a datetime index.
2. Rows are immutable.
3. New rows *usually* have later datetime values than existing rows.
4. Backlogged data are inserted within a known interval.

The key characteristic of this scenario is that backlogged rows are inserted into the source table within a known interval. For example, if rows were added en masse at the end of a day, then an additional pass of the synchronization strategy may take place over the day's interval to "look back" and "capture" any missing rows. To do this, a new parameter is required for the synchronization procedure, the "end time" (ET) datetime. The procedure would have the following steps:

1. **Determine the "start time" (ST) datetime.**

   Because the synchronization has a known interval, the ST will be the earliest datetime in the interval.

   The ST and RT have the same values.

2. **Determine the "end time" (ET) datetime.**

   The ET will be the latest datetime in the interval.

   The ET bounds the synchronization to reduce transferring duplicate rows.

3. **Fetch a source and target sample, each with rows greater than or equal to the ST and less than or equal to the ET.**

   Omitting ET will unbound the fetch all rows newer than ST, which may be detrimental to performance if the backlogged interval is significantly older than the newest target datetime value.

4. **Filter out rows of the target sample from the source sample.**

5. **Insert the filtered sample into the target table.**

The pseudocode for this scenario would be the following:

procedure SYNCHRONIZE($Source$, $Target$, $StartTime$, $EndTime$)

$SourceSample \leftarrow$ FETCH($Source, StartTime, EndTime$)

$TargetSample \leftarrow$ FETCH($Target, StartTime, EndTime$)

$FilteredSample \leftarrow$ FILTER($SourceSample, TargetSample$)

INSERT($Target$, $FilteredSample$)

The SQL query to be executed on the source database may have the following structure:

```
SELECT *
FROM source
WHERE datetime >= '2021-01-01 00:00:00'::TIMESTAMP
  AND datetime <= '2021-01-02 00:00:00'::TIMESTAMP
```

The following figure illustrates how a target table may be bounded to a known interval to "catch" missing, backlogged rows. The exact interval depends on the specific scenario, such as performing daily or weekly passes to verify the integrity of the "regular" synchronization process.

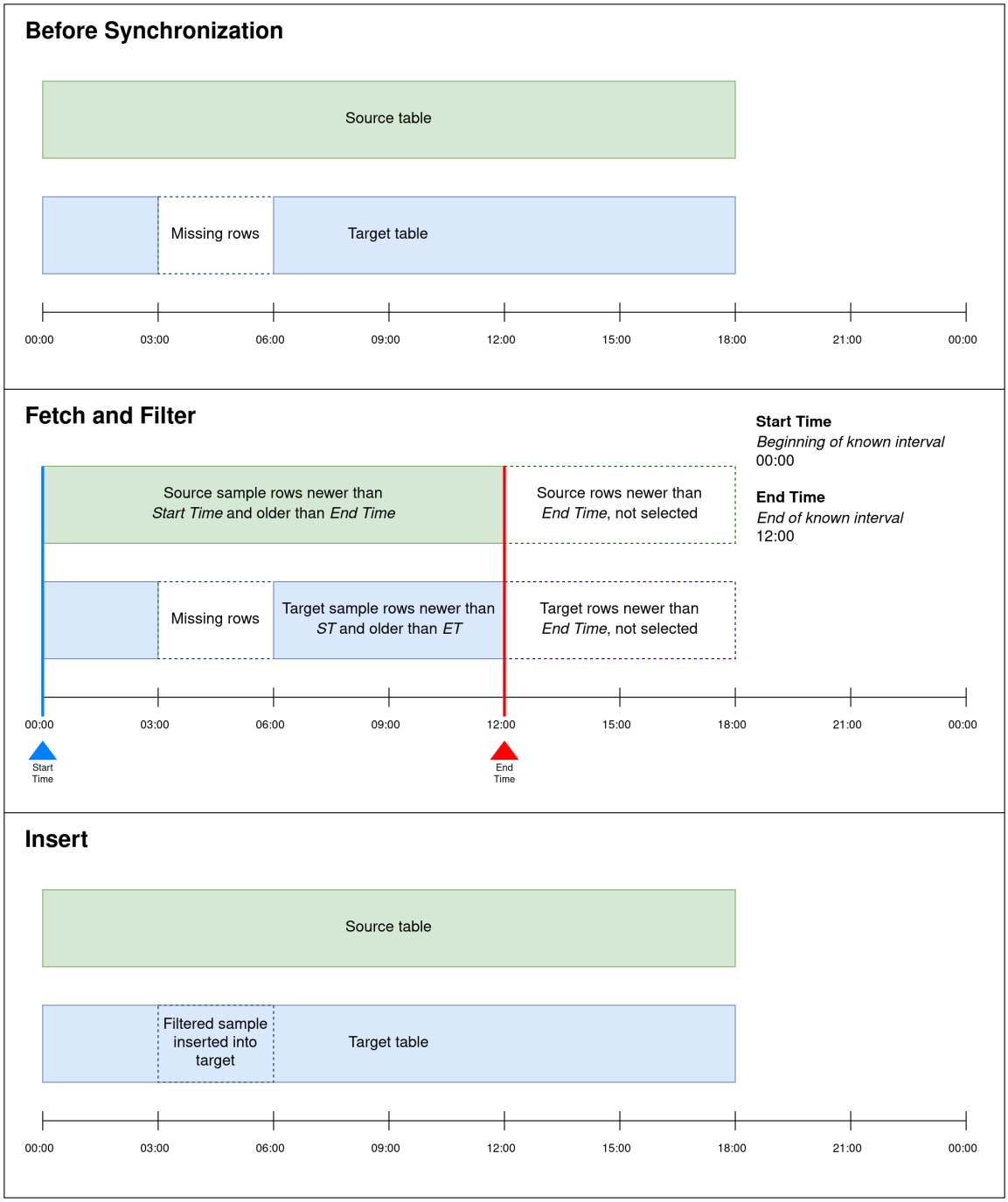

Figure 2.2: A synchronization may be bounded to account for backlogged rows.

2.2.2 Multiple IDs with Known Backlogged Data

Extending the previous situation by adding IDs allows for finer tuning when fetching samples. Suppose a pipe contains many regular append-only IDs, but a single sensor dumps all of its records into the source table once at the end of a month. Per the previous scenario (subsection 2.2.1), a verification pass over the interval of the entire month would successfully fetch the missing rows, but due to the number and frequency of most of the IDs, many duplicated rows would be fetched and sent over the network just to catch the single "slow" sensor.

The key takeaway is that the $fetch()$ query may be constrained by the ID. For example, to address the situation described in the previous paragraph, a fetch interval could be constructed for the "slow" ID to avoid transferring over redundant records. A SQL query like the following could be constructed:

```
SELECT *
FROM source
WHERE id = 1
  AND datetime >= '2021-01-01 00:00:00'::TIMESTAMP
  AND datetime <= '2021-02-01 00:00:00'::TIMESTAMP
```

Furthermore, separate intervals may be designated for multiple IDs, meaning a discontinuous sample may be constructed. This sample may be created by several methods; multiple simple passes could be executed (one query per ID), sub-queries could be combined (via the `UNION ALL` keyword), a single query could augment multiple queries through logic in the `WHERE` clause, and a single query could augment multiples queries by joining on a temporary table of datetimes and IDs.

The approach for executing smaller, distinct queries per ID yields several advantages:

1. Each individual query is "easy" to execute.

   The simple logic of each query allows the source database to optimize searching across the table's indices.

2. The time between queries may be spent on other operations.

   This time allows the potentially sensitive source database to "cool down."

3. Fetching and filtering may happen in parallel.

   While the syncing service is filtering two samples, more samples may be concurrently fetched.

4. An ongoing synchronization may be paused and resumed.

   A verification synchronization may be extended over a long period of time to disperse the load on the databases.

However, executing multiple queries introduces disadvantages as well:

1. The source table might change between queries.

   Because the table is not locked between queries, data may be malformed.

2. This synchronization may take longer to execute.

   Although fetching and filtering in parallel will reduce execution time, another query may lock the source table, halting the ongoing synchronization process.

3. It does not allow the database to fully optimize the request.

   Execution engines can reduce processing time when given the full context of the query.

4. It could overwhelm the databases.

   If not throttled appropriately, an onslaught of queries could overload the databases' active connections.

Below is a simple pseudocode representation of a scheduler for executing queries per ID. The logic of the *getStartTime*() and *getEndTime*() functions depends on the circumstances of the specific data stream. One possible solution may simply return

the same values for every ID, or a more specialized implementation could further constrain the intervals.

procedure SYNCHRONIZE($Source$, $Target$)
  for all $id \in Source$ **do**
    $StartTime \leftarrow$ GETSTARTTIME($Source, id$)
    $EndTime \leftarrow$ GETENDTIME($Source, id$)
    $SourceSample \leftarrow$ FETCH($Source, StartTime, EndTime, id$)
    $TargetSample \leftarrow$ FETCH($Target, StartTime, EndTime, id$)
    $FilteredSample \leftarrow$ FILTER($SourceSample, TargetSample$)
    INSERT($Target$, $FilteredSample$)

A dedicated scheduler offers more control over how specific portions of the pipe are synchronized. However, because the synchronization of pipes are themselves managed by a scheduler in the Meerschaum system, adding an additional layer of iteration introduces unnecessary complexity into the overall system. Splitting a pipe into smaller sub-pipes and joining them after the fact would offer many of the same benefits described above without nesting schedulers.

Three methods for mimicking multiple queries in a single transaction are (1) appending sub-queries (per subsection 3.1.4), (2) augmenting sub-queries in the `WHERE` clause, and performing a `LEFT OUTER JOIN` on IDs and datetimes (per subsection 3.1.5). For example, the following SQL query performs the same functionality as the scheduler defined above, but rather than filtering and fetching in parallel, it instead locks the table and returns all of the rows at once.

```
SELECT *
FROM source
WHERE id = 1
  AND datetime >= '2021-01-01 00:00:00'::TIMESTAMP
  AND datetime <= '2021-01-01 06:00:00'::TIMESTAMP
UNION ALL
SELECT *
FROM source
WHERE id = 2
  AND datetime >= '2021-01-01 09:00:00'::TIMESTAMP
  AND datetime <= '2021-01-01 18:00:00'::TIMESTAMP
```

The following SQL query performs the same function but with the logic contained within the `WHERE` clause. Instead of executing multiple sub-queries and combining the result at the end, the intervals are defined as a series of statements joined by `OR` keywords, allowing the databases' execution engines to optimize the entire context of the query.

```
SELECT *
FROM source
WHERE (
  id = 1
    AND datetime >= '2021-01-01 00:00:00'::TIMESTAMP
    AND datetime <= '2021-01-01 06:00:00'::TIMESTAMP
) OR (
  id = 2
    AND datetime >= '2021-01-01 09:00:00'::TIMESTAMP
    AND datetime <= '2021-01-01 18:00:00'::TIMESTAMP
)
```

Finally, the following SQL query demonstrates how to achieve the same logic by joining on a temporary table of datetimes and IDs (similar to the approach taken in *Simple Join Sync* in subsection 3.1.5).

```
WITH definition AS (
  SELECT *
  FROM source
), sync_times AS (
  SELECT *
  FROM (
    VALUES
      (1, '2021-01-01 00:00:00'::TIMESTAMP, '2021-01-01 06:00:00'::TIMESTAMP),
      (2, '2021-01-01 09:00:00'::TIMESTAMP, '2021-01-01 18:00:00'::TIMESTAMP)
  ) AS t(id, begin, end)
) SELECT d.*
FROM definition AS d
LEFT OUTER JOIN sync_times AS st
  ON st.id = d.id
WHERE (
  d.datetime >= st.begin
  AND
  d.datetime <= st.end
) OR st.id IS NULL
```

The following figure demonstrates a scenario where two distinct intervals are required.

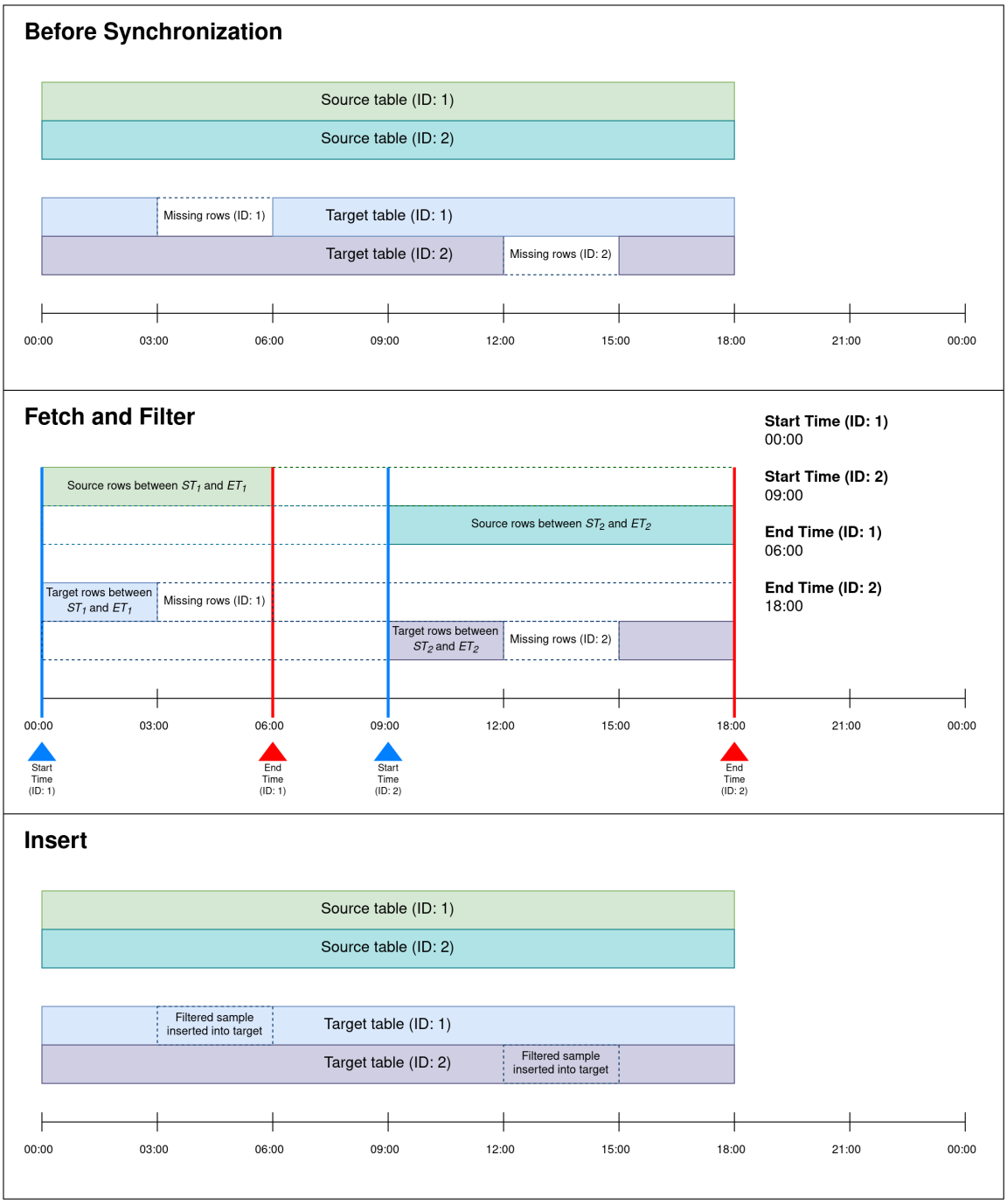

Figure 2.3: A sample may be fetched from discontinuous intervals by querying on both the ID and datetime indices.

2.2.3 Unknown Backlogged Data

A more realistic scenario for backlogged data is a data stream with an unknown backlog interval. The methods to address "unknown" scenarios present trade-offs which rely upon the priorities of the end user:

Among run-time, accuracy, and bandwidth, which is most important?

The fast-and-loose strategies presented for the previous scenarios (e.g. *Simple Backtrack Sync*) typically perform well in terms of run-time and bandwidth due to their limited capacity to traverse "into the past" to detect backlogged rows. For cases such as synchronizing large volumes of utilities data where excluding some rows is not mission critical, the simple strategies are acceptable. But in scenarios where data are frequently backlogged or accuracy is a top priority (such as when handling financial data), then methods to find overlooked records require consideration.

Introducing *Iterative Syncs*

To accurately synchronize tables where the extent of backlogging is unknown, the entire interval of the pipe must be synchronized. The most accurate and straightforward approach would be to drop the target table and duplicate the source (i.e. *Naïve Sync*), but a class of synchronization strategies called *Iterative Syncs* (section 3.2) maximizes accuracy by traversing the datetime axis to detect backlogged rows. *Iterative Syncs* depend on the following assumptions:

1. The data stream has a datetime index.
2. Rows are immutable.
3. Rows with later datetime values are higher priority than those with older datetime values.

In section 2.2 when discussing known backlog intervals, a modified version of *Simple Backtrack Sync* (subsection 1.3.1) demonstrated how to synchronize a known interval bounded by start and end times (ST and ET). Without a known interval, the interval becomes "all time," and executing an unbounded synchronization would be less efficient than the baseline naïve method of duplicating the source table.

Not only would performing one large synchronization unnecessarily impose significant stress on the source database and network, the procedure would lock up the table, and the fetched rows would not be available in the target until the synchronization had completed. In the context of the assumptions stated above, rows from only a few days ago would be of greater interest than rows from years ago (if this were not the case, then an explicit interval could be stated per subsection 2.2.1).

To distribute the server's load over time and quickly synchronize "priority" rows, a series of small synchronizations may be executed across the pipe's entire datetime interval. The algorithm described below and in Figure 2.2.3 demonstrates how to iterate across the datetime axis and perform a series of bounded synchronizations (introduced in subsection 2.2.1) to synchronize unknown backlogged data.

1. **Determine the initial, newer "reference time" (RT_0) datetime.**

   If not provided, RT_0 will be the newest target datetime value.

2. **Determine the next, older "reference time" (RT_1) datetime.**

   If not provided, RT_1 will be the oldest target datetime value.

3. **Determine the first "backtrack interval" (BTI_1).**

   If not provided, BTI_1 will be a default value of 1 hour.

   BTI_1 has the subscript 1 because the first backtrack interval is used during the second iteration (where $i = 1$).

4. **Synchronize rows newer than RT_0.**

   For the first iteration ($i = 0$), consider RT_0 as the "start time" (ST_0) and fetch rows newer than ST_0.

5. **While $ST_i > RT_1$, set values for ST_i, ET_i, and BTI_i and synchronize the intervals.**

   (a) Grow BTI_i according to a $growBTI()$ function.

   By default, $growBTI()$ scales BTI_i by 40% ($BTI_i = 1.4(BTI_{i-1})$). The value used in $fetch()$ is rounded and capped at a maximum value (768 hours or 32 days by default).

   (b) Set ET_i to the value of the previous "start time" (ST_{i-1}).

   (c) Set ST_i to ET_i minus BTI_i.

   If ST_i is less than RT_1, set ST_i to RT_1.

   (d) If rows may be counted prior to fetching, skip identical partitions.

   If possible, check the the number of rows in the defined interval (e.g. synchronizing via SQL), otherwise simply fetch and filter the interval (e.g. synchronizing via a simple bounded interface like a web API).

   Intervals with the same number of rows may be skipped only if the data are immutable.

6. **Synchronize rows older than RT_1.**

   For the final iteration, consider RT_1 as the last ET and fetch rows older than this ET.

function GROWBTI(BTI, $maxBTI$)

    return MIN($1.4 * BTI$, $maxBTI$)

procedure SYNCHRONIZE($Source$, $Target$, RT_0, RT_1, $BTI_1, maxBTI$)

    if RT_0 is $None$ **then**

        $RT_0 \leftarrow$ GETNEWESTTIME($Target$)

    if RT_1 is $None$ **then**

        $RT_1 \leftarrow$ GETOLDESTTIME($Target$)

    if BTI_1 is $None$ **then**

        $BTI_1 \leftarrow$ 1 hour

    if $maxBTI$ is $None$ **then**

        $maxBTI \leftarrow$ 768 hours

    $ST_0 \leftarrow RT_0$

    $SourceSample_0 \leftarrow$ FETCH($Source, ST_0$)

    $TargetSample_0 \leftarrow$ FETCH($Target, ST_0$)

    $FilteredSample_0 \leftarrow$ FILTER($SourceSample_0, TargetSample_0$)

    INSERT($Target$, $FilteredSample_0$)

    $i \leftarrow 1$

    while $ST_i < RT_1$ **do**

        if $i \neq 1$ **then**

            $BTI_i \leftarrow$ GROWBTI($BTI_{i-1}, maxBTI$)

        $ET_i \leftarrow ST_{i-1}$

        $ST_i \leftarrow ET_i -$ ROUND(BTI_i)

▷ *If the current context allows for remotely counting rows (e.g. executing SQL statements), check the number of rows and only sync when different interval counts are returned.* ◁

if GETROWCOUNT($Source, ST_i, ET_i$) = GETROWCOUNT($Target, ST_i, ET_i$) **then**

  $i \leftarrow i + 1$

  Continue to the next loop iteration

$SourceSample_i \leftarrow$ FETCH($Source, ST_i, ET_i$)

$TargetSample_i \leftarrow$ FETCH($Target, ST_i, ET_i$)

$FilteredSample_i \leftarrow$ FILTER($SourceSample_i, TargetSample_i$)

INSERT($Target$, $FilteredSample_i$)

$i \leftarrow i + 1$

$ST_{final} \leftarrow RT_{final}$

$SourceSample_{final} \leftarrow$ FETCH($Source, None, ST_{final}$)

$TargetSample_{final} \leftarrow$ FETCH($Target, None, ST_{final}$)

$FilteredSample_{final} \leftarrow$ FILTER($SourceSample_{final}, TargetSample_{final}$)

INSERT($Target$, $FilteredSample_{final}$)

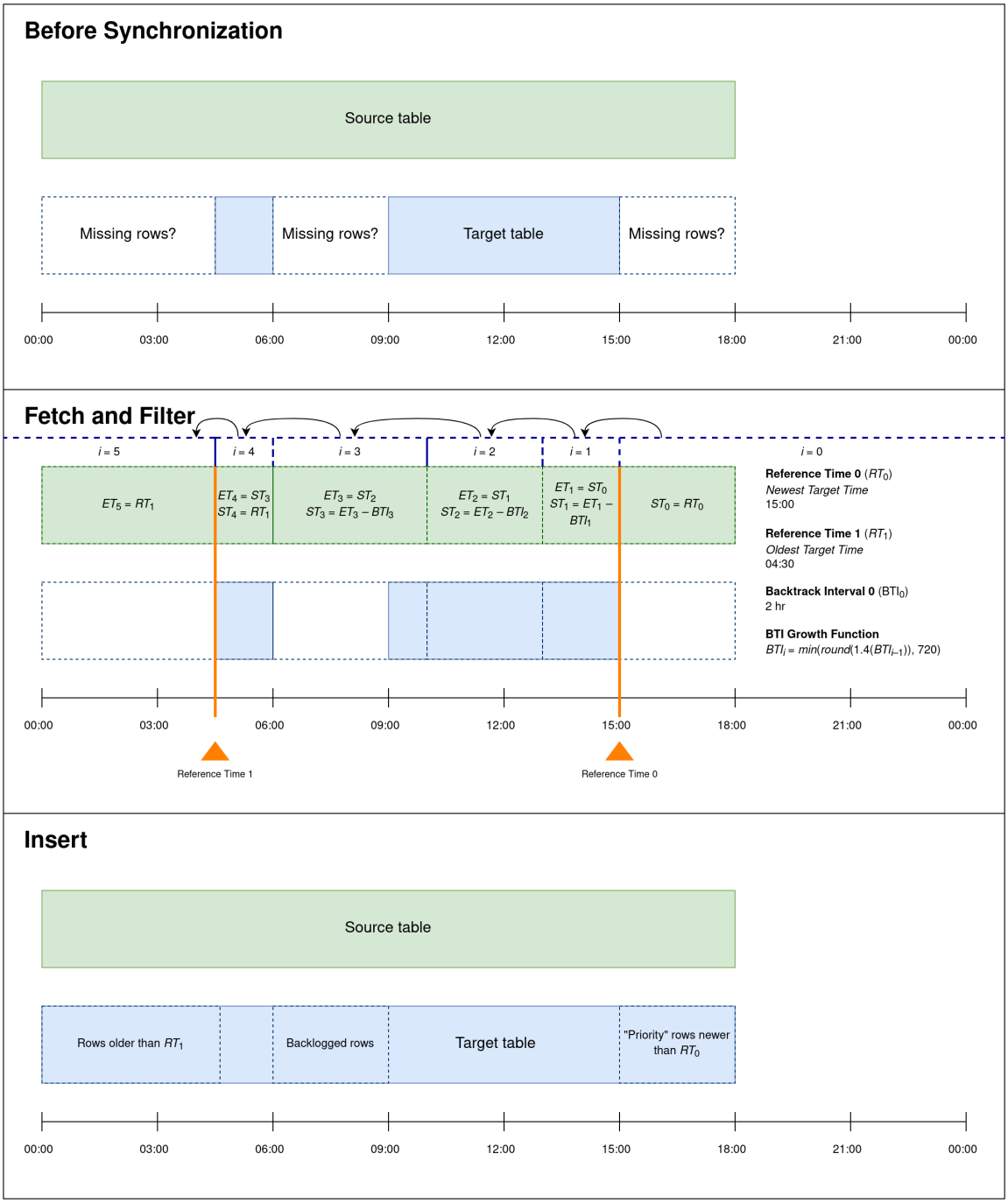

Figure 2.4: Many small synchronizations can be executed to iteratively detect unknown backlogged rows.

Chapter 3

Strategies

The possibilities for target-based synchronization strategies are potentially limitless, but to address the scenarios outlined in chapter 2, the strategies may be grouped into one of three classes: (1) **simple syncs**, (2) **iterative syncs**, and (3) **corrective syncs**. The intended goals of each algorithm vary depending on the use case, but in general, the behaviors of these strategies are designed with respect to processing time, bandwidth, and accuracy. Comparative results against these metrics are presented in chapter 4.

3.1 Speed-First: *Simple Syncs*

The *Simple Syncs* class of synchronization strategies prioritizes run-time and bandwidth over accuracy due to the shared behavior of focusing on "new" rows in the future. The strategies within *Simple Syncs* vary in the metrics of run-time, bandwidth, and accuracy, but as a whole, they perform best when backlogging is rare.

3.1.1 *Simple Sync*

Simple Sync is characterized by setting a ST and ET, fetching source and target samples within the bounds, and inserting the difference. The default behavior of *Simple Sync* uses the newest target datetime value as the ST, thereby always fetching the "newest" rows. This approach "covers" every value on the datetime axis exactly once. Depending on the probability of backlogging, the algorithm may yield an accurate result, but for the most part, *Simple Sync* trades accuracy for run-time and bandwidth.

3.1.2 *Simple Backtrack Sync*

Simple Backtrack Sync behaves very similarly to *Simple Sync* with the key distinction that the ST is offset by a BTI to widen the bounds for which rows are fetched. This variation always performs worse in bandwidth than *Simple Sync* because the fetch window is always larger. However, especially in situations where rows are frequently backlogged near the newest target datetime value (subsection 2.2.1), this backtracking may potentially increase the accuracy over *Simple Sync* without significantly sacrificing run-time.

3.1.3 *Simple Slow-ID Sync*

Simple Slow-ID Sync takes a similar approach to *Simple Backtrack Sync* except that the mechanism to determining the ST relies on the ID column of the target table. The ST is set as the oldest of the newest datetime values per ID, such as the following example:

```
WITH sync_times AS (
  SELECT id, MAX(datetime) AS sync_time
  FROM source
  GROUP BY id
) SELECT MIN(sync_time)
FROM sync_times
```

This method for determining the ST effectively results in a dynamic BTI. In scenarios where IDs regularly "lag behind" the newest IDs, *Simple Slow-ID Sync* may offer higher accuracy than *Simple Backtrack Sync*. One serious caveat to *Simple Slow-ID Sync* is that once an ID "dies," the BTI grows larger with time — performance gradually declines and effectively becomes *Naïve Sync*.

3.1.4 *Simple Append Sync*

Simple Append Sync was addressed in subsection 2.2.2 as a way to augment many small SQL queries in one transaction. The method combines the accuracy benefits of *Simple Slow-ID Sync* while circumventing the "dead ID" problem. The approach consists of performing *Simple Sync* per ID and appending the queries together with `UNION ALL`. As the number of IDs increases, the $fetch()$ query of *Simple Append Sync* grows and run-time performance declines.

3.1.5 *Simple Join Sync*

The end result of *Simple Join Sync* is the same as *Simple Append Sync*: performing *Simple Sync* on a per-ID basis within a single transaction. Rather than appending together many sub-queries, *Simple Join Sync* performs a `LEFT OUTER JOIN` on a temporary table to determine the ST for each ID. A typical $fetch()$ query for *Simple Join Sync* may look like the following:

```
WITH sync_times AS (
  SELECT *
  FROM (
    VALUES
      (1, '2021-01-01 00:00:00'::TIMESTAMP),
      (2, '2021-01-01 09:00:00'::TIMESTAMP)
  ) AS t(id, begin)
) SELECT source.*
FROM source
LEFT OUTER JOIN sync_times
  ON source.id = sync_times.id
WHERE (
  source.datetime > sync_times.begin
  OR sync_times.id IS NULL
)
```

Like *Simple Append Sync*, *Simple Join Sync* grows in complexity as the number of IDs increases. The trade-off for simple ID-focused strategies like *Simple Join Sync* is to save bandwidth in exchange for a bit of additional run-time. Depending on the costs of bandwidth and processing time, this approach would cease to be advantageous when the number of IDs grows too large.

3.2 No-Compromises Accuracy: *Iterative Syncs*

The strategies in the *Iterative Syncs* class take the opposite approach from *Simple Syncs*; guaranteeing an accurate sync comes before run-time and bandwidth. First introduced in subsection 2.2.3, *Iterative Syncs* traverse the entire datetime axis to locate missing rows. This is not unlike the partitioning and hashing strategies used by Ahluwalia et al. [2010], Choi et al. [2010], and Tang et al. [2010], except that row-counts are used in place of hashing because the tables are assumed to be immutable.

Strategies from the *Iterative Syncs* class will be appealing to users for whom accuracy is a non-negotiable requirement. Each strategy below may be *bounded* or *unbounded*, meaning that the duration of the iteration may be limited to a maximum

interval. In situations where backlogging far "into the past" is unlikely, bounding the search interval decouples the run-time performance from size of the underlying tables with the caveat that the synchronization technically does not guarantee perfect accuracy.

3.2.1 *Iterative Simple Sync*

Iterative Simple Sync mostly follows the behavior of the example iterative algorithm described in subsection 2.2.3: for each partition of the datetime axis, compare row-counts and fetch samples when row-counts differ. This approach is specifically denoted as "simple" because it does not contain any additional mechanisms to tighten the intervals surrounding missing rows. After accuracy, this method prefers run-time to bandwidth as it attempts to complete the synchronization in fewer transactions than the other iterative approaches.

3.2.2 *Daily Row-Count Sync*

Although included within *Iterative Syncs*, *Daily Row-Count Sync* does not iterate in the same way as the other iterative strategies. Instead, *Daily Row-Count Sync* builds tables of days' row-counts and performs *Simple Sync* on days with differing row-counts (akin to *Simple Join Sync*). To bound the search, the earliest datetime value is included in the `WHERE` clauses. Consider the following example of the query first executed on the source and target tables to determine which days require synchronization:

```
SELECT
  DATE_TRUNC('day', datetime) AS days,
  COUNT(*) AS rowcount
FROM table
WHERE datetime > '2021-01-01 00:00:00'::TIMESTAMP
GROUP BY days
```

3.2.3 *Binary Search Sync*

Binary Search Sync combines the approaches of *Iterative Simple Sync* and *Daily Row-Count Sync* for determining which intervals need to be synchronized. The algorithm traverses the datetime axis like other iterative approaches, calculating source and target row-counts for each partition. When different row-counts are detected, it recursively performs a binary search, comparing sub-intervals' row-counts until a sufficiently small interval is encountered (1 day to be comparable to *Daily Row-Count Sync*).

Binary Search Sync executes more transactions than *Iterative Simple Sync* and *Daily Row-Count Sync*, but the end result is that fewer rows are transferred than *Iterative Simple Sync* due to identifying specific days, and the databases do not need to calculate row-counts for every single day like is the case with *Daily Row-Count Sync*[1]. Therefore, *Binary Search Sync* may offer the same bandwidth savings as *Daily Row-Count Sync* without as much overhead in cases where backlogging is limited.

3.2.4 *Iterative CPISync*

Iterative CPISync is inspired by *TBDS* [Ahluwalia et al., 2010] (section 1.2.1), Partitioned-*CPISync* [Tang et al., 2010] (section 1.2.2), and *P-CPI* [Jin et al., 2012] (section 1.2.2) with the primary distinction that *Iterative CPISync* takes advantage of the properties of immutable time-series data.

As discussed in subsection 1.2.2, *CPISync* performs best when the number of differences between sets is small (lower $\bar{m}$ is better). *CPISync* effectively performs $filter()$ before $fetch()$, so bandwidth performance is near-optimal when maintaining

[1]Because the datetime axis is assumed to be indexed, this difference may be negligible, and the increased number of transactions would overshadow any potential savings

perfect accuracy. However, these bandwidth savings come at the cost of longer run-times. For example, characteristic polynomials must be evaluated on the source and target databases, and if the source is a critical production database, then *CPISync* may be infeasible. Additionally, the run-time performance of *Iterative CPISync* degrades as backlogging becomes more frequent. When iterating across the datetime axis (per subsection 2.2.3), the number of differences between the partitions ($\bar{m}$) corresponds to the difference in row-counts, and when the number of missing rows is high, then a *Simple Sync* is more practical.

A few considerations need to be kept in mind to address the limitations of *Iterative CPISync*:

1. *CPISync* fundamentally works with integers (technically any rational values may be used, but working with integers makes calculating over a Galois finite field significantly easier and more performant). Therefore, rows must be mapped to integers.

2. *CPISync* can be computationally demanding if the range of values is large. Rows could be mapped to integers with something like a hash function, but the resulting range would be much too large, so instead the datetime value alone is used to map to an integer.

3. To minimize the range of integers, individual IDs must be synchronized separately. A composite integer could be constructed by appending an integer ID to the Unix timestamp of the datetime value of a row, but caveats such as string IDs and large IDs complicate this possibility.

4. An acceptable interval size depends on the data set's temporal resolution. For data streams with 1-second resolution, a regular Unix timestamp may be used.

Increasing the resolution inversely decreases the interval size, so data streams with nanosecond precision would have intervals 1-billionth the size of a 1-second stream.

The number of out-of-sync rows in an interval determines $\bar{m}$ for that iteration of *CPISync*. Analogs to $\bar{m}$ for this algorithm are the frequency of the source data stream and its temporal resolution. If a data stream has a known, fixed frequency, then multiples of the frequency may be sequentially numbered — for a resolution of one record per 15 minutes, the timestamps `00:00`, `00:15`, `00:30`, and `0:45` would be numbered `1`, `2`, `3`, and `4`. In case the resolution is not fixed, the upper bound of the frequency may be assumed to be the temporal resolution (maximizing the number of possible rows in an interval). One approach of limiting the assumed frequency is to determine the minimum interval between rows of an interval in the source database. This will introduce overhead to determine the intervals' frequencies but will save processing time when evaluating $\bar{m}$ evaluation points Z_k.

For example, a data stream with a frequency of one record per 15 minutes and a 1-second temporal resolution may have a maximum of 96 rows per day. This would dramatically speed up *Iterative CPISync* by limiting the range of possible values. However, limiting timestamps into known intervals is a tight restriction which would rule out *Iterative CPISync* from most use-cases, and determining a minimum resolution may prove more complicated than using a fixed resolution. Therefore, for this thesis, the implementation of *Iterative CPISync* uses a 1-second resolution, and the Unix timestamps of the datetime values are subtracted from the beginning datetime to limit the range of integers (and the next largest prime above three time the largest determined value is used to limit calculations to a finite field).

Consider the following example of how a characteristic polynomial for the Z

value -1 may be evaluated in SQL (where `1609459200` is the Unix timestamp for midnight of January 1, 2021 UTC and `494101` is the chosen prime number for the finite field greater than three-times the number of seconds in the interval).

```
WITH RECURSIVE t(c) AS (
  SELECT (-1 - EXTRACT(EPOCH FROM datetime) - 1609459200)::BIGINT
  FROM table
  WHERE id = 1
    AND datetime >= '2021-01-01 00:00:00'::TIMESTAMP
    AND datetime < '2021-01-02 00:00:00'::TIMESTAMP
), r(c, n) AS (
  SELECT t.c, row_number() OVER ()
  FROM t
), p(c, n) AS (
  SELECT c, n
  FROM r
  WHERE n = 1
  UNION ALL
  SELECT (r.c * p.c) % 494101, r.n
  FROM p
    JOIN r ON p.n + 1 = r.n
) SELECT c
FROM p
WHERE n = (
  SELECT MAX(n)
  FROM p
)
```

3.3 Best of Both Worlds: *Corrective Syncs*

The first two classes of strategies were designed to prioritize certain metrics: *Simple Syncs* for run-time and bandwidth and *Iterative Syncs* for accuracy. In situations where accuracy is valued but some levels of inaccuracy are tolerated, *Corrective Syncs* seek to balance run-time, bandwidth, and accuracy. For "everyday" synchronizations, *Corrective Syncs* perform a lightweight synchronization like *Simple Sync* which may accumulate errors, and more expensive iterative synchronizations are performed intermittently (e.g. monthly) to locate missing rows and "correct" the target table. Because the tables grow continuously, the accuracy rate will trend upwards over time.

3.3.1 *Simple Monthly Naïve Sync*

Simple Monthly Naïve Sync combines the simplest opposite strategies: *Simple Sync* and *Naïve Sync*. The strategy employs *Simple Sync* daily and performs a *Naïve Sync* each month. This method drastically improves performance over daily *Naïve Sync* but is still subject to its scaling issues. In situations where the tables are not too large and bandwidth is cheap, *Simple Monthly Naïve Sync* is an easy way to regularly "flush" the pipes.

3.3.2 *Simple Monthly Iterative Simple Sync*

Simple Monthly Iterative Simple Sync performs a *Simple Sync* daily and an *Iterative Simple Sync* monthly. The overall approach is that the datetime axis is always pushed "into the future," and each point on the axis "in the past" is searched once per month. This strategy shares behavior with *Iterative Simple Sync* but disperses it over time. The *bounded* variant of *Simple Monthly Iterative Simple Sync* only backtracks the previous month, and in cases where backlogging more than one month is unlikely, it is likely to offer adequate run-time performance.

3.3.3 *Simple Monthly Daily Row-Count Sync*

Like the other corrective strategies, *Simple Monthly Daily Row-Count Sync* performs daily *Simple Syncs* and verifies monthly with an iterative strategy, in this case *Daily Row-Count Sync*. The benefit to using *Daily Row-Count Sync* intermittently is that the backtracking may be bounded and all of the row-counts evaluated within a single query. The bounded variant may also be easily adjusted without significantly altering the iteration behavior (e.g. changing the row-counts interval or beginning datetime boundary).

3.3.4 *Simple Monthly Binary Search Sync*

As discussed above in subsection 3.2.3, both *Binary Search Sync* and *Daily Row-Count Sync* first locate which days contain missing rows and only differ in the mechanisms by which these days are determined. Therefore, the comparative performance between *Simple Monthly Binary Search Sync* and *Simple Monthly Daily Row-Count Sync* should be similar to the relationship between *Binary Search Sync* and *Daily Row-Count Sync* with the primary distinction that the iterative approaches are dispersed monthly.

3.3.5 *Simple Monthly Iterative CPISync*

Finally, *Simple Monthly Iterative CPISync* promises attractive bandwidth savings without detrimental run-time performance by combining the fast and bandwidth-saving (but potentially inaccurate) *Simple Sync* with the slow but bandwidth-saving *Iterative CPISync*. This strategy may prove to be a practical alternative to *Iterative CPISync* in situations where bandwidth comes at a premium and accuracy does not need to be completely guaranteed at every synchronization.

Chapter 4

Experimental Results

To test the performance of the synchronization strategies described in chapter 3, the Meerschaum plugin `syncx` was written to simulate the scenarios from chapter 2 [Meares, 2021b].

Performance Metrics

Each simulation increases a source table by one record per ID per hour over the course of a year, and a target table is synchronized with the source table each day according to a strategy. Metrics are collected to compare the methods in three areas:

1. **Run-time** – The duration in seconds of each synchronization.
2. **Bandwidth** – The number of rows fetched from the source database.
3. **Accuracy** – The ratio of the number of correctly synchronized rows to the number of all source rows.

Figure Interpretation

The following figures are presented to visualize the strategies' simulation results.

1. **Daily metrics line graphs** — The daily line graphs represent the strategies' daily performances to illustrate their behaviors. For example, the daily metrics line graphs portray the scaling issues of *Naïve Sync* and display the intermittent verification syncs of *Corrective Syncs*.

2. **Summary bar charts** — The bar charts compare the aggregated values of the daily line graphs: the total number of seconds, total number of rows fetched, and average accuracy rate. This allows for direct comparisons of specific metrics, i.e. "lower is better" for the total run-time and rows fetched and "higher is better" for the average accuracy rate.

3. **Summary radar charts** — The radar charts are structured as intuitive visual representations of the "skills" of each technique such that "higher is better" for every metric; the accuracy axis is the same as the bar chart's, and run-time and bandwidth are normalized to a scale between the performance of *Simple Sync* and fifteen-times worse performance[1].

4. ***Choice Index* bar charts** — Introduced in section 4.2, the *Choice Index* is the weighted average of the normalized values presented in the summary radar charts. The bar charts offer a 3x3 grid of rankings organized by prioritized metrics. The rankings allow for consideration of all three metrics when choosing strategies.

[1]This scale was chosen to clearly distinguish the "best" and "worst" strategies.

Tested Scenarios

Each strategy was tested against four scenarios, although not every combination is included for brevity. Consult the `syncx` repository for the complete set of results.

1. *single-append-only* (subsection 2.1.1)

   A single ID reporting hourly with a 0% "outage" probability.

2. *multiple-large-n-append-only* (subsection 2.1.2)

   Many IDs (100) reporting hourly with a 0% "outage" probability.

3. *single-known-backlog* (subsection 2.2.1)

   A single ID reporting hourly with a 10% "outage" probability. Records which are generated during an "outage" are backlogged within a known interval (24 hours).

4. *unknown-backlog* (subsection 2.2.3)

   An unknown number of IDs (3) reporting at an unknown frequency (hourly) with an unknown "outage" probability (10%). Backlogged rows are inserted over an unknown interval (priority queue where a random number of records are backlogged per iteration).

Technical Context

The following simulations and calculations were executed on a single core of an Intel i7-4790 running at 3.901GHz with 32 gigabytes of available memory. The simulations used an in-memory instance of `duckdb` [Raasveldt and Mühleisen, 2020], on which rows consisting of a timestamp, integer, and float (as described in subsection 2.1.1) were regularly inserted into the source tables.

4.1 Comparing Classes of Strategies

This section compares the relative performances of strategies within each class (*Simple*, *Iterative*, and *Corrective Syncs*) to identify nuances in performances. Comparisons made across these boundaries for the purposes of metric optimizations are presented in section 4.2.

4.1.1 Establishing a Baseline: *Simple Sync* vs. *Naïve Sync*

The strategies *Simple Sync* and *Naïve Sync* are contrasted to provide context for the later combinations. As expected, *Naïve Sync* exhibits abysmal performance in run-time and bandwidth, with a perfect accuracy rate as its only redeeming quality.

Because *Naïve Sync* includes every previously fetched record for each synchronization, run-time and bandwidth complexity scales linearly ($O(n)$), thereby squaring the cumulative rows fetched ($O(n^2)$). Similar performances to those exhibited in Figure 4.1 were observed across all of the scenarios tested.

Inclusion of datetime boundaries ST and ET decouples *Simple Sync* from the underlying tables such that it exhibits constant complexity ($O(1)$) and reduces the cumulative row-count to its optimal linear reality ($O(n)$). Of course additional unaccounted latency may increase the daily run-time as n grows large (i.e. data gravity), but partitioning schemes like TimescaleDB's hyper tables may serve to mitigate any additional latency.

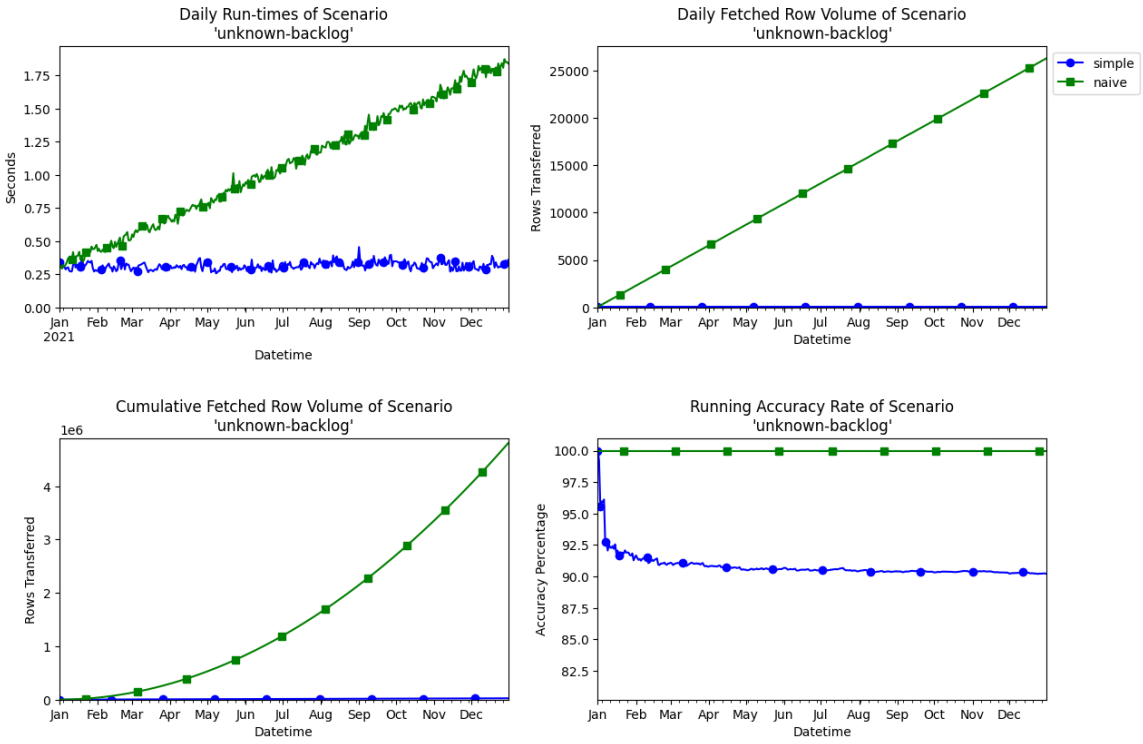

Figure 4.1: The daily performances of *Simple Sync* and *Naïve Sync* for the scenario *unknown-backlog*
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

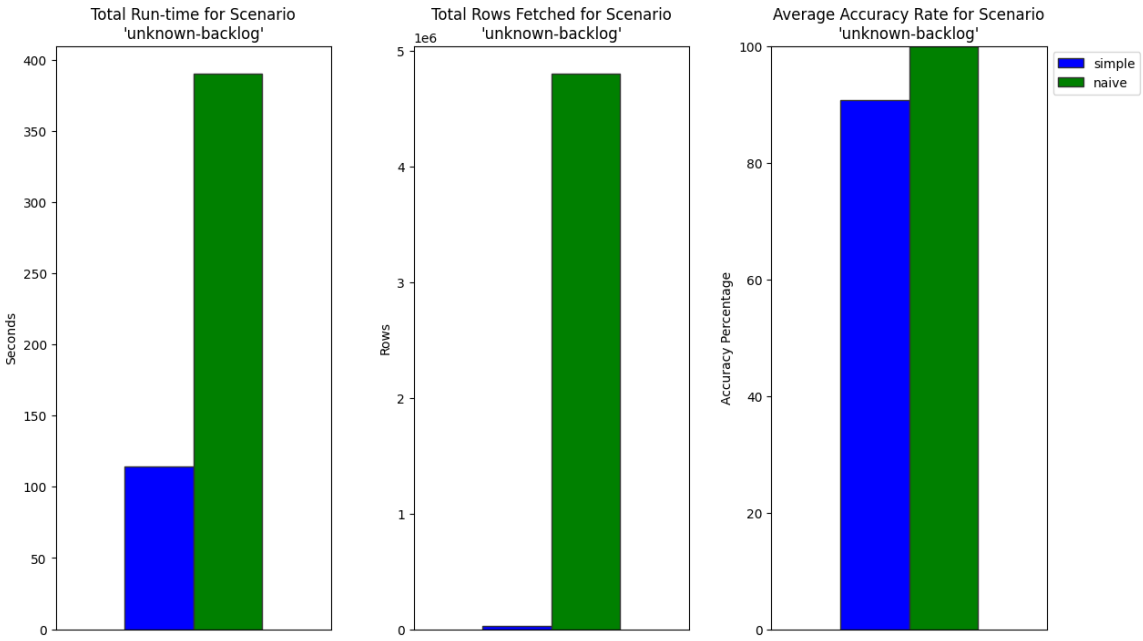

Figure 4.2: *Simple Sync* requires significantly less bandwidth than *Naïve Sync*. *"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.*

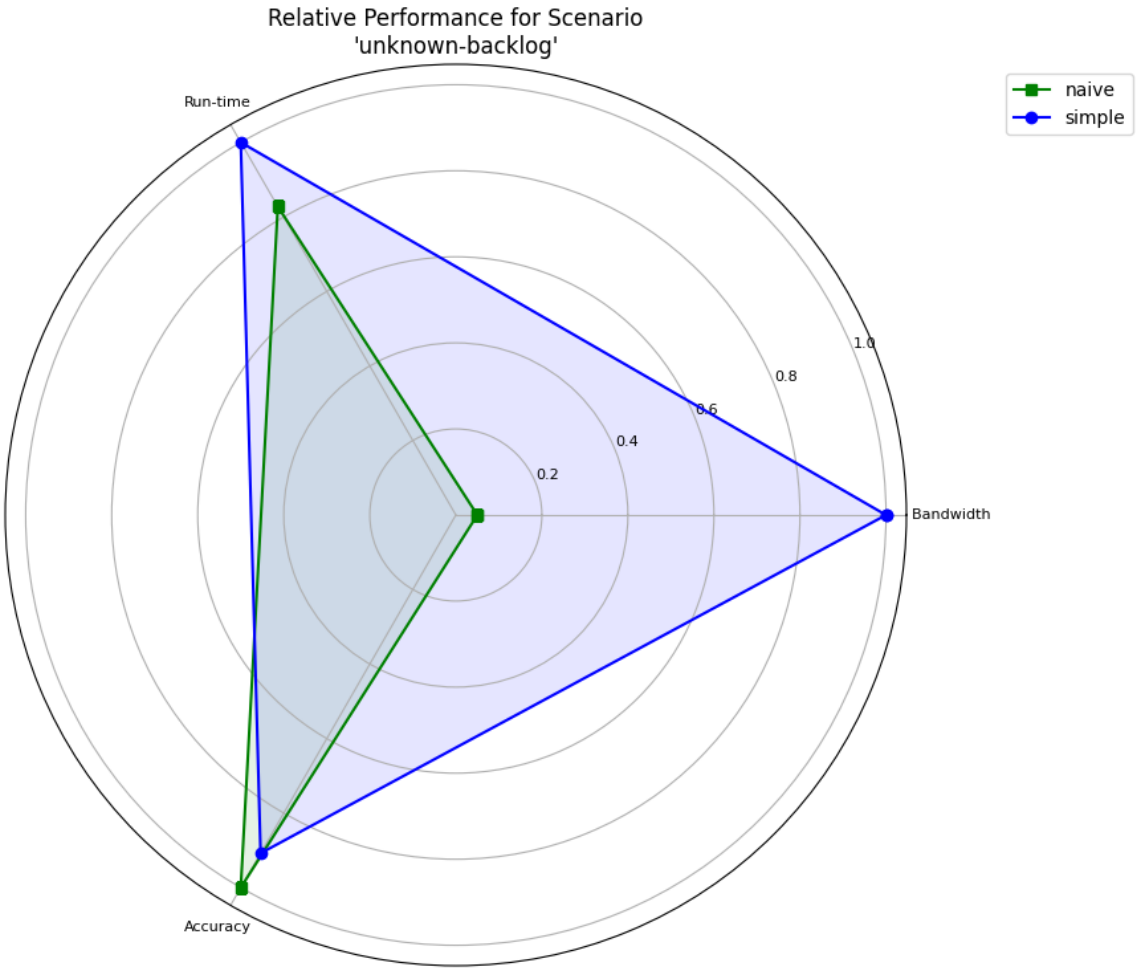

Figure 4.3: *Simple Sync* outperforms *Naïve Sync* in run-time and bandwidth but not accuracy.
"Higher is better" for all metrics.

4.1.2 Comparing *Simple Syncs*

Simple Sync is utilized as a benchmark strategy due to its straightforward design and admirable performance. As described in section 3.1, variations of the *Simple Sync* approach are designed to improve its accuracy and bandwidth without significantly sacrificing its run-time. Due to their common roots, the performances of the *Simple Syncs* are mostly comparable with a few noteworthy observations.

Best in Bandwidth: *Simple Join Sync* and *Simple Append Sync*

First, the ID-focused variants (*Simple Append Sync* and *Simple Join Sync*) slightly improve bandwidth at the expense of run-time (see Figure 4.4). The run-time degradation is most significant with a large number of IDs — when syncing 100 IDs, the daily synchronizations takes approximately 400% longer than *Simple Sync* in exchange for a 4% reduction on bandwidth (2475 versus 2376 rows per day). In situations with many IDs reporting frequently, the potential bandwidth savings are significant. Accuracy is not affected because no amount of backtracking takes place.

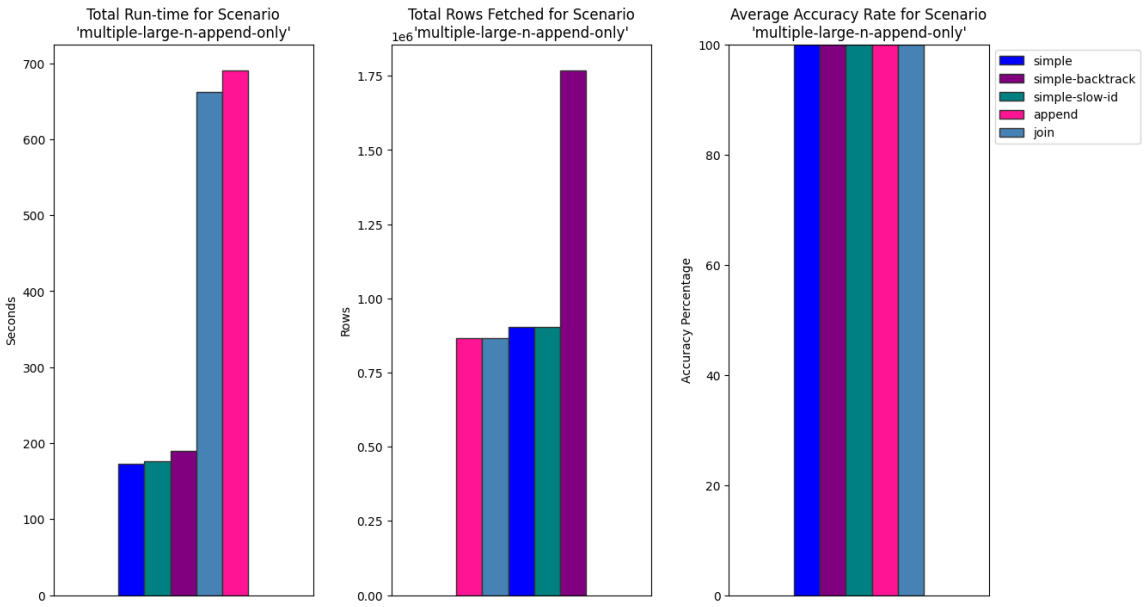

Figure 4.4: The ID-focused *Simple Syncs* slightly reduce the bandwidth of *Simple Sync* but suffer significantly as the number of IDs scales.
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

Best in Accuracy: *Simple Backtrack Sync*

Second, the BTI-focused variants (*Simple Backtrack Sync* and *Simple Slow-ID Sync*) exhibit expected behavior of diminished run-time and bandwidth in exchange for improvements in accuracy. However, the improved accuracy is negligible or greatly diminished in situations like *unknown-backlog*, but in the situation *single-known-backlog* where backlogged records (10% of all rows) are inserted within 24 hours, the strategy *Simple Backtrack Sync* inversely trades bandwidth for accuracy with no noticeable effect on run-time.

For example, *Simple Backtrack Sync* with a BTI of 1 hour demonstrates a 0.35% accuracy improvement at approximately a 4% increase in bandwidth. This intuitively makes sense because the 1-hour BTI only "catches" $\frac{1}{24}$ of the backlogged records, so 4% of the 10% backlogged records is an improved accuracy of 0.4%. When the BTI is increased to 24 hours, accuracy improves to nearly 100% at a bandwidth cost of 200% (slight discrepancies may be due to programming oversights; see Figure 4.5). Therefore, the daily run-time complexity remains constant ($O(1)$), the daily fetched row volume scales as a function of the BTI size ($O(m)$), and the cumulative fetched row volume scales linearly with an increased slope according to the BTI size ($O(mn)$).

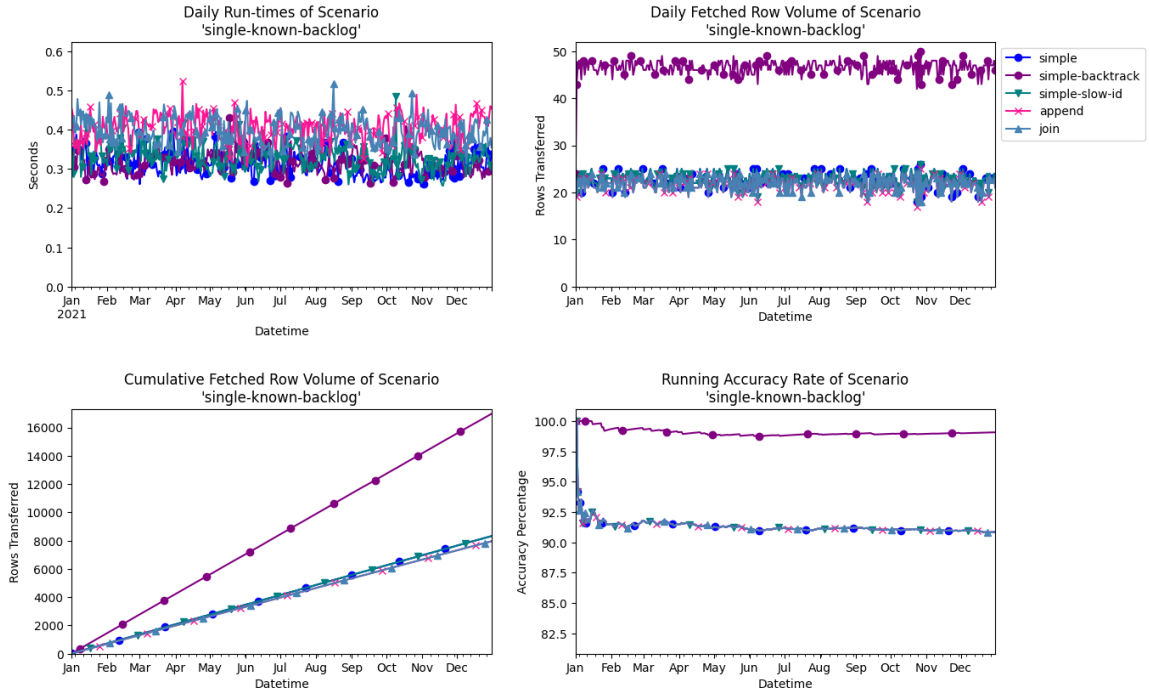

Figure 4.5: When backlogged rows are quickly inserted, *Simple Backtrack Sync* picks up considerable accuracy at the cost of bandwidth but not run-time.
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

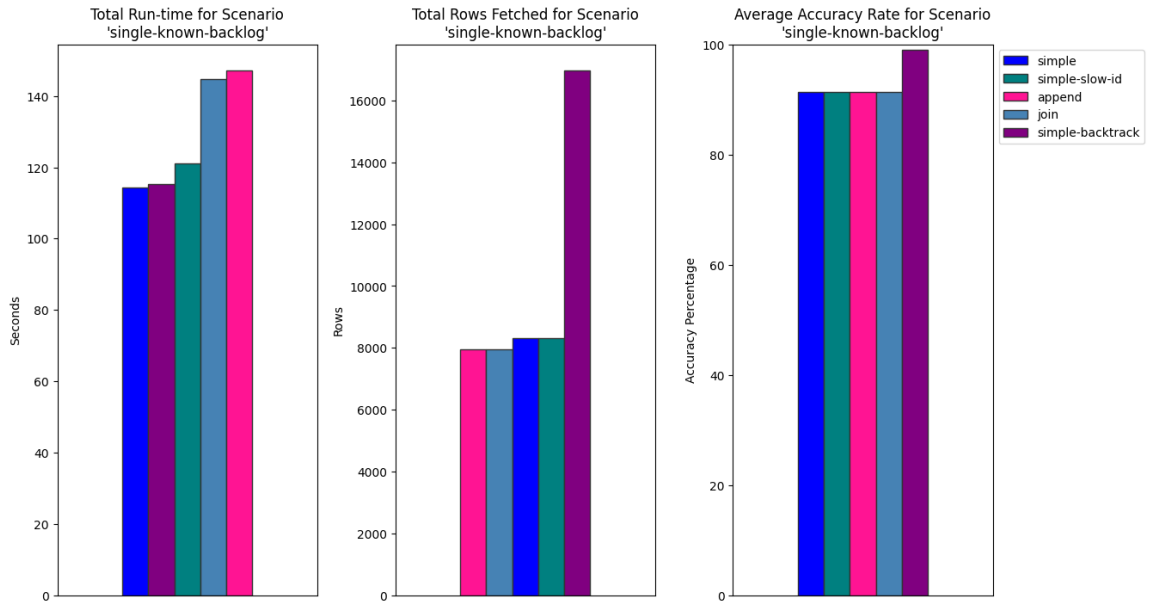

Figure 4.6: When bandwidth is cheap and rows are frequently backlogged within a known interval, *Simple Backtrack Sync* is an attractive option.
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

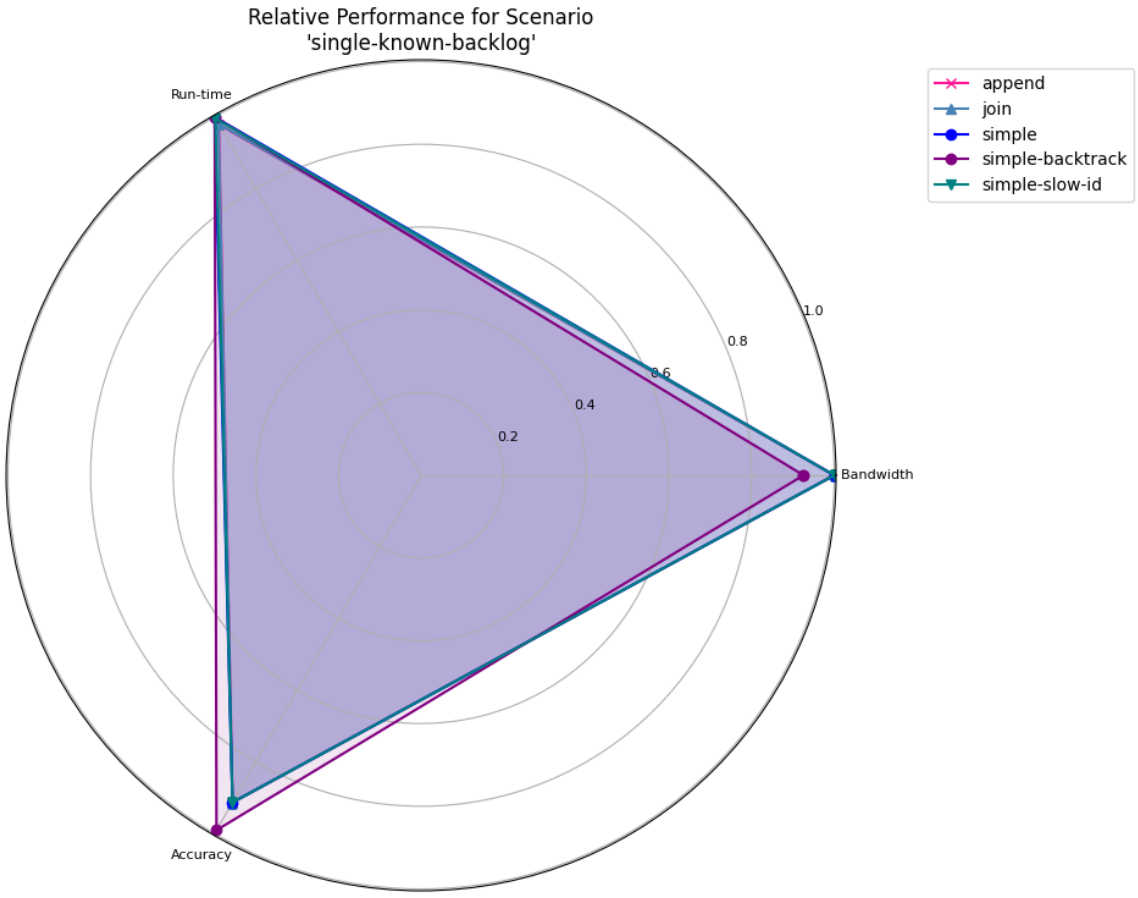

Figure 4.7: Apart from slight variations, *Simple Syncs* behave comparably when the number of IDs is small.
"Higher is better" for all metrics.

4.1.3 Comparing *Iterative Syncs*

The *Iterative Syncs* all share the design goal of maintaining perfect accuracy. As noted by Byers et al. [2002], exact synchronizations require significantly more effort than approximate solutions, so every strategy under-performs *Simple Sync*. However, the different designs of these strategies result in further choices which vary in performance.

Best in Bandwidth: *Iterative CPISync*

Consider the stark performance difference of *Iterative CPISync* (Figure 4.8 and Figure 4.9). Just like its signature appeal, *CPISync* achieves an exact synchronization with optimal communication complexity — that is, the additional bandwidth atop *Simple Sync* is used to fetch only the missing rows. This is possible because *CPISync* executes $filter()$ prior to fetching backlogged records. However, due to the high execution cost, *Iterative CPISync* should ideally only be employed when the source database can handle the increased load.

Best in Run-time: *Daily Row-Count Sync*

Like *Simple Backtrack Sync*, *Daily Row-Count Sync* trades bandwidth for accuracy without seriously compromising run-time. To perform an exact synchronization, more run-time is required than *Simple Backtrack Sync*, though the same principle is at play: *Daily Row-Count* casts a wide net — but not too wide to the point of degrading run-time. At the cost of 300% of the bandwidth required of *Simple Sync*, it only requires 280% of the run-time (260% when bounded) to achieve perfect accuracy, significantly less time than the 3,000% that comes with *CPISync*.

Balancing Run-Time and Bandwidth: *Iterative Simple Sync* and *Binary Search Sync*

Iterative Simple Sync and *Binary Search Sync* perform comparably with *Binary Search Sync* slightly preferring bandwidth over run-time (*Binary Search Sync* saves approximately 2% bandwidth over *Iterative Simple Sync* at a roughly 10% increase in run-time). Both strategies occupy the middle ground between *Iterative CPISync* and *Daily Row-Count Sync* and offer acceptable run-time and bandwidth while maintaining exact synchronization. For example, *Iterative Simple Sync* requires 240% of the bandwidth and 980% of the run-time of *Simple Sync* (compared to 110% and 3,000% for *Iterative CPISync* and 300% and 280% for *Daily Row-Count Sync*)

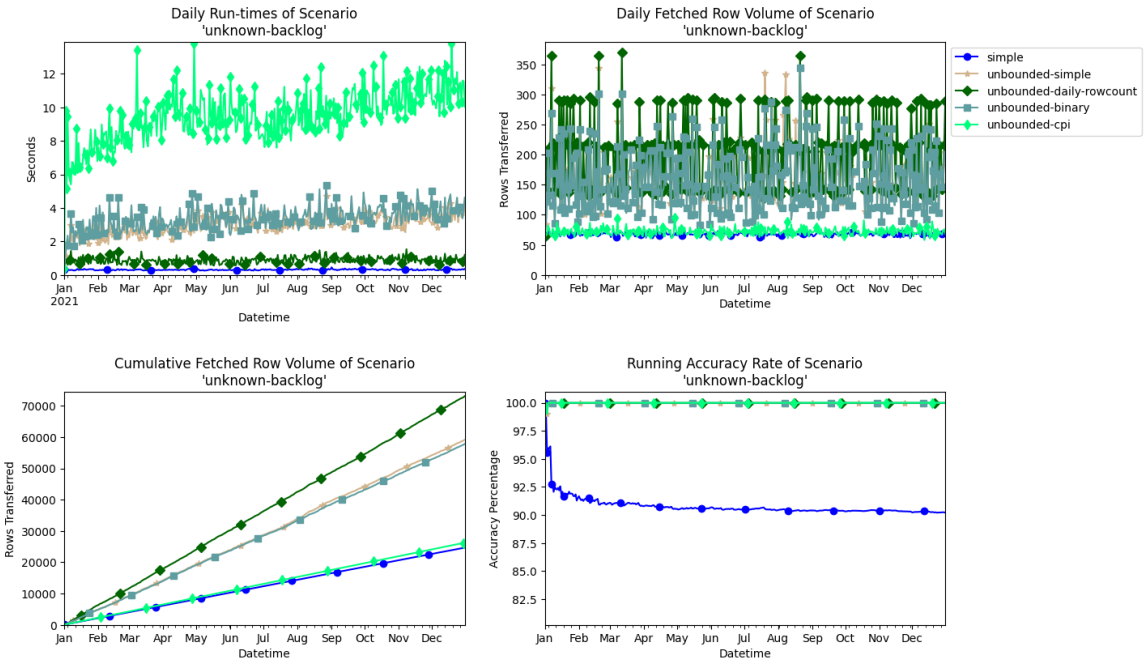

Figure 4.8: The run-times of *Iterative Syncs* scale with the underlying tables in the name of maintaining guaranteed accuracy.
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

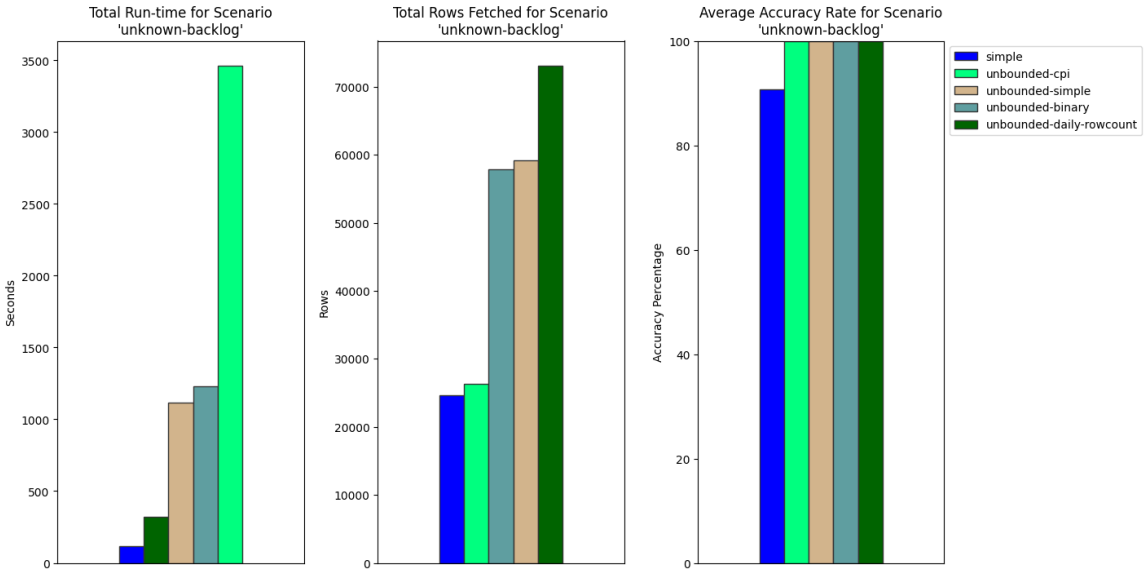

Figure 4.9: The trade-offs made by *Iterative Syncs* in order to maintain guaranteed perfect accuracy become clear when contrasting the summary metrics.
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

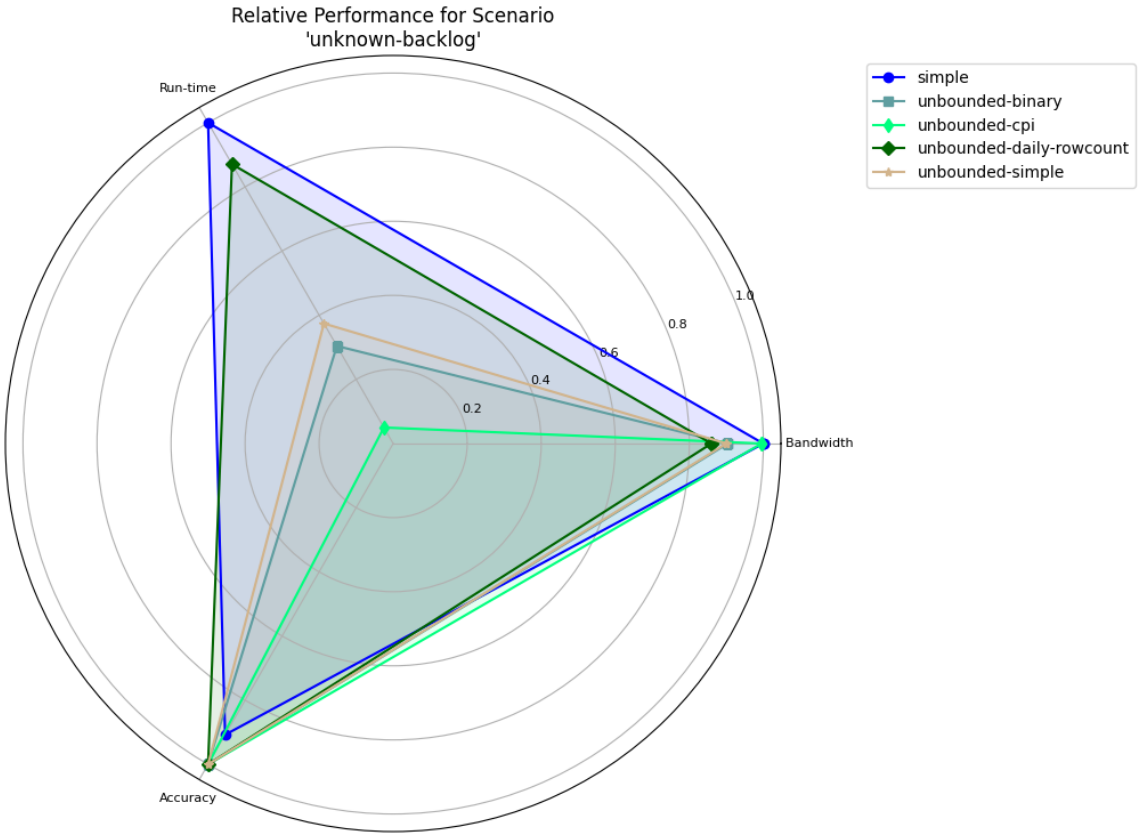

Figure 4.10: *Iterative Syncs* maximize accuracy at the expense of run-time and bandwidth.
"Higher is better" for all metrics.

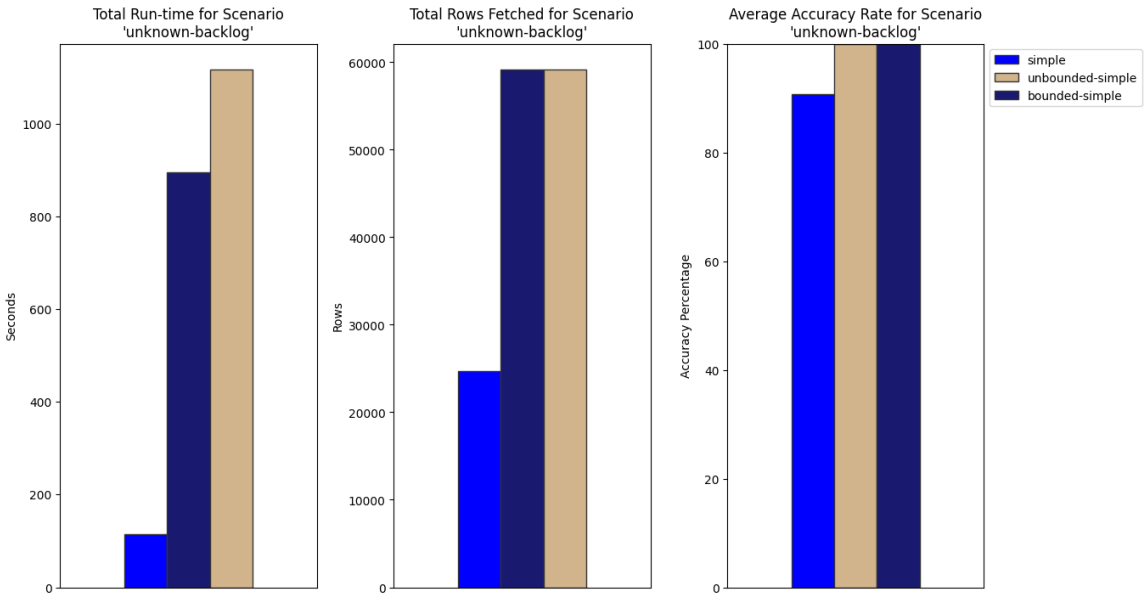

Figure 4.11: Bounded *Iterative Syncs* only vary from unbounded versions in that run-time remains mostly constant rather than scaling with the size of the tables. *"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.*

4.1.4 Comparing *Corrective Syncs*

Although not perfect synchronizations, the amortized accuracy rate of *Corrective Syncs* asymptotically approaches 100%. In practice, the average accuracy rate is slightly depressed because on most days, at least several rows are missing from the target table. The number of outstanding rows is restricted to the most recent month, so as the table grows, the accuracy rate rises.

Bandwidth-First: *Simple Monthly Iterative CPISync*

Executing iterative synchronizations once per month dramatically reduces the expensive processing requirement of *Iterative CPISync* while still achieving a high accuracy rate and near-optimal bandwidth performance. With an average accuracy rate of 98%, *Simple Monthly Iterative CPISync* reduces the total run-time from 3,000% of *Simple Sync* to just 690% (670% when bounded).

The Perfect Balance: *Simple Monthly Daily Row-Count Sync* and *Simple Monthly Iterative Simple Sync*

Simple Monthly Daily Row-Count Sync is a "jack of all trades; master of none" as far as immutable time-series synchronization strategies go. In the tested scenario, it improves the average accuracy rate from 91% to 98% and spreads the cost between run-time and bandwidth; its total run-time performance is 170% of *Simple Sync* and bandwidth is 200%, meaning that it fails to optimize for any particular metrics but achieves a sustainable balance between the three.

It's worth noting that the bounded variant of *Simple Monthly Iterative Simple Sync* actually outperforms bounded *Simple Monthly Daily Row-Count Sync* by 13% in run-time and 1% in bandwidth (see Figure 4.14), but the same does not hold true

for the pure iterative versions. Because *Iterative Simple Sync* checks the row-counts of each partition in separate transactions, the repetitive counting results in significantly wasted run-time. Bounding the iteration caps the number of transactions and reduces the wasted effort previously spent on counting. Therefore, *Simple Monthly Iterative Simple Sync* is preferable when choosing a bounded corrective strategy.

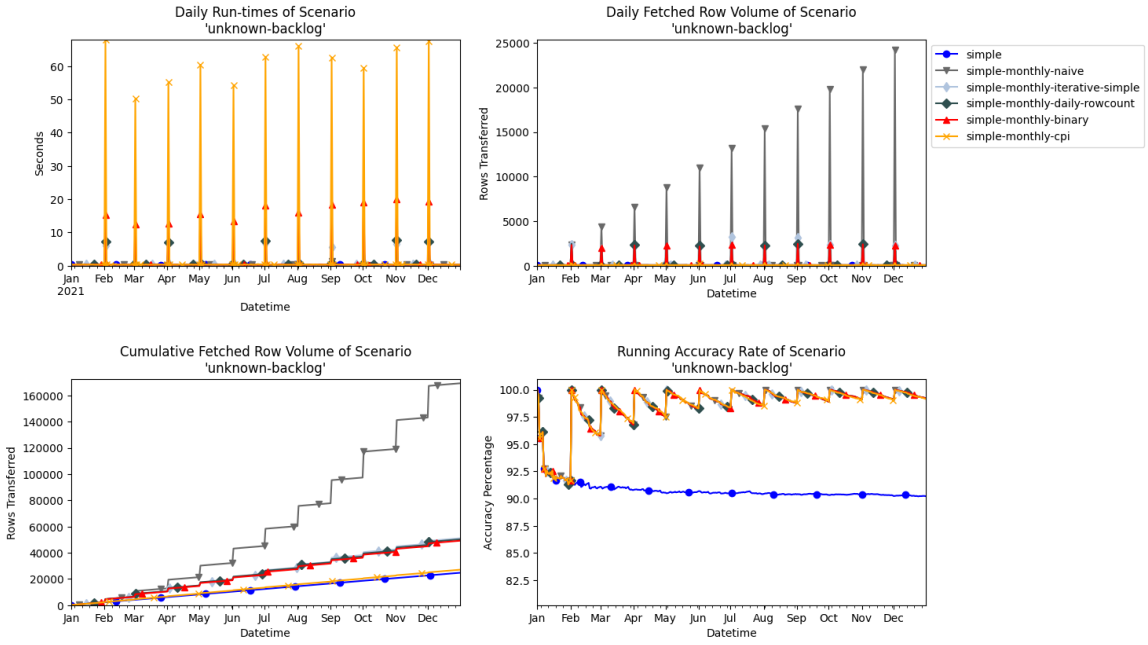

Figure 4.12: The *Corrective Syncs* are characterized by monthly, expensive verification syncs.
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

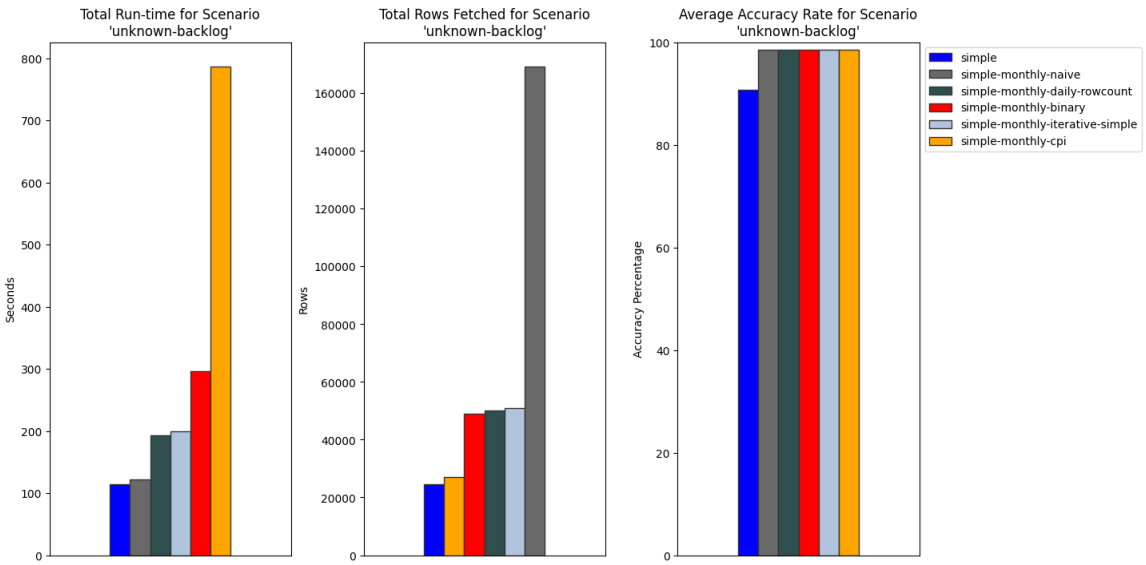

Figure 4.13: *Simple Monthly Daily Row-Count Sync* is the most balanced unbounded corrective strategy.
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

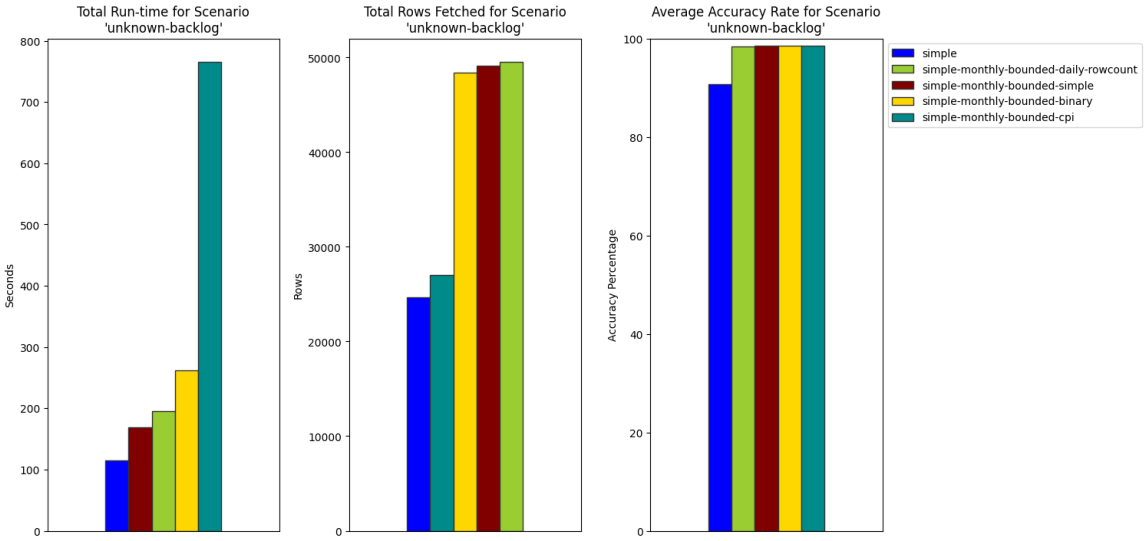

Figure 4.14: *Simple Monthly Iterative Simple Sync* outperforms *Simple Monthly Daily Row-Count Sync* when the iterations are bounded.
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

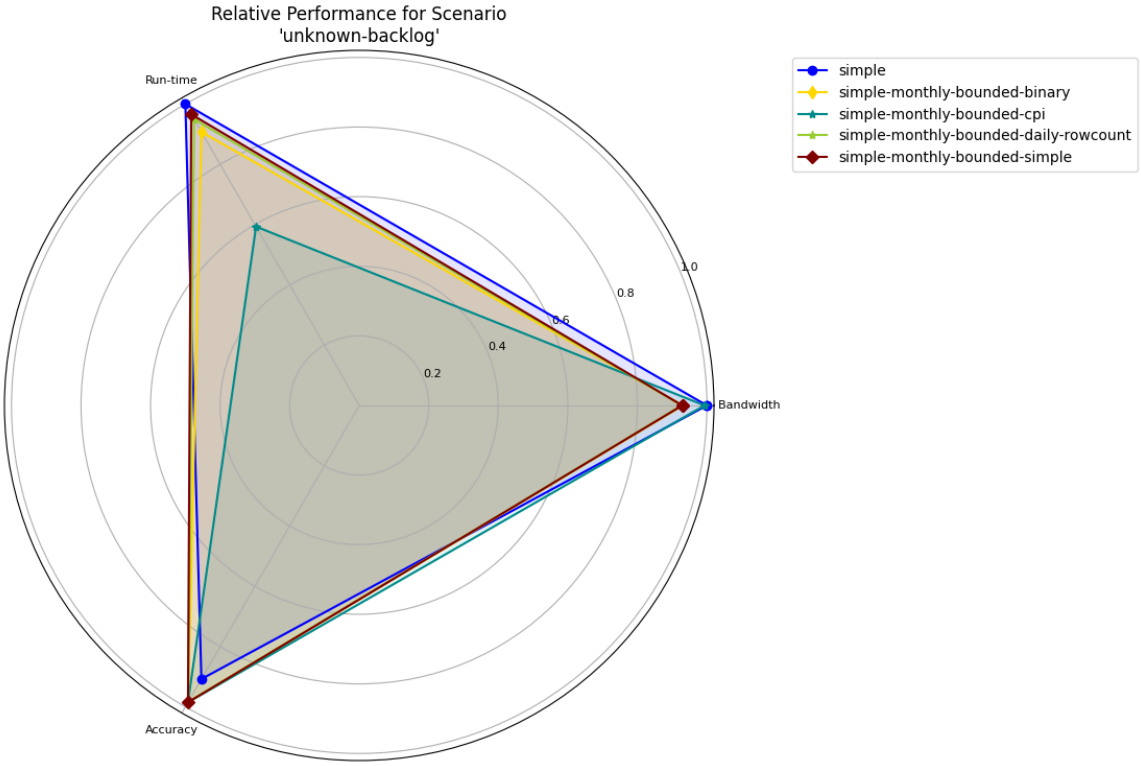

Figure 4.15: *Corrective Syncs* provide flexibility for improving run-time or bandwidth without significantly sacrificing accuracy.
"Higher is better" for all metrics.

4.2 Ranking Strategies

After examining the performances of strategies within their classes, it follows that a mechanism for directly ranking strategies is worth consideration. Although the summary bar chart Figure 4.16 below offers a sense of the advantages of each choice, it lacks an intuitive way to directly rank them according to the readers' priorities. For example, if the reader is most concerned with a balance of run-time and accuracy, how can we provide a list of suggested strategies?

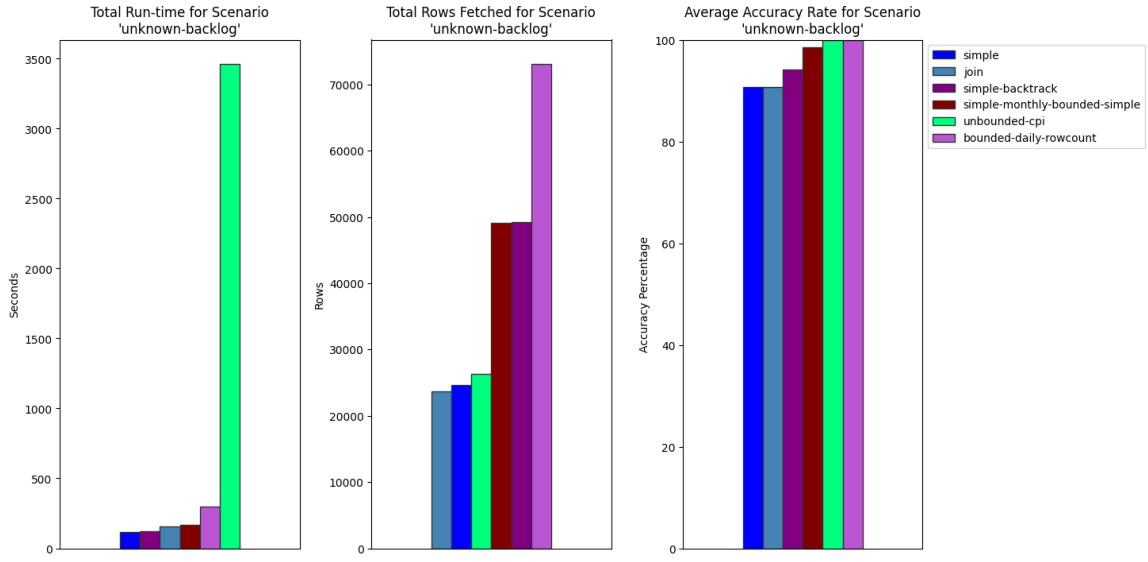

Figure 4.16: The summary bar chart visualizes the advantages of strategies but does not allow for direct rankings beyond single metrics.
"Lower is better" for run-time and bandwidth, and "higher is better" for accuracy.

The Choice Index

One solution is to calculate a weighted average of the normalized performance scores presented as "skills" in the summary radar charts. This "score" (referred to as the *Choice Index*) allows us to craft a ranked list of suggestions according to the user's priorities.

4.2.1 One Metric

Like the summary bar charts, Figure 4.17 below ranks the preferred strategies by applying a 100% weight to a single metric. This weight distribution clearly makes direct comparisons for specific metrics.

Analysis

In this case, *Simple Sync* is the fastest, *Simple Join Sync* is the most bandwidth-efficient, and the iterative strategies *Daily Row-Count Sync* and *Iterative CPISync* maintain perfect accuracy. The trade-offs made to achieve "best in class" are visible, but to recognize the value of "middle-of-the-road" strategies like *Simple Monthly Iterative Simple*, multiple metrics need to be considered.

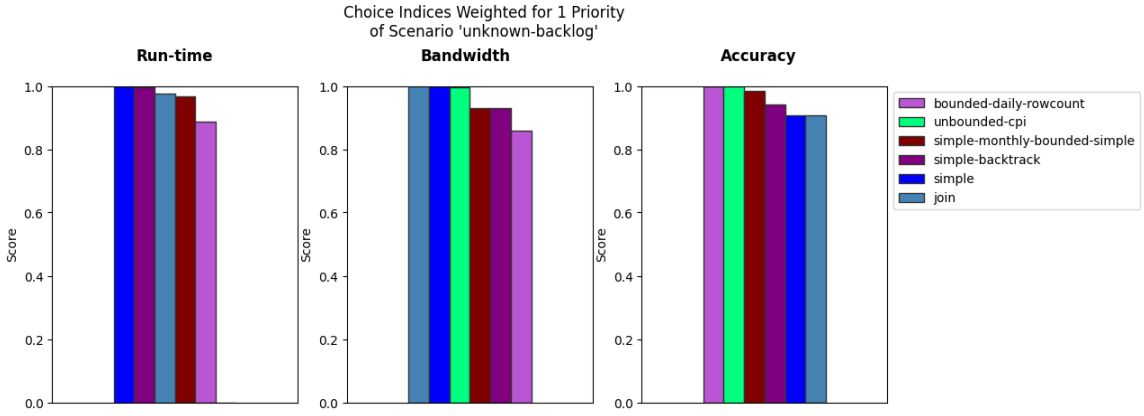

Figure 4.17: Preferred strategies ranked by a single metric.
"Higher is better" for all metrics.

4.2.2 Two Metrics

Extending the *Choice Index* to two priorities involves distributing the weights for each combination of metrics. The resulting rankings are arranged into a 3x3 grid where the column corresponds to the first priority (weighted at 66.67%) and the

row to the second metric (weighted at 33.33%). This layout includes the rankings of Figure 4.17 in its diagonal and allows for additional discretion when choosing preferred metrics.

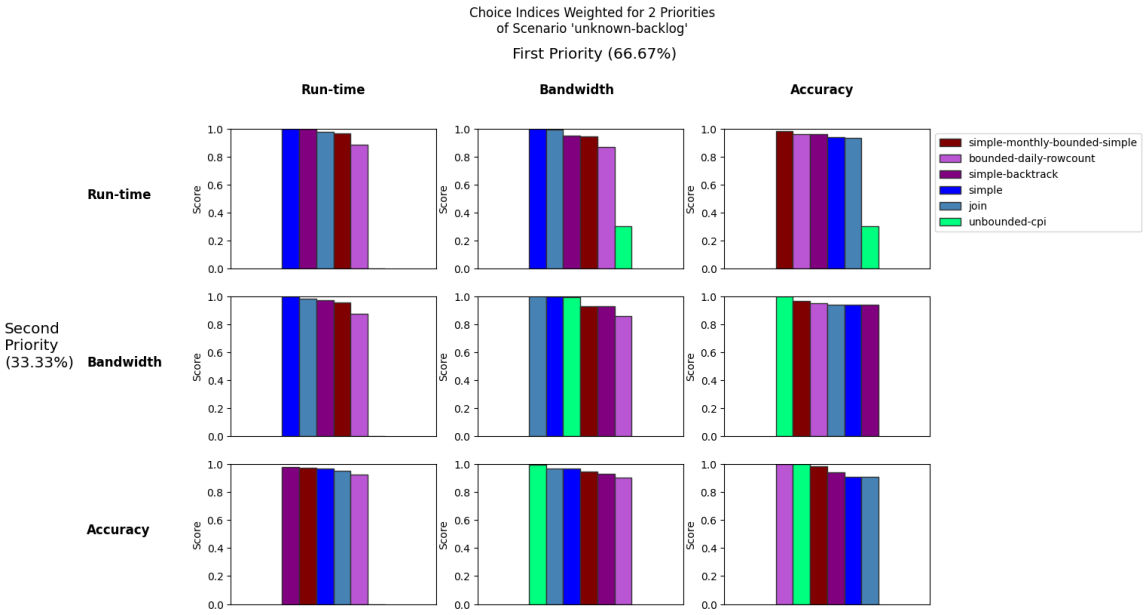

Figure 4.18: Preferred strategies ranked by two metrics.
"Higher is better" for all metrics.

Analysis

Simple Sync typically prevails when run-time or bandwidth is weighted highest, but the added dimension emphasizes *Simple Monthly Iterative Simple Sync* with accuracy as a second priority. Similarly, when ranking bandwidth and accuracy, *Iterative CPISync* is able to rise to the top because its detrimental run-time is not considered. Ranking for accuracy results in the most diversity in choice because it allows the unique qualities of the strategies to shine through.

4.2.3 Three Metrics

The rankings in Figure 4.19 consider all three metrics; the first metric is weighted at 57.14%, the second at 28.57%, and the third at 14.29%[2]. Comparing strategies in this way gives us the ability to determine more balanced strategies while allowing for emphasis on our preferred metrics.

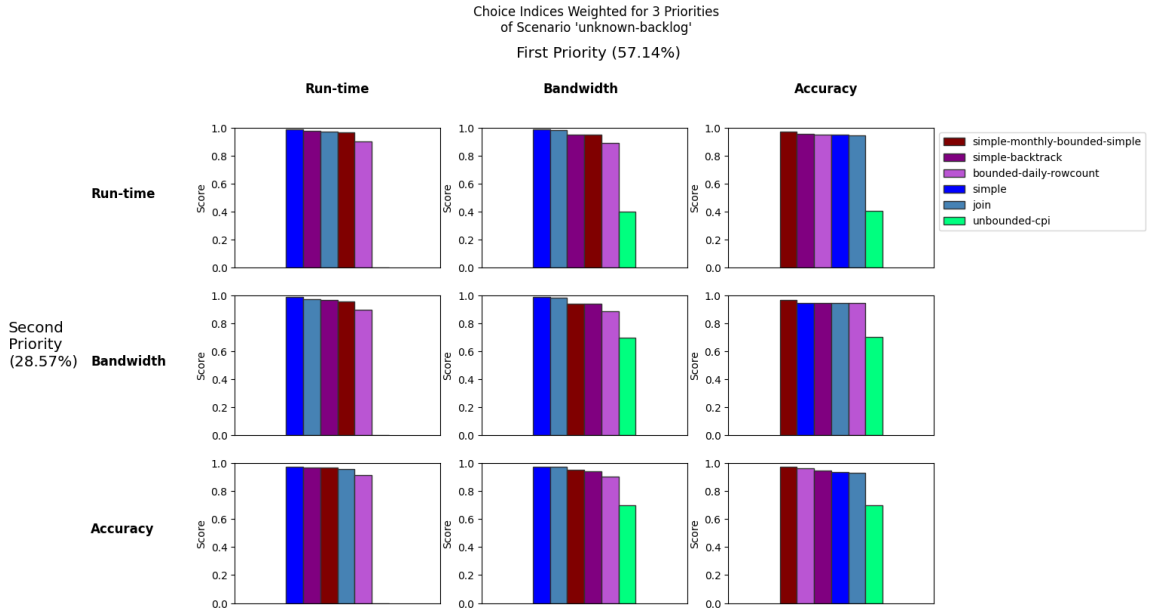

Figure 4.19: Preferred strategies ranked by three metrics.
"Higher is better" for all metrics.

Analysis

When preferentially weighing run-time and bandwidth, *Simple Sync* always comes out on top. Despite exhibiting the worst accuracy of all the methods, its lightweight design more than makes up for what it lacks in accuracy. It may be tempting to always choose *Simple Sync*, but keep in mind its sensitivity to external conditions — its accuracy rate is entirely dependent on the prevalence of outages. For

[2]The weights were chosen as multiples of $\frac{1}{7}$ such that each subsequent priority has half the weight of the last.

simple analytical purposes where the stream of data always increases and aggregations are frequently performed (e.g. averaging weather data), the "good enough" accuracy of *Simple Sync* is worth the efficient synchronization.

When perfect accuracy is an absolute requirement, then the pool of choices is reduced to the *Iterative Syncs* (see subsection 4.1.3). Otherwise, a strategy from the *Corrective Syncs* is likely the best option. One of the most well-rounded strategies is the bounded variant of *Simple Monthly Iterative Simple Sync*, which tremendously increases the accuracy of *Simple Sync* while maintaining acceptable run-time and bandwidth.

Chapter 5

Conclusion

5.1 Choosing a Strategy

The sections below summarize the findings from chapter 4 to aid the reader in choosing a strategy. Keep in mind not every situation will neatly fit into the described scenarios and that the following suggestions are to an extent subjective; please refer to chapter 4 for more comprehensive comparisons.

5.1.1 Identify the Scenario

The first step to choosing a synchronization strategy is identifying which features of the following scenarios best describe your situation.

1. *single-append-only* — Backlogging in the source table is not a concern, and a single ID makes up the data stream.

2. *multiple-large-n-append-only* — Backlogging is not a concern, and many IDs make up the data stream.

3. *single-known-backlog* — Records are backlogged into the source table within a known interval.

4. *unknown-backlog* — Records are backlogged into the source table at random. An unknown number of IDs (3) makes up the data stream.

Append-Only

As mentioned in subsection 2.1.1, the ideal strategy for a single-ID, append-only data stream is *Simple Sync*. The other strategies intend to mitigate backlogging or reduce bandwidth, but *single-append-only* already is the ideal scenario which requires no additional measures to accommodate backlogging.

The scenario *multiple-large-n-append-only* adds another dimension which may be used to alter the behavior of *Simple Sync*. When the number of IDs are large and the frequency of records is high, the ID-focused strategies offer the best bandwidth savings at a steep increase in run-time. Therefore, when optimizing for bandwidth, the strategy *Simple Join Sync* is the best choice if run-time is not a concern, otherwise *Simple Sync* remains an appropriate choice for append-only situations.

Known Backlog

For situations with regular "outages" which are resolved quickly (e.g. within 24 hours), the strategy *Simple Backtrack Sync* offers an excellent balance between run-time, bandwidth, and accuracy. Although the number of rows fetched may be a multiple of *Simple Sync*, when compared to *Naïve Sync*, its bandwidth performance is still acceptable.

Unknown Backlog

The scenario *unknown-backlog* offers the most choices. To choose a strategy which suits your needs, you first need to prioritize the metrics. Similar to the *Choice Index* rankings from section 4.2, we offer the strategy recommendation chart below to guide the reader when choosing a strategy. Select the column of the highest priority metric, then choose the row corresponding to the second priority.

		First Priority		
		Run-time	**Bandwidth**	**Accuracy**
Second Priority	**Run-time**	*Simple*	*Simple*	*Daily Row-Count*
	Bandwidth	*Simple*	*Simple Join*	*Iterative CPISync*
	Accuracy	*Simple Monthly Iterative Simple (bounded)*	*Iterative CPISync*	*Daily Row-Count*

Table 5.1: Strategy recommendation chart for situations similar to *unknown-backlog*

Practical Example:
Clemson Energy Visualization and Analytics Center

A real-world example of choosing an appropriate strategy is the initial design process of the Clemson Energy Visualization and Analytics Center (CEVAC). CEVAC was faced with the situation described in subsection 1.1.1 where utilities data tables resided on sensitive production databases. The first solution resembled *Simple Sync*, but due to regular backlogging in the source databases, a version of *Simple Backtrack Sync* was devised. CEVAC was tasked with frequently extracting many sub-streams (often with many IDs) from the source databases over a high-bandwidth link, so run-time was the chief priority. Because the situation mostly consisted of known backlogging intervals, *Simple Backtrack Sync* (with variable BTIs) best fit the situation and desired priorities.

5.2 Future Work

The findings in this thesis demonstrate the surprising variability amongst target-based approaches to the immutable time-series synchronization problem. Given the tremendous real-world applications of synchronization strategies and the growing importance of the Internet of Things, future work may reveal cost-saving solutions. Below are points of interest which may warrant further research.

5.2.1 Additional Strategies

Fetch Strategies

The $fetch()$ strategies outlined in this thesis fell into the classes *Simple Syncs*, *Iterative Syncs*, and *Corrective Syncs*. This framework is useful for organization but does not at all represent the sphere of possible designs. For example, a class may be constructed on whether strategies rely on prior context. Other strategies may be written depending on the network layout; this thesis assumes a source database, target database, and intermediate syncing service, but other combinations such as cluster configurations may result in intricate strategies as opposed to the straightforward methods outlined in chapter 3.

Filter Strategies

The stages $fetch()$, $filter()$, and $insert()$ were introduced in subsection 1.3.1, but chapter 3 only discusses fetch strategies. Filtering mechanisms implemented in SQL and `pandas` are mentioned in subsection 1.3.3, but other possibilities may enhance performance. After all, $filter()$ is a smaller, localized version of the larger set reconciliation problem. In this context, algorithms introduced in section 1.2 like Bloom Filters may be applied.

Insert Strategies

The opportunity for optimization of the *insert*() stage in production is mentioned in subsection 1.3.4, such as the Meerschaum implementation which takes advantage of PostgreSQL's bulk insertion features. Strategies at this stage of the synchronization process may also experiment with chunking: does a dynamic chunk size outperform a static size, and if so, how should the chunk size be determined? Factors which were overlooked — such as the network link, filtered sample size, and insertion protocol — may lead to opportunities to further optimize the overall synchronization procedure.

Further Optimizations

Opportunities for optimization exist outside the *synchronize*() stages. For example, multi-stage pipes which derive from existing pipes or pipes which join multiple data sources consist of multiple instances of *synchronize*(). The manners in which these complicated data sets are constructed influence the performance of the overall synchronization.

5.2.2 Time-Series Properties

Mutability

One key property of this thesis is the assumption that rows remain immutable. This is often the case for the fundamental data streams, but any amount of aggregation requires immutability. Solutions like EventDB get around this by storing changes as immutable "events" so that the tables may be quickly "rolled back" to earlier states [Zhao et al., 2019]. This design has its advantages but storing each change as a distinct "event" requires more space than may be necessary. Therefore,

if sufficient interest exists, then follow-up studies may investigate modified strategies which account for varying degrees of mutability.

Frequency, Timescale, and Resolution

The strategies in this thesis primarily rely on the datetime axis alone in order to retain flexibility for all frequencies and timescales. However, strategies may be designed with specific frequencies, timescales, and resolutions in mind. For example, the run-time performance of *Iterative CPISync* may be improved when the frequency and resolution are known (subsection 3.2.4).

5.2.3 Experiment Design

Evaluation Metrics

The results in chapter 4 are presented to most clearly illustrate the effects of the strategies on straightforward metrics. The three metrics — run-time, bandwidth, and accuracy — by no means encompass every metric by which strategies may be compared but rather serve to avoid complicated analyses by only showing what a user would most likely care to see (e.g. the weighted *Choice Index* rankings would be difficult to read with more than three metrics). A future study may include additional metrics when determining the *Choice Index*.

Scenario Simulations

The simulated scenarios are intended to simplify the data streams to their characteristics found in real-world data streams rather than mirror the many combinations found in real-world scenarios. This design choice highlights strengths and weaknesses of the algorithms but may obscure qualities of real-world situations. Therefore,

follow-up papers may conduct field studies and identify unforeseen shortcomings of the strategies.

5.3 Summary

In the context of immutable time-series data streams, we demonstrate several novel synchronization strategies. In addition to performance analyses, we recommend algorithms depending on scenarios' characteristics and the reader's priorities. In general, the following strategies are declared as the "winners":

1. *Simple Sync* for minimal run-time and bandwidth (with decent accuracy).

2. *Daily Row-Count Sync* for minimizing run-time and bandwidth while maintaining perfect accuracy.

3. The bounded variant of *Simple Monthly Iterative Simple Sync* for balancing all three metrics.

Bibliography

Madhu Ahluwalia, Ruchika Gupta, Aryya Gangopadhyay, Yelena Yesha, and Michael McAllister. Target-Based Database Synchronization. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, page 1643–1647, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605586397. doi: 10.1145/1774088.1774443. URL `https://doi.org/10.1145/1774088.1774443`.

Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't Thrash: How to Cache Your Hash on Flash, 2012.

Burton H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692. URL `https://doi.org/10.1145/362686.362692`.

Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An Improved Construction for Counting Bloom Filters. In *14th Annual European Symposium on Algorithms, LNCS 4168*, pages 684–695, 2006.

Pranav Byali, Md Zaid S Bevinahalli, and Vaishnavi Chavan. Bloom Filter: A Data Structure for Quick Searching. *International Journal of Engineering Research & Technology*, 8(15):204–206, 2020.

John Byers, Jeffrey Considine, and Michael Mitzenmacher. Fast Approximate Reconciliation of Set Differences. In *BU Computer Science TR*, pages 2002–19, 2002.

Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*, volume 15, pages 30–39, 01 2004. doi: 10.1145/982792.982797.

Mi Young Choi, Eun Ae Cho, Dae Ha Park, Chang Joo Moon, and Doo Kwon Baik. A database synchronization algorithm for mobile devices. *IEEE Transactions on Consumer Electronics*, 56(2):392–398, May 2010. ISSN 0098-3063. doi: 10.1109/TCE.2010.5505945.

David Eppstein and Michael T Goodrich. Straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, 23(2):297–306, 2010.

David Eppstein, Michael T. Goodrich, Frank Uyeda, and George Varghese. What's the Difference? Efficient Set Reconciliation without Prior Context. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 218–229, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307970. doi: 10.1145/2018436.2018462. URL `https://doi.org/10.1145/2018436.2018462`.

Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332798. doi: 10.1145/2674005.2674994. URL `https://doi.org/10.1145/2674005.2674994`.

Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3): 281–293, 2000. doi: 10.1109/90.851975.

Michael T. Goodrich and Michael Mitzenmacher. Invertible Bloom Lookup Tables, 2011.

Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *ACM J. Exp. Algorithmics*, 25, March 2020. ISSN 1084-6654. doi: 10.1145/3376122. URL `https://doi.org/10.1145/3376122`.

Thomas Greiner and Anton Donner. Data Management in Mass Casualty Incidents: The e-Triage Project. In *Workshop zur IT-Unterstützung von Rettungskräften im Rahmen der GI-Jahrestagung*, pages 192–198, Sep 2010. URL `https://subs.emis.de/LNI/Proceedings/Proceedings176/192.pdf`.

Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Time Series Management Systems: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2581–2600, 2017. doi: 10.1109/TKDE.2017.2740932.

Jiaxi Jin, Wei Si, David Starobinski, and Ari Trachtenberg. Prioritized Data Synchronization for Disruption Tolerant Networks. In *IEEE Military Communications Conference MILCOM*, pages 1–8, 10 2012. ISBN 978-1-4673-1729-0. doi: 10.1109/MILCOM.2012.6415678.

Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

Sanja Lazarova-Molnar and Nader Mohamed. Collaborative Data Analytics for Smart Buildings: Opportunities and Models. *Cluster Computing*, 22(1):1065–1077, Jan 2019. ISSN 1573-7543. doi: 10.1007/s10586-017-1362-x. URL https://doi.org/10.1007/s10586-017-1362-x.

Bennett Meares. Meerschaum. https://github.com/bmeares/Meerschaum, 2021a.

Bennett Meares. syncx. https://github.com/bmeares/syncx, 2021b.

Ralph C Merkle. Protocols for Public Key Cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122. IEEE, 1980.

Yaron Minsky and Ari Trachtenberg. Set Reconciliation with Nearly Optimal Communication Complexity. In *In International Symposium on Information Theory*, pages 2213 – 2218. IEEE, 2001.

Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive Cuckoo Filters. *ACM J. Exp. Algorithmics*, 25, March 2020. ISSN 1084-6654. doi: 10.1145/3339504. URL https://doi.org/10.1145/3339504.

Muhammad Muhammad, Stefan Erl, and Matteo Berioli. Efficient Synchronization of Multiple Databases over Broadcast Networks. In Riadh Dhaou, André-Luc Beylot, Marie-José Montpetit, Daniel Lucani, and Lorenzo Mucchi, editors, *Personal Satellite Services*, pages 77–89, Cham, 2013. Springer International Publishing. ISBN 978-3-319-02762-3. doi: 10.1007/978-3-319-02762-3_7.

Onica. Accelerating Fraud Detection and Enabling Real-Time Dynamic Data Querying through an Efficient Data Pipeline, 2020. URL https://onica.com/case-study/m1-finance/.

Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, page 823–829, USA, 2005. Society for Industrial and Applied Mathematics. ISBN 0898715857.

Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow.*, 8(12):1816–1827, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824078. URL https://doi.org/10.14778/2824032.2824078.

Sinh Pham. Key-Value Storage System Synchronization in Peer-to-Peer Environments. Master's thesis, University of Saskatchewan, 2014.

Felix Putze, Peter Sanders, and Johannes Singler. Cache-, Hash-, and Space-Efficient Bloom Filters. *ACM J. Exp. Algorithmics*, 14, January 2010. ISSN 1084-6654. doi: 10.1145/1498698.1594230. URL https://doi.org/10.1145/1498698.1594230.

Mark Raasveldt and H. Mühleisen. Data Management for Data Science - Towards Embedded Analytics. In *CIDR*, 2020.

Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. LittleTable: A Time-Series Database and Its Uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 125–138, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3056102. URL https://doi.org/10.1145/3035918.3056102.

Chen Tang, Anton Donner, Javier Mulero Chaves, and Muhammad Muhammad. Performance of Database Synchronization Algorithms via Satellite. In *2010 5th Advanced Satellite Multimedia Systems Conference and the 11th Signal Processing for Space Communications Workshop*, pages 455–461, 2010. doi: 10.1109/ASMS-SPSC.2010.5586921.

Daniel Ting and Rick Cole. *Conditional Cuckoo Filters*, page 1838–1850. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383431. URL https://doi.org/10.1145/3448016.3452811.

Ari Trachtenberg, David Starobinski, and Sachin Agarwal. Fast PDA Synchronization Using Characteristic Polynomial Interpolation. In *IEEE INFOCOM*, pages 1–10, 2002.

Andrew Tridgell and Paul Mackerras. The rsync Algorithm. Technical report, Australian National University, 1996. URL https://www.cs.cmu.edu/~15-749/READINGS/required/cas/tridgell96.pdf.

Shyam Vyas. Joint Polar Satellite System (JPSS) Ground System Concept of Operations. Technical report, National Oceanic and Atmospheric Administration and National Aeronautics and Space Administration, 2019. URL https://www.jpss.noaa.gov/assets/pdfs/474-00054_JPSS-GS-ConOps_E%20(5).pdf.

Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A. McGrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. Apache IoTDB: Time-Series Database for Internet of Things. *Proc. VLDB Endow.*, 13(12):2901–2904, August 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415504. URL https://doi.org/10.14778/3415478.3415504.

Yang Yang, Qiang Cao, and Hong Jiang. EdgeDB: An Efficient Time-Series Database for Edge Computing. *IEEE Access*, 7:142295–142307, 2019. doi: 10.1109/ACCESS.2019.2943876.

Wenjia Zhao, Yong Qi, Di Hou, Peijian Wang, Xin Gao, Zirong Du, Yudong Zhang, and Yongfang Zong. EventDB: A Large-Scale Semi-structured Scientific Data Management System. In Jianhui Li, Xiaofeng Meng, Ying Zhang, Wenjuan Cui, and Zhihui Du, editors, *Big Scientific Data Management*, pages 105–115, Cham, 2019. Springer International Publishing. ISBN 978-3-030-28061-1.