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

A Thesis Defense Presentation by Bennett Meares
8 October 2021

1. **Practical**
    a. Intended for real-world use
    b. Situation-dependent

2. **Target-Based**
    a. Limited context of the source database
    b. Typical of client-server model when using third party APIs

3. **Synchronization Strategies**
    a. Database reconciliation algorithms
    b. Used for identifying rows in a source table which are missing from a target table

4. **Immutable Time-Series Data**
    a. Timestamped rows which do not change once generated
    b. Typical in IoT and industrial applications

Outline and Problem Statement

1. **Introduction**
   - 1.1. Synchronization in Practice
   - 1.2. Related Works
   - 1.3. Overview of the Algorithm
2. **Scenarios**
   - 2.1. Append-Only Data Streams
   - 2.2. Backlogged Data
3. **Strategies**
   - 3.1. Speed-First: *Simple Syncs*
   - 3.2. No-Compromises Accuracy: *Iterative Syncs*
   - 3.3. Best of Both Worlds: *Corrective Syncs*
4. **Experimental Results**
   - 4.1. Comparing Classes of Strategies
   - 4.2. Ranking Strategies
5. **Conclusion**
   - 5.1. Choosing a Strategy
   - 5.2. Future Work
   - 5.3. Summary

Given several scenarios, which target-based immutable time-series synchronization strategies best optimize run-time, bandwidth, and accuracy?

Chapter 1
Introduction

Understanding the synchronization context

1.1 Synchronization in Practice

1. **Smart Buildings**
    a. Clemson University Facilities and CEVAC
        i. Rapidly fetch, aggregate, and display building data from many sources
        ii. ~300 GB MSSQL table which grows by tens of thousands of rows per day

2. **Environmental Observations**
    a. NOAA and NASA
        i. Joint Polar Satellite System (JPSS)
        ii. Comprehensive Large Array-Data Stewardship System (CLASS)
    b. NWS
        i. Public weather API
        ii. `noaa` Meerschaum plugin

3. **Financial Transactions**
    a. M1 Finance
        i. Fraud-detection system
        ii. Streams data from PostgreSQL via AWS DMS into warehouse
        iii. ~400 GB S3 bucket of Parquet files
        iv. Process data with Amazon Redshift Spectrum
    b. Apex Clearing Corporation
        i. Clearing house for M1 Finance
        ii. Transactions history provided via a simple datetime-bounded API
        iii. `apex` Meerschaum plugin

1.2 Related Works

1. **Hashing Partitions**
   a. [2010] Target-Based Database Synchronization
   b. [2010] Synchronization Algorithms based on Message Digest

2. **Characteristic Polynomial Interpolation**
   a. [2002] CPISync
   b. [2010] Partitioned-CPISync
   c. [2012] Priority CPISync
   d. [2013] Efficient Synchronization over Broadcast Networks

3. **Space-Efficient Approximate Synchronization**
   a. [1970] Bloom Filters
      i. [2002] Approximation Reconciliation Trees
      ii. [2010] Invertible Bloom Filters
      iii. [2011] Invertible Bloom Lookup Tables
      iv. [2011] Difference Digest
      v. [2014] Key-Value Storage System Synchronization in Peer-to-Peer Environments
   b. [2014] Cuckoo Filters
      i. [2020] Adaptive Cuckoo Filters
      ii. [2021] Conditional Cuckoo Filters
   c. [2020] XOR Filters

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.*

1.3 Overview of the Algorithm

Stage	Description	SQL Example

Fetch	Retrieve samples from the source and target databases.	`SELECT *`
		`FROM source_table`
		`WHERE datetime > '2021-01-01 00:00:00'::TIMESTAMP`

Filter	Remove rows found in the target sample from the source sample.	`SELECT ss.*`
		`FROM source_sample AS ss`
		`LEFT JOIN target_sample AS ts ON (`
		`  ss.id = ts.id`
		`  AND ss.datetime = ts.datetime`
		`)`
		`WHERE ts.datetime IS NULL`

Insert	Add the filtered sample to the target table.	`COPY target_table (datetime, id, value)`
		`FROM STDIN`
		`WITH CSV`

Simple Backtrack Sync

1. Determine the "reference time" (*RT*).
   a. Use the newest target datetime value by default (e.g. `'2021-01-01 00:00:00'::TIMESTAMP`)

2. Determine the "backtrack interval" (*BTI*).
   a. By default, use a value of 1 minute.

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

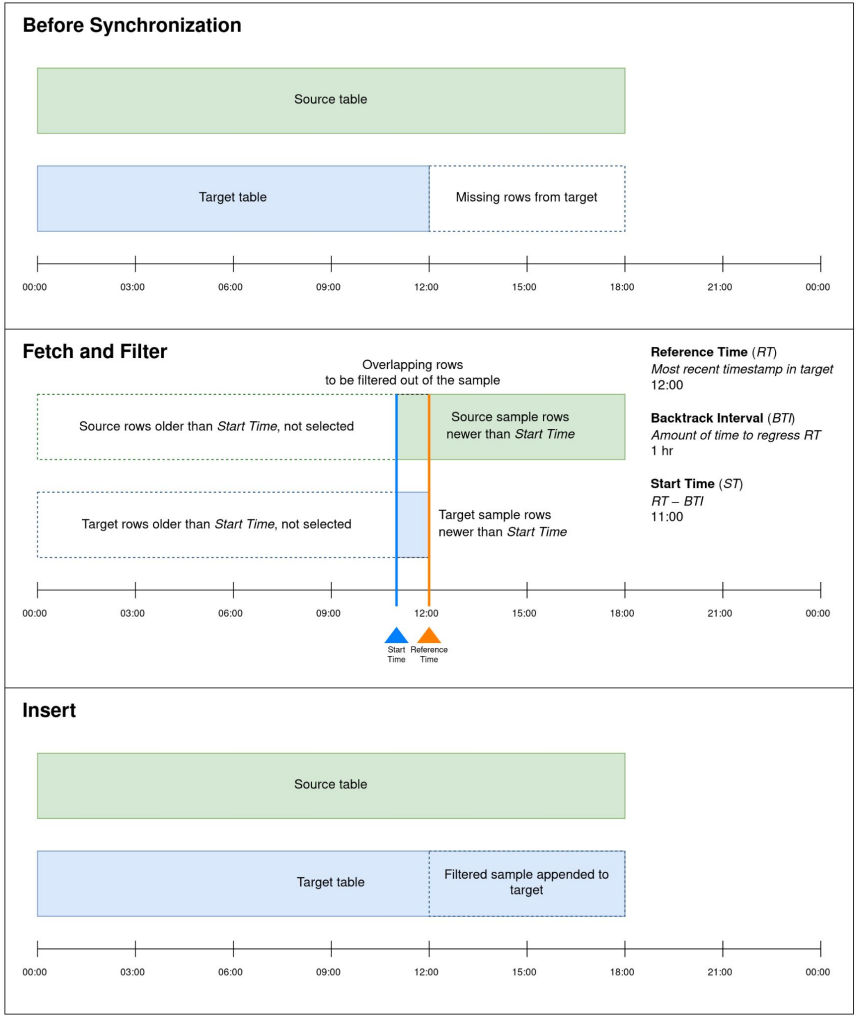
```
'2021-01-01 00:00:00'::TIMESTAMP - INTERVAL '1 minute'
```

4. Fetch source and target samples with rows greater than *ST*.

```
SELECT *
FROM table
WHERE datetime > ST
```

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

6. Insert the filtered sample in the target table.



Chapter 2
Scenarios

A look at types of immutable time-series data streams

2.1 Append-Only Data Streams

2.1.1 A Single Append-Only ID

Attributes

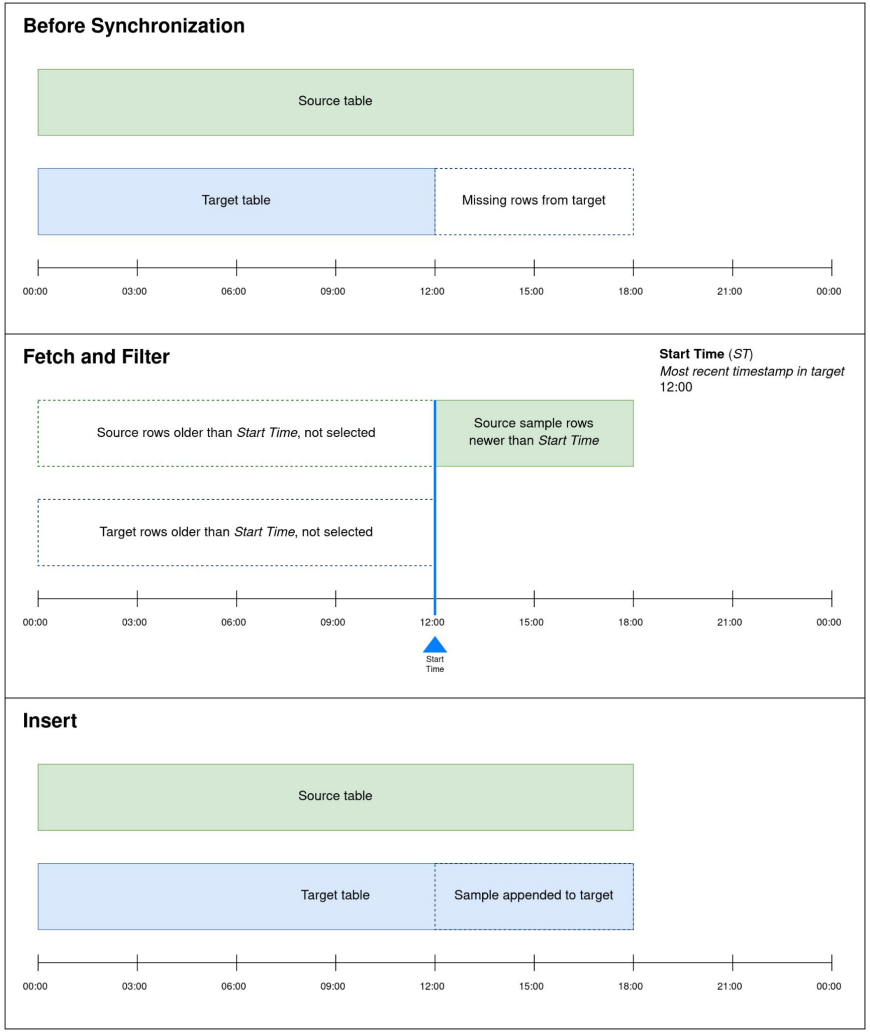
1. The data stream has a datetime index.
2. Rows are immutable.
3. New rows always have later datetime values than existing rows.

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

Simple Sync

1. Determine the most recent datetime from the target table as the "start time" (*ST*).
2. Fetch data from the source table newer than *ST*.
3. Insert the fetched data into the target table.

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



2.1.2 Multiple Append-Only IDs

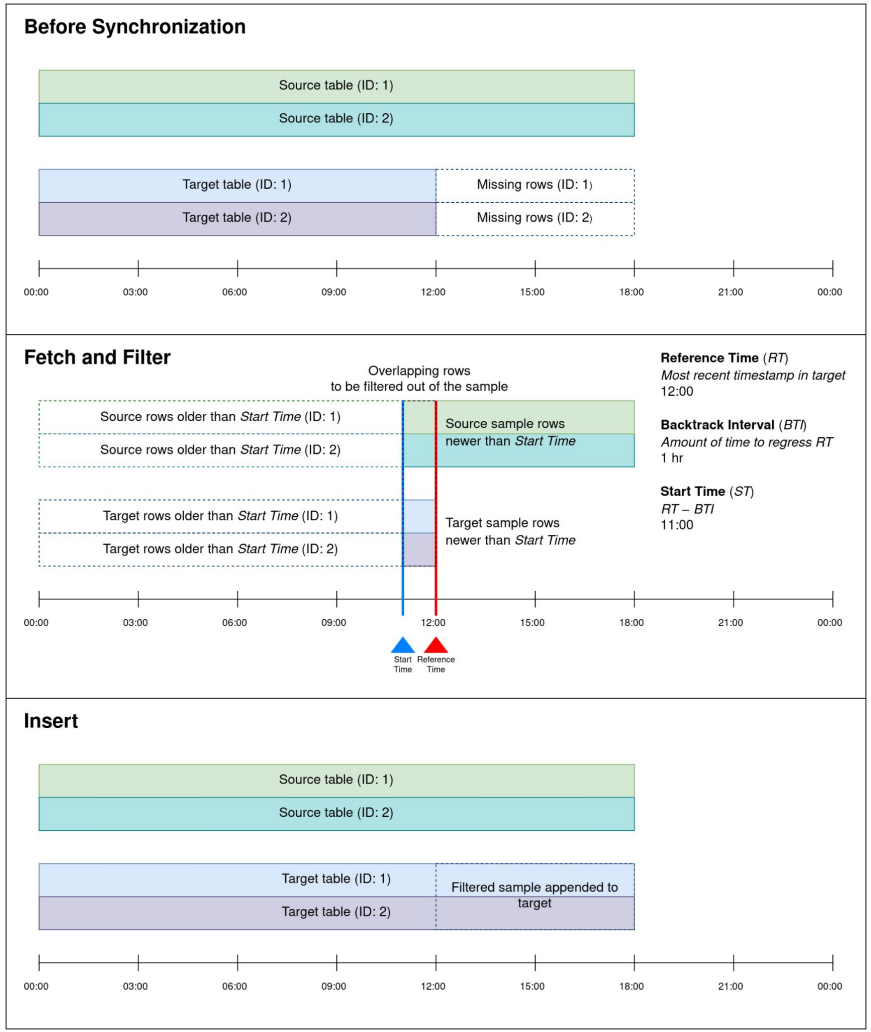
Attributes

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. All sensors report within a known interval of each other.

Simple Backtrack Sync

1. Determine the most recent target datetime as the *RT*.
2. Subtract the *BTI* (e.g. 1 hour) to get the *ST*.
3. Fetch source and target samples newer than *ST*.
4. Filter target rows from source sample.
5. Insert the filtered sample into the target table.

```
SELECT *
FROM source
WHERE datetime >= (
  '2021-01-01 00:00:00' + INTERVAL '1 hour'
)
```



2.2 Backlogged Data

2.2.1 A Single ID with Known Backlogged Data

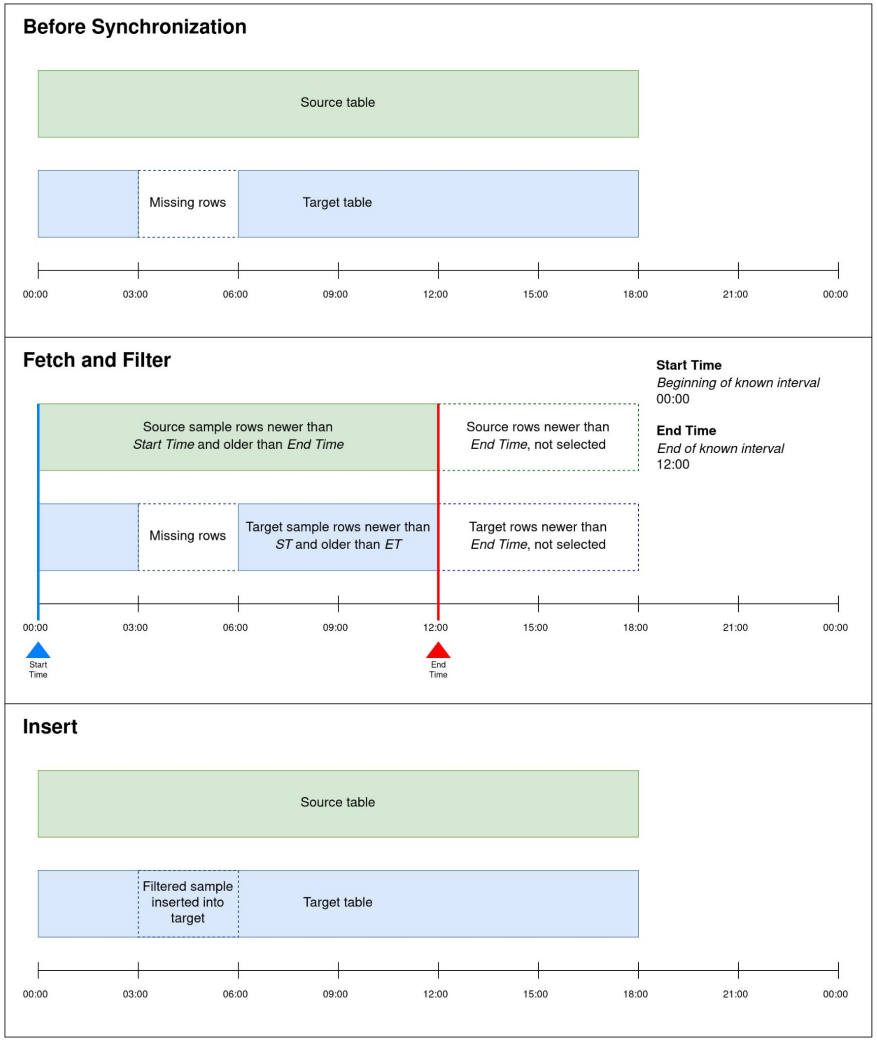
Attributes

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.

Bounded Simple Sync

1. Determine the “start time” (*ST*).
2. Determine the “end time” (*ET*).
3. Fetch source and target samples between *ST* and *ET*.
4. Filter target rows from source sample.
5. Insert the filtered sample into the target table.

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



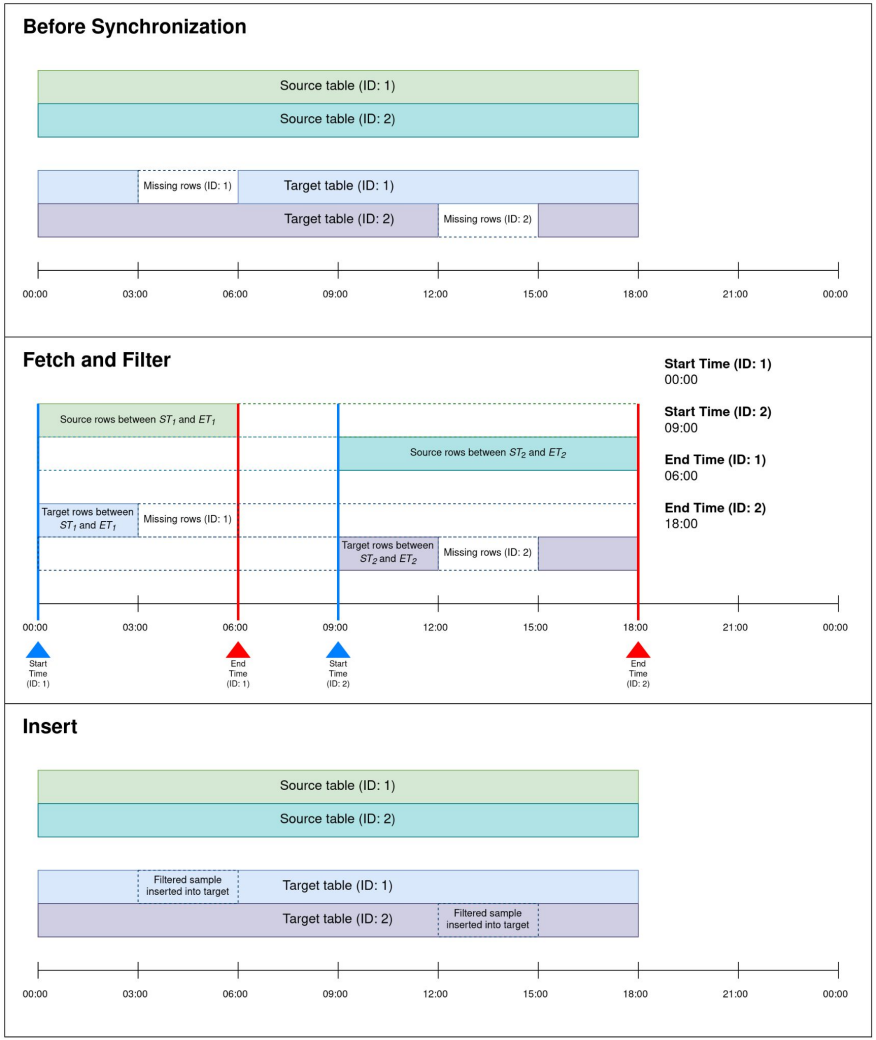
2.2.2 Multiple IDs with Known Backlogged Data

Discontinuous samples may be constructed by bounding fetch queries by ID and datetime.

1. Multiple transactions

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

2. Single transaction
    a. Sub-queries may be appended
    b. Logic in the WHERE clause
    c. Joining on a temporary table



Building a Discontinuous Sample with Multiple Transactions

Pros	Cons
● 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.* ● The time between queries may be spent on other operations. *This time allows the potentially sensitive source database to “cool down.”* ● Fetching and filtering may happen in parallel. *While the syncing service is filtering two samples, more samples may be concurrently fetched.* ● 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.*	● The source table might change between queries. *Because the table is not locked between queries, data may be malformed.* ● The 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.* ● 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.* ● It could overwhelm the database. *If not throttled appropriately, an onslaught of queries could overload the databases’ active connections.*

Building a Discontinuous Sample in a Single Transaction

Appending Sub-queries

```
SELECT *
FROM source
WHERE id = 1
  AND datetime >= ST1
  AND datetime <= ET1
UNION ALL
SELECT *
FROM source
WHERE id = 2
  AND datetime >= ST2
  AND datetime <= ET2
```

Logic in the WHERE Clause

```
SELECT *
FROM source
WHERE (
  id = 1
    AND datetime >= ST1
    AND datetime <= ET1
) OR (
  id = 2
    AND datetime >= ST2
    AND datetime <= ET2
)
```

Joining a Temporary Table

```
WITH bounds AS (
  SELECT *
  FROM (
    VALUES
    (1, ST1, ET1),
    (2, ST2, ET2)
  ) AS t(id, begin, end)
)
SELECT s.*
FROM source AS s
LEFT OUTER JOIN bounds AS b
  ON b.id = s.id
WHERE (
  s.datetime >= st.begin
  AND
  s.datetime <= st.end
) OR st.id IS NULL
```

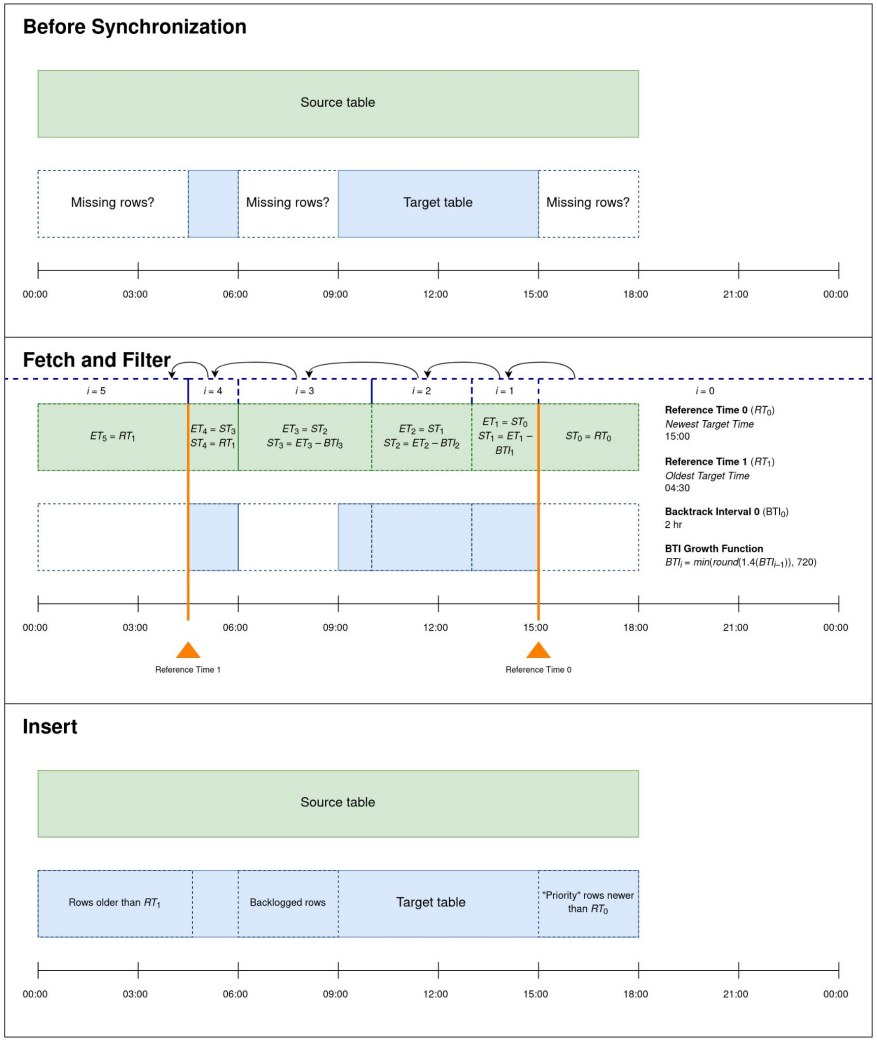
2.2.3 Unknown Backlogged Data

Attributes

1. The data stream has datetime and ID indices.
2. Rows are immutable.
3. Rows with later datetime values are higher priority than those with older datetime values.
4. Backlogged rows are inserted within an unknown interval at unknown times.

Iterative Simple Sync

1. Determine the newest and oldest datetimes in the target table (RT_0 and RT_1).
2. Determine an initial *BTI* (1 hour).
3. Sync rows newer than RT_0.
4. Traverse the datetime axis by growing the *BTI* and syncing intervals.
    a. If possible, skip partitions with identical row-counts.
5. Sync rows older than RT_1.



Chapter 3
Strategies

Methods designed with certain priorities in mind

3.1 Speed First: *Simple Syncs*

Strategy	Description	SQL Example
Simple Sync	Select rows newer than the latest target datetime value.	`SELECT * FROM source WHERE datetime >= '2021-01-01 00:00:00'::TIMESTAMP`
Simple Backtrack Sync	Select rows newer than a “walked back” latest target datetime value.	`SELECT * FROM source WHERE datetime >= ( '2021-01-01 00:00:00'::TIMESTAMP - INTERVAL '1 DAY' )`
Simple Slow-ID Sync	Select rows newer than the oldest datetime of each ID’s latest datetime values.	`WITH sync_times AS ( SELECT id, MAX(datetime) AS sync_time FROM source ) SELECT MIN(sync_time) FROM sync_times`
Simple Append Sync	Generate *Simple Sync* queries for each ID and append them into a single transaction.	`SELECT * FROM source WHERE id = 1 AND datetime >= '2021-01-01 00:00:00'::TIMESTAMP UNION ALL SELECT * FROM source WHERE id = 2 AND datetime >= '2021-01-01 09:00:00'::TIMESTAMP`
Simple Join Sync	Left join a temporary table of latest datetime values to emulate *Simple Sync* per each ID.	`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`

3.2 No-Compromises Accuracy: *Iterative Syncs*

Strategy	Description	SQL Example
Iterative Simple Sync	For each partition of the datetime axis, compare row-counts and perform *Simple Sync* when row-counts differ.	`SELECT * FROM source WHERE datetime >= '2021-01-01 00:00:00'::TIMESTAMP AND datetime < '2021-02-01 00:00:00'::TIMESTAMP`
Daily Row-Count Sync	Build a table of days' row-counts and perform *Simple Sync* on days with differing row-counts.	`SELECT DATE_TRUNC('day', datetime) AS days, COUNT(*) AS rowcount FROM table WHERE datetime >= '2021-01-01 00:00:00'::TIMESTAMP GROUP BY days`
Binary Search Sync	For each partition of the datetime axis, compare row-counts and recursively binary search partitions with different row-counts until sufficiently small intervals are encountered. Perform *Simple Sync* on the small intervals.	`SELECT * FROM source WHERE datetime >= '2021-01-01 00:00:00'::TIMESTAMP AND datetime < '2021-01-02 00:00:00'::TIMESTAMP`
Iterative CPISync	For each partition of the datetime axis, compare row-counts and perform *CPISync* when row-counts differ.	`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)`

Caveats of *CPISync*

1. *CPISync* fundamentally works with integers.
2. *CPISync* can be computationally demanding if the range of values is large.
3. Individual IDs must be synchronized separately.
4. An acceptable interval size depends on the temporal resolution.
    a. The tested implementation uses 1-second resolution.

3.3 Best of Both Worlds: *Corrective Syncs*

Strategy	Description
Simple Monthly Naïve Sync	Perform *Simple Sync* daily and *Naïve Sync* monthly to intermittently “flush” the pipes.
Simple Monthly Iterative Sync	Perform *Simple Sync* daily and *Iterative Simple Sync* monthly to catch backlogged rows.
Simple Monthly Daily Row-Count Sync	Perform *Simple Sync* daily and *Daily Row-Count Sync* monthly to catch backlogged rows.
Simple Monthly Binary Search Sync	Perform *Simple Sync* daily and *Binary Search Sync* monthly to catch backlogged rows.
Simple Monthly Iterative CPISync	Perform *Simple Sync* daily and *Iterative CPISync* monthly to catch backlogged rows.

Chapter 4
Experimental Results

Strategies' performances in a simulated environment

Performance Metrics

1. **Run-time**
   The duration in seconds of each synchronization.

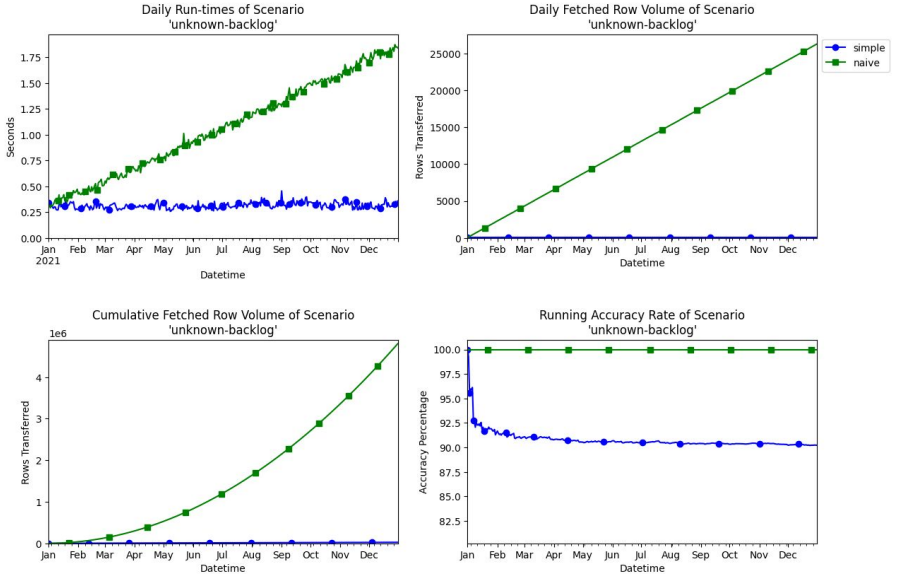
2. **Bandwidth**
   The number of rows fetched from the source database.

3. **Accuracy**
   The ratio of the number of correctly synchronized rows to the number of all source rows.

Figure Interpretation

Daily metrics line graphs

The daily line graphs represent the strategies' daily performances to illustrate their behaviors.



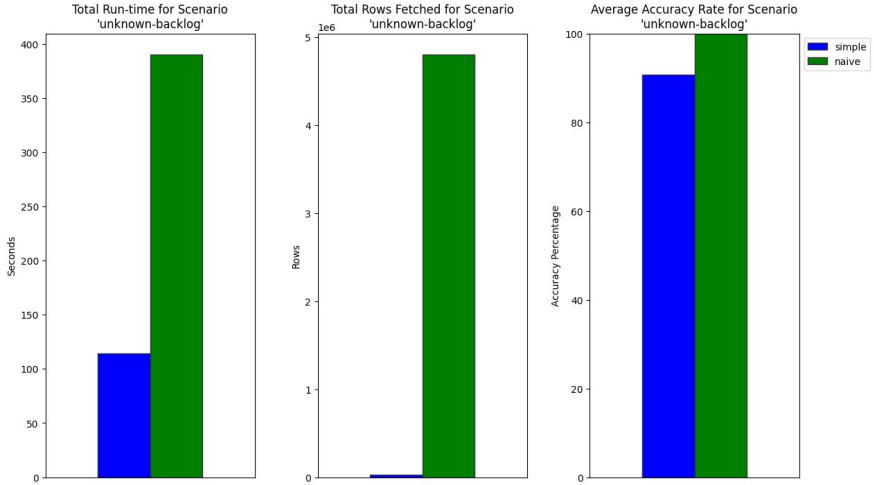
"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.

Figure Interpretation

Summary bar charts

The bar charts compare the aggregated values of the daily line graphs:

- Total number of seconds
- Total number of rows fetched
- Average accuracy rate



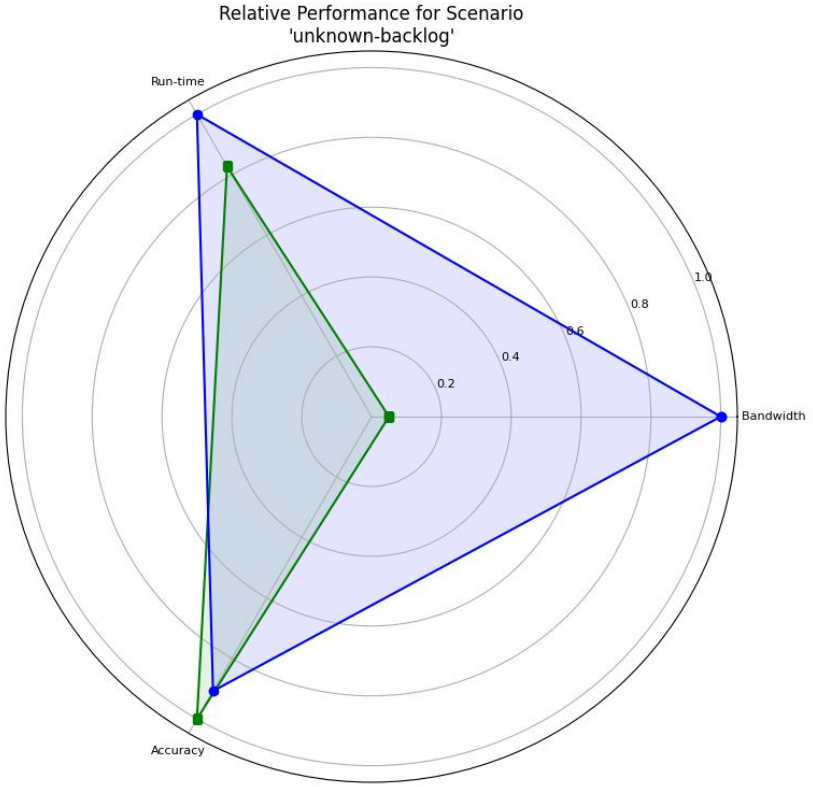
“Lower is better” for the total run-time and rows fetched and “higher is better” for the average accuracy rate.

Figure Interpretation

Summary radar charts

The radar charts are structured as intuitive visual representations of the “skills” of each technique.

- Accuracy is the same (0 to 100%)
- Run-time and bandwidth are normalized to a scale between the performance of Simple Sync and fifteen times worse performance.

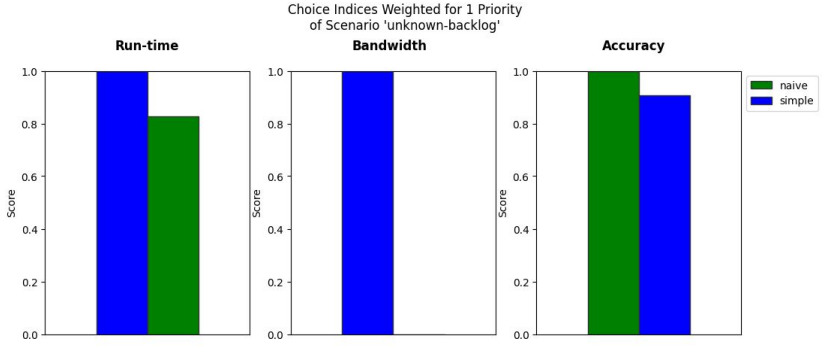


“Higher is better” for every metric.

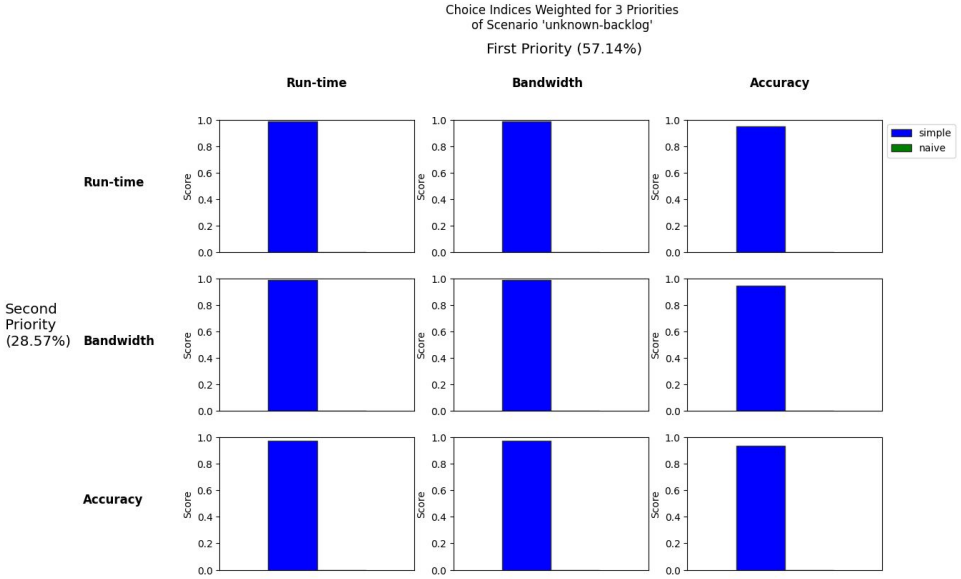
Figure Interpretation

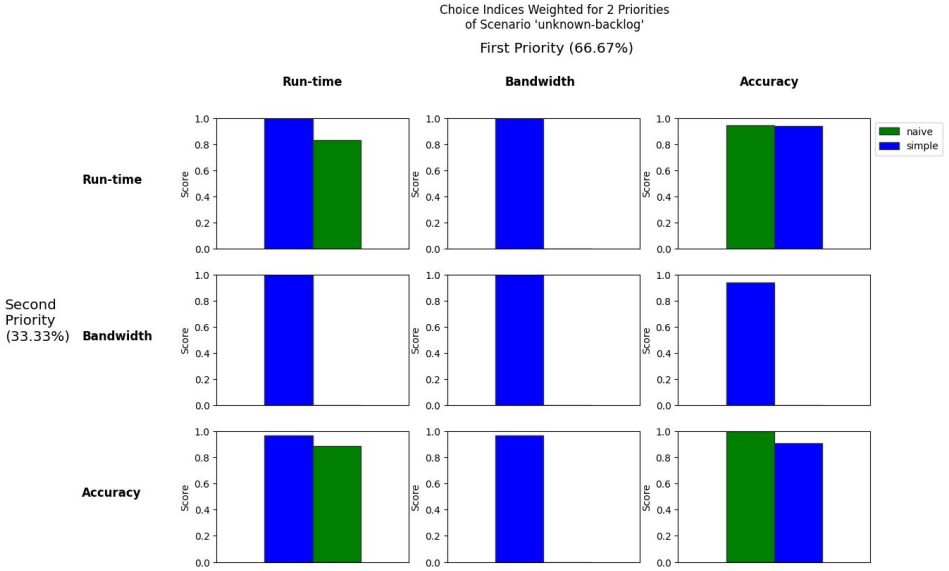
Choice Index bar charts

The *Choice Index* is the weighted average of the normalized values presented in the summary radar charts.



"Higher is better" for every metric.





Tested Scenarios

1. **Single append-only**
   a. A single ID grows a table by one record per hour for 1 year.
   b. Nothing is ever backlogged.

2. **Multiple (large-N) append-only**
   a. A large number of IDs (100) grow a table one record per ID per hour for 1 year.
   b. Nothing is ever backlogged.

3. **Single known backlog**
   a. A single ID regularly grows a table, with each record having a probability of an “outage.”
   b. Records which are detected as occurring during the “outage” are later backlogged into the source table within a known interval (24 hours).

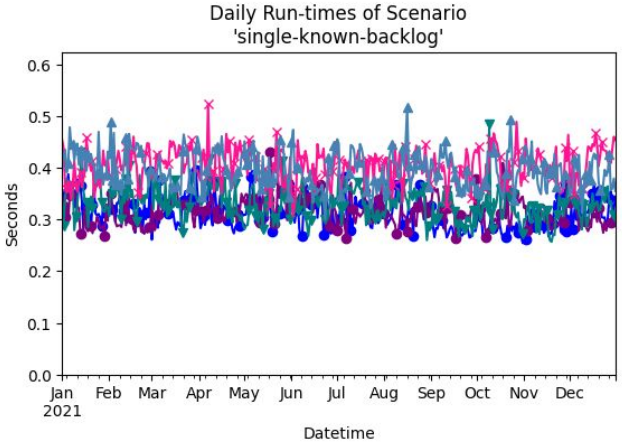
4. **Unknown backlog**
   a. An unknown number of IDs (3) grow a table by an unknown frequency (one record per ID per hour for 1 year).
   b. Records are backlogged with an unknown frequency over an unknown interval.

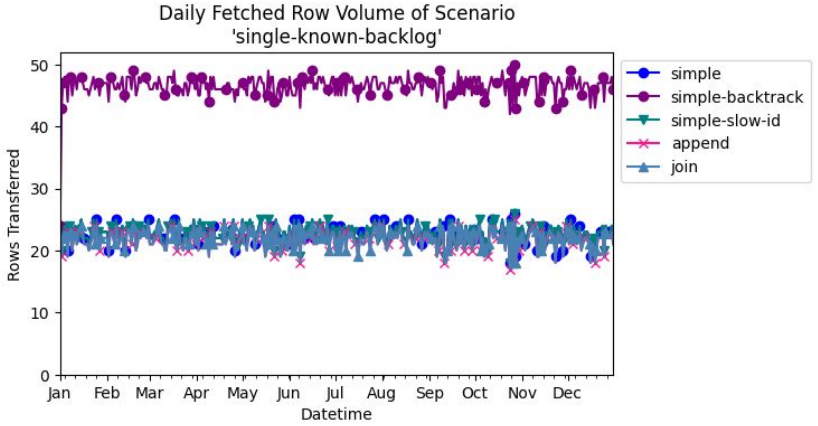
4.1 Comparing Classes of Strategies

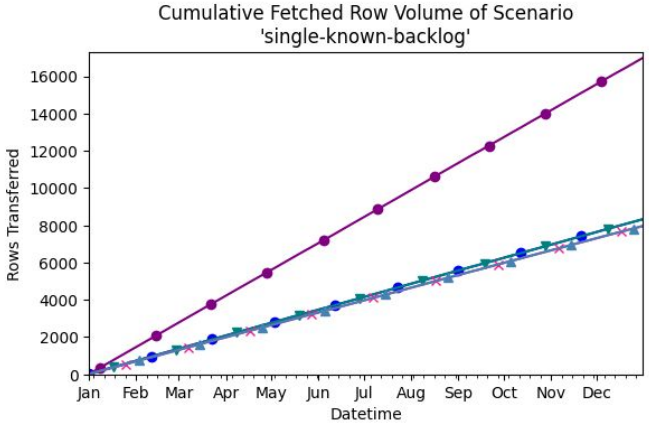
Simple Syncs

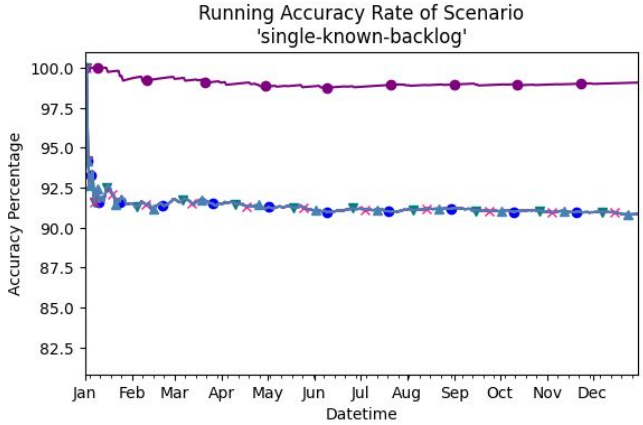
Simple Syncs Daily Performance

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.



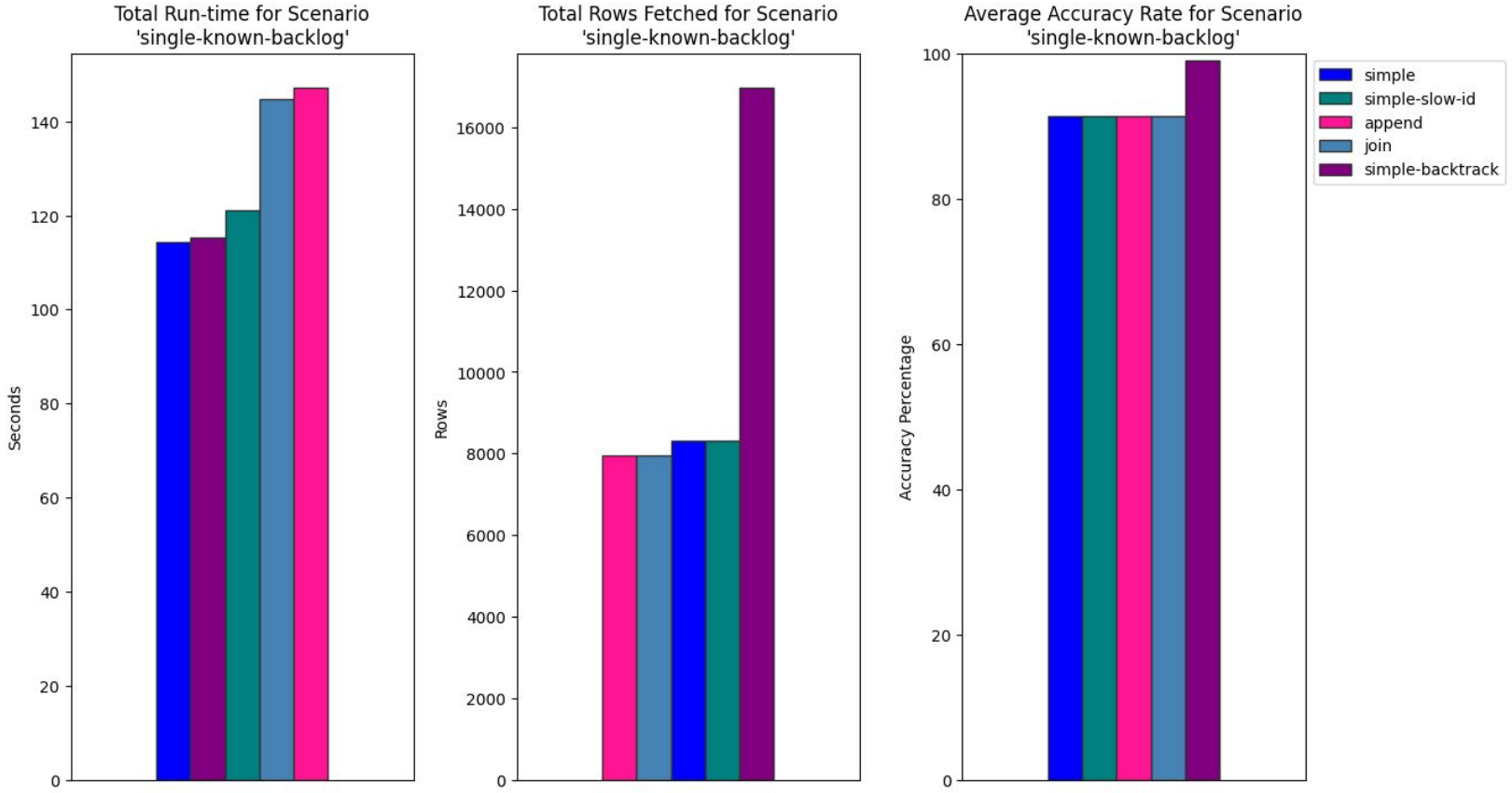






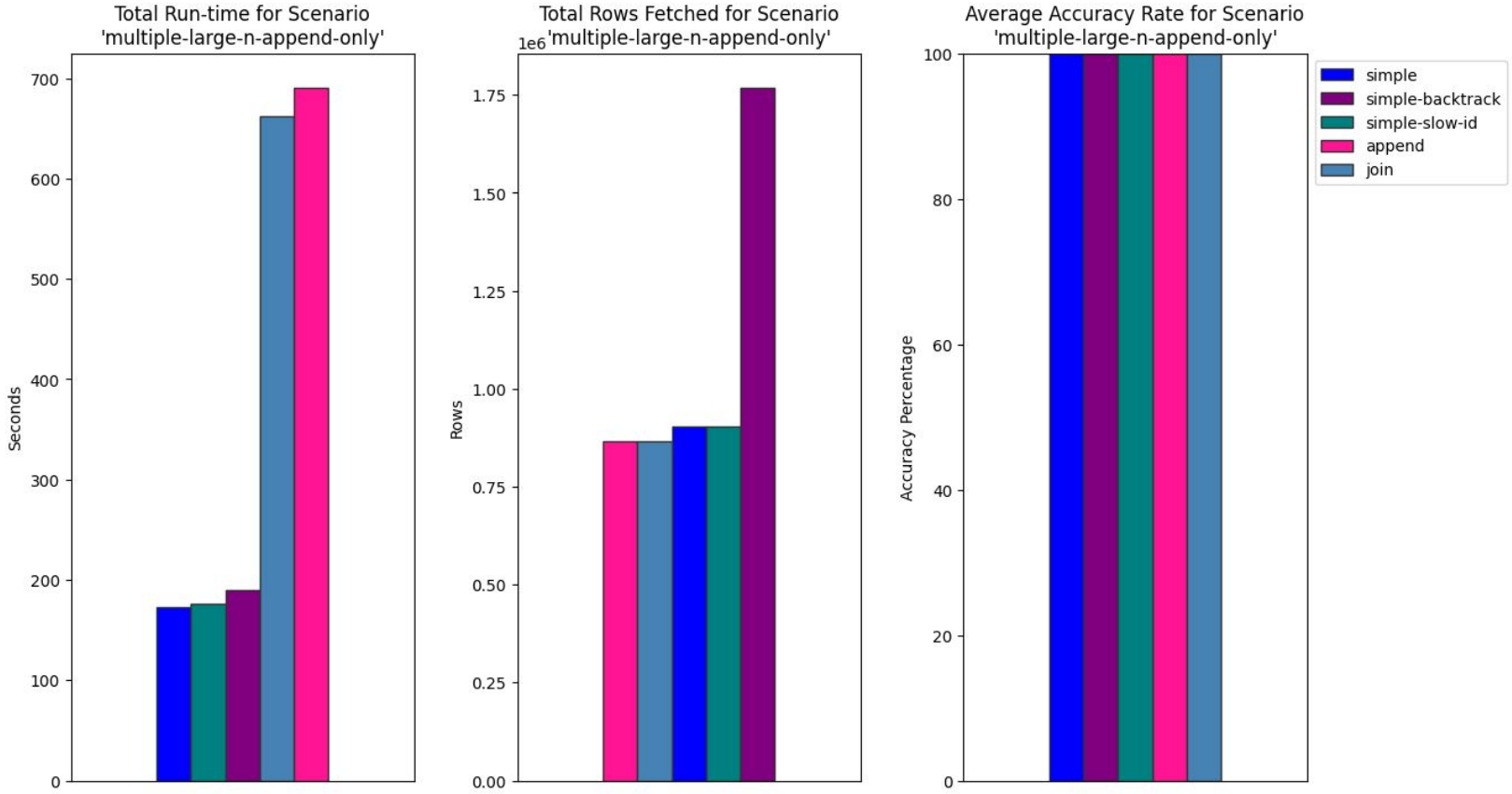
Simple Syncs Performance Summary

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.



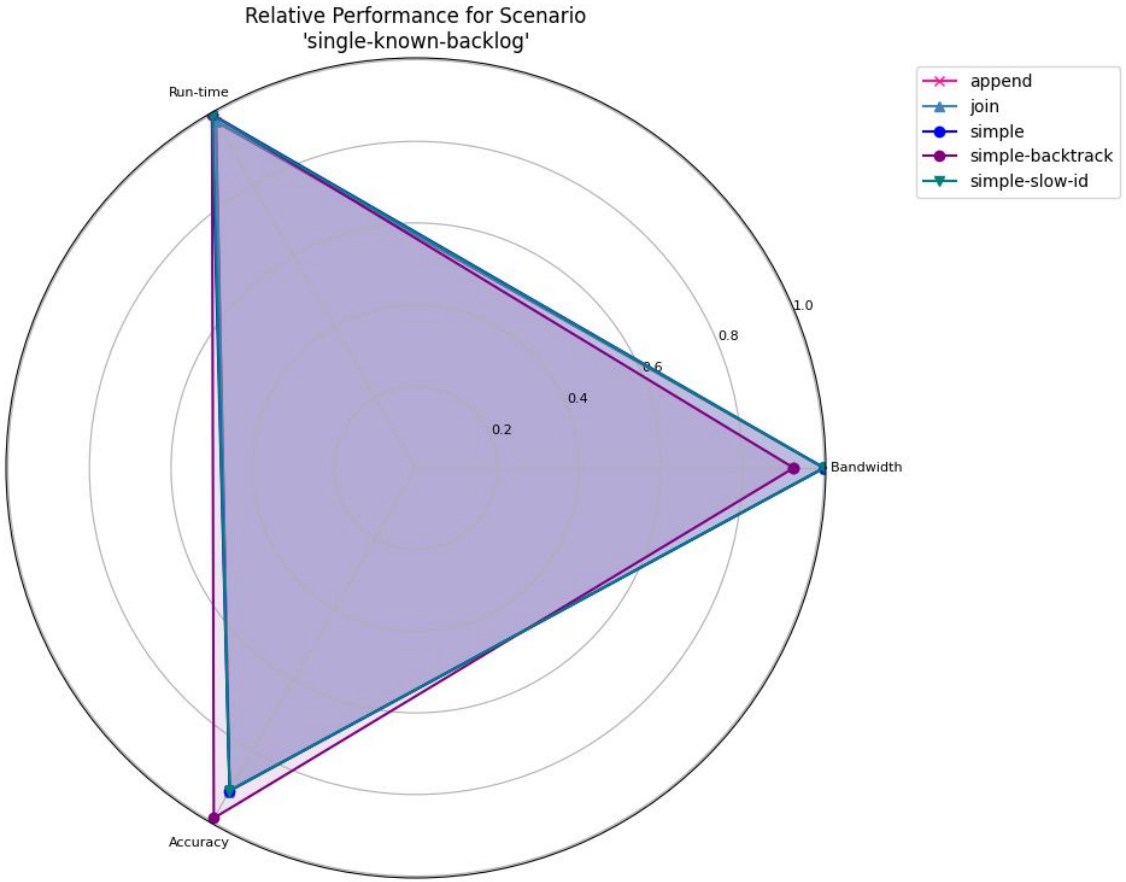
Simple Syncs Performance Summary (large N)

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.



Simple Syncs Relative Performances

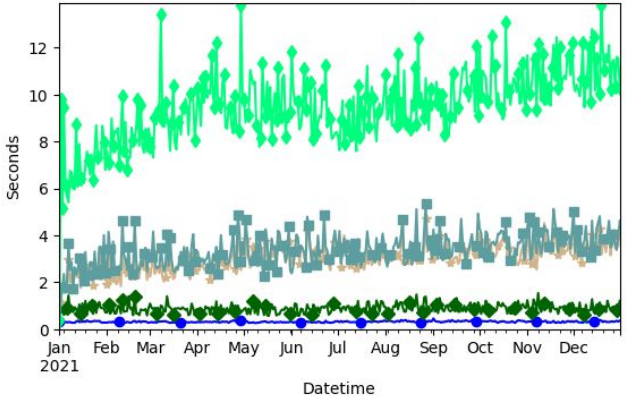
"Higher is better" for every metric.

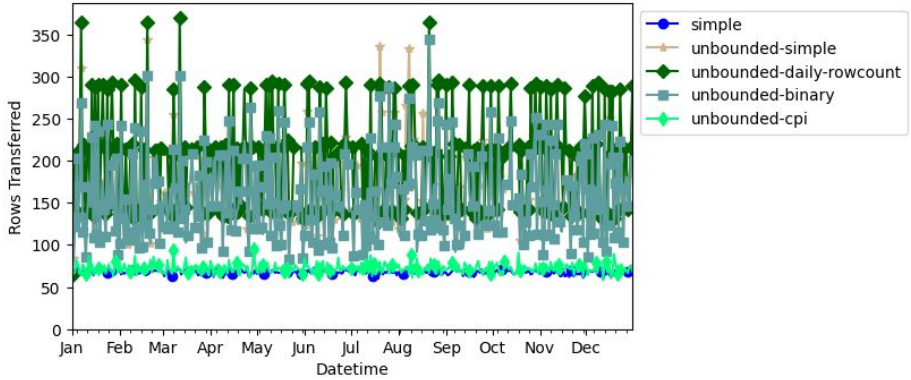


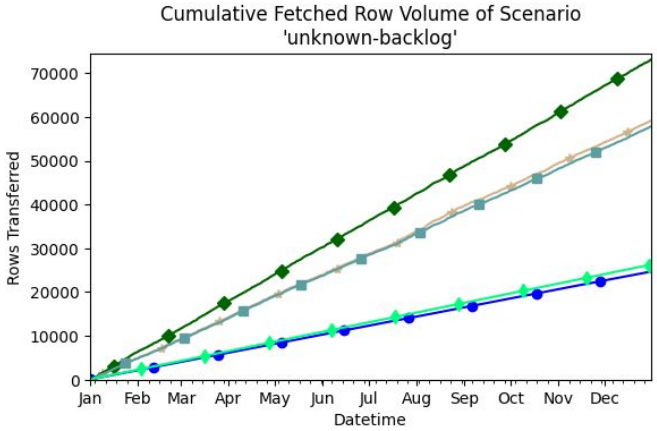
Iterative Syncs

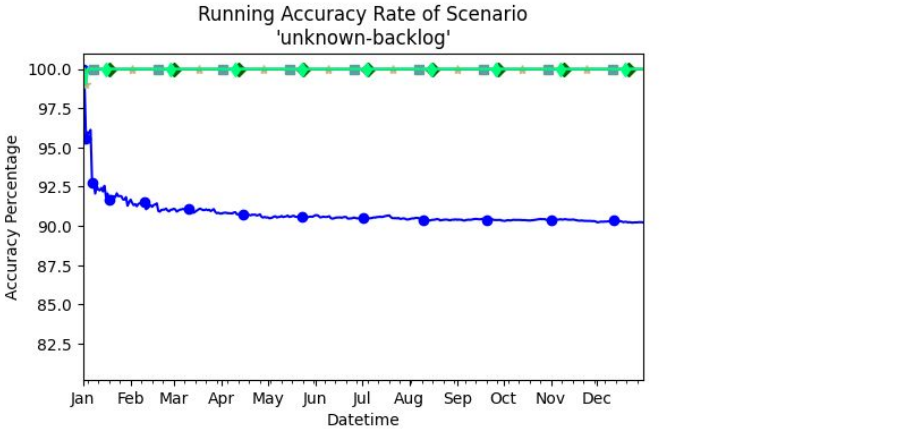
Iterative Syncs Daily Performance

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.



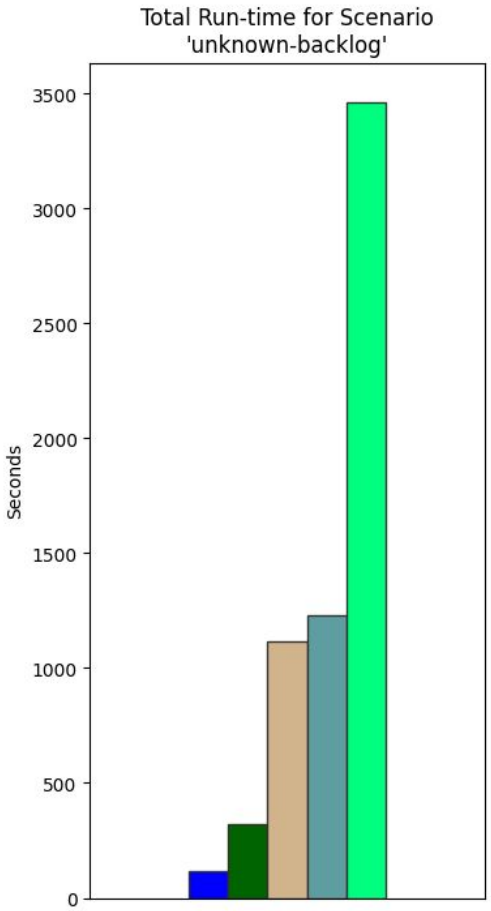


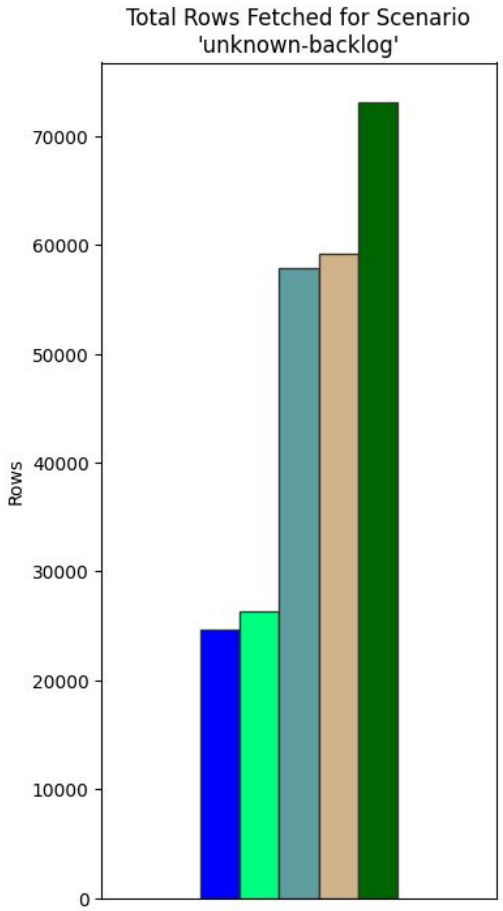


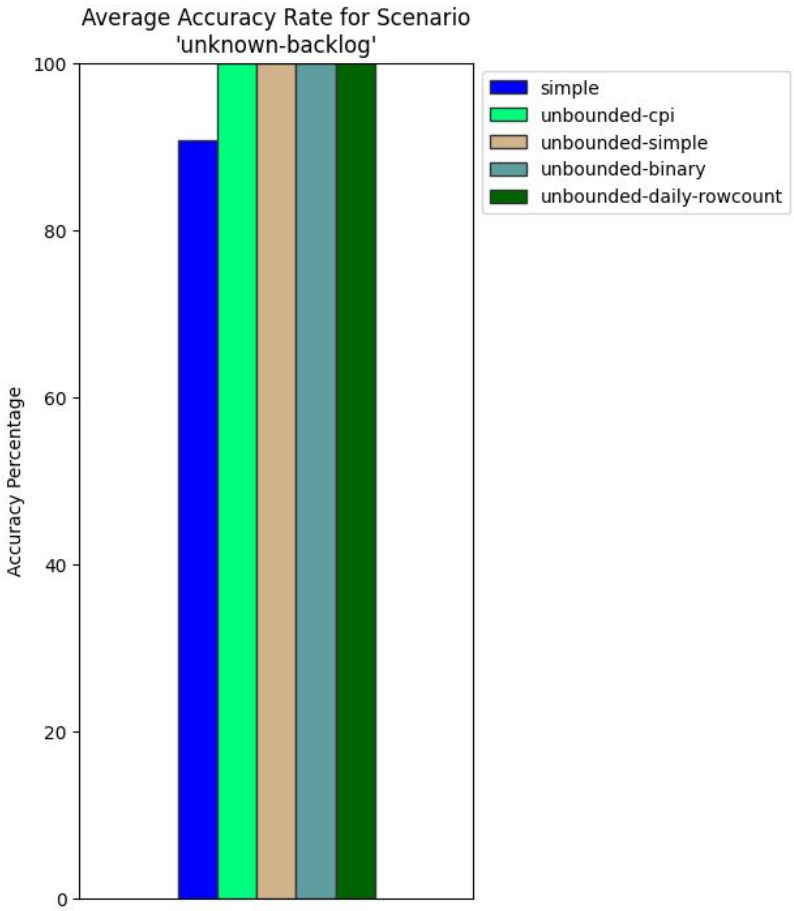


Iterative Syncs Performance Summary

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.

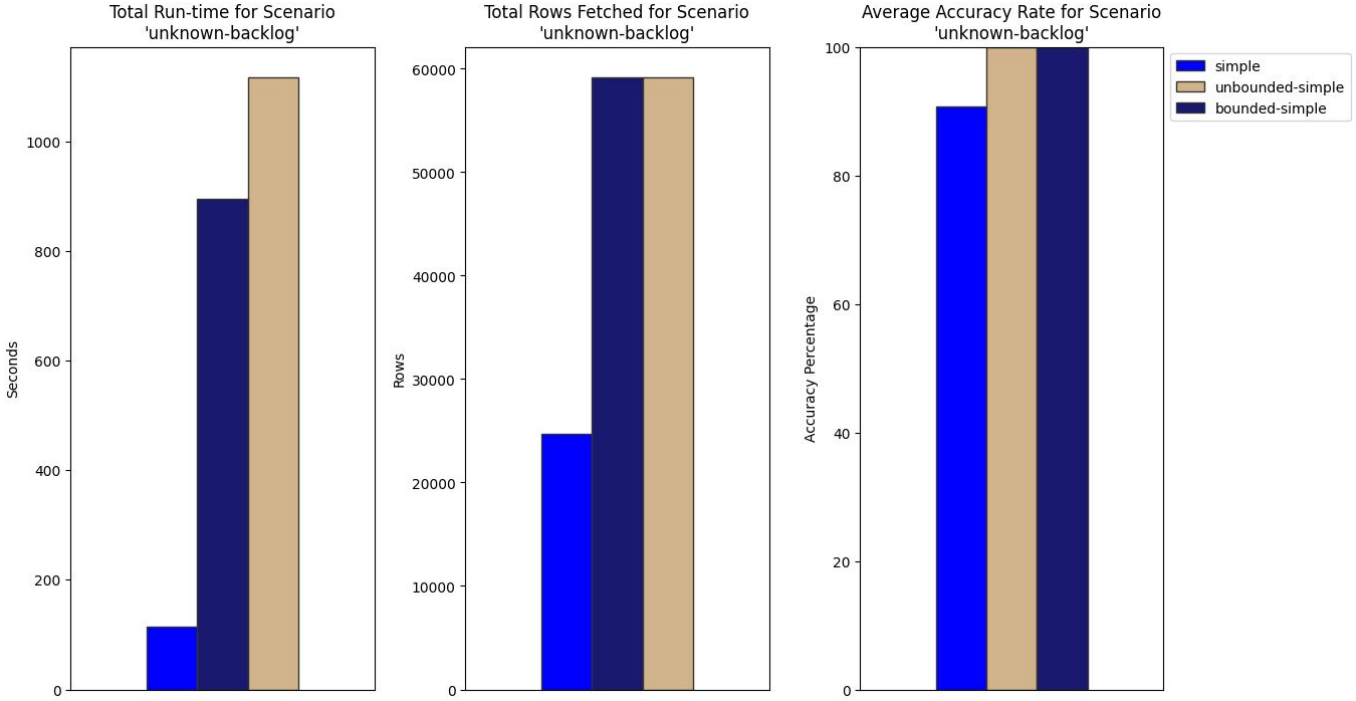






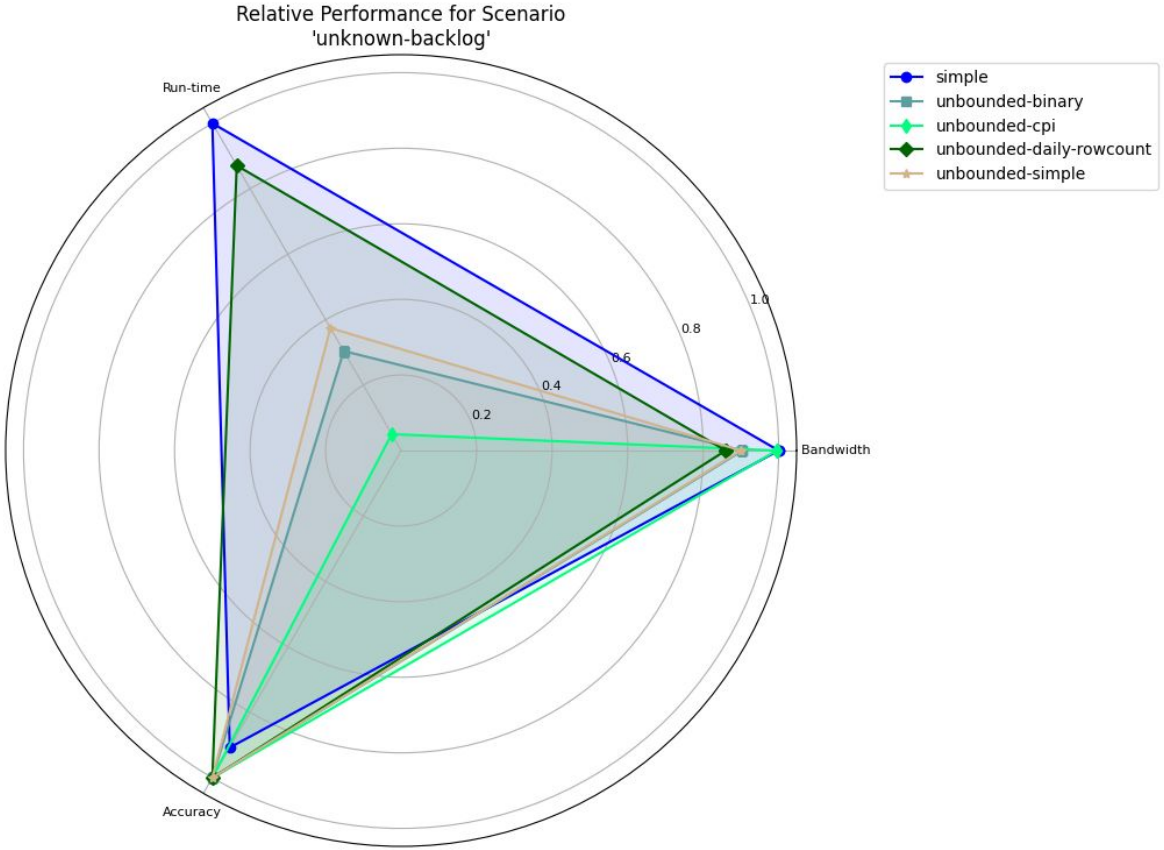
Bounded vs Unbounded *Iterative Syncs*

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.



Iterative Syncs Relative Performances

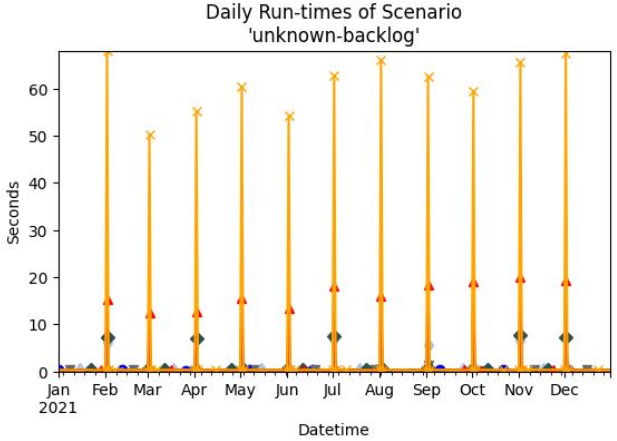
"Higher is better" for every metric.

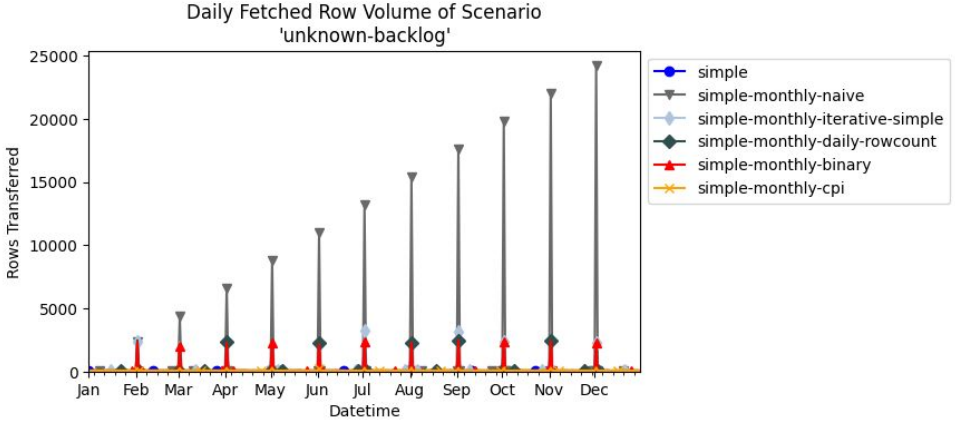


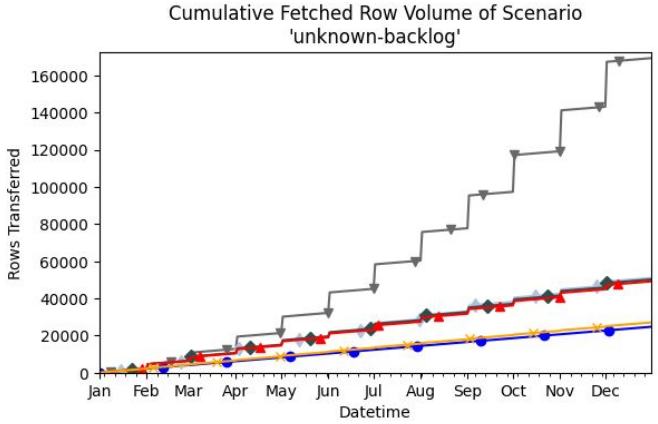
Corrective Syncs

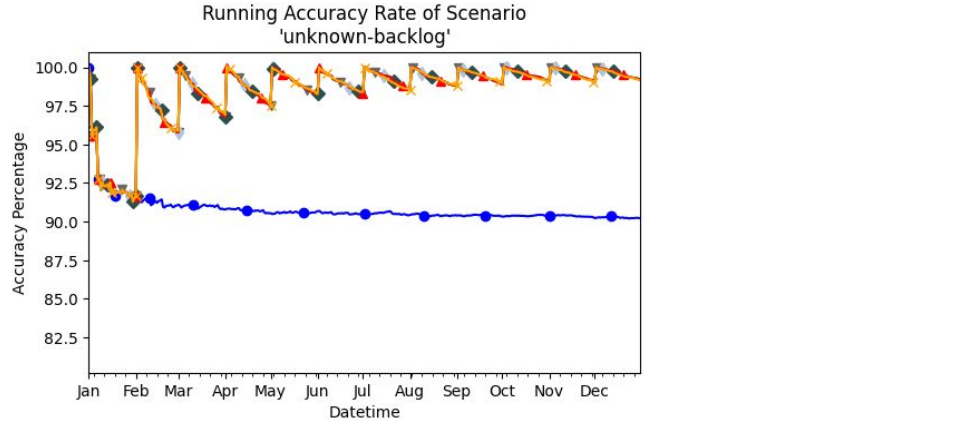
Corrective Syncs Daily Performance

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.



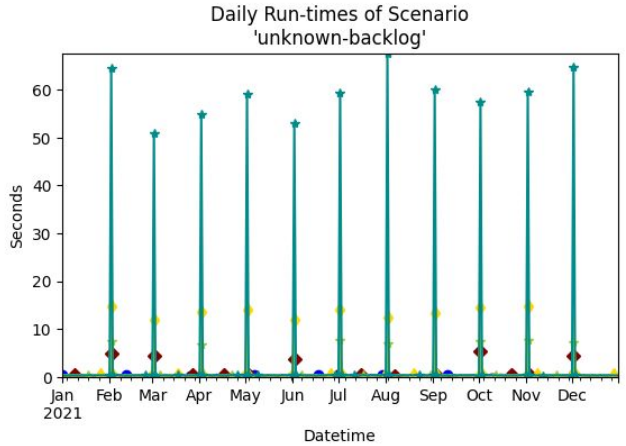


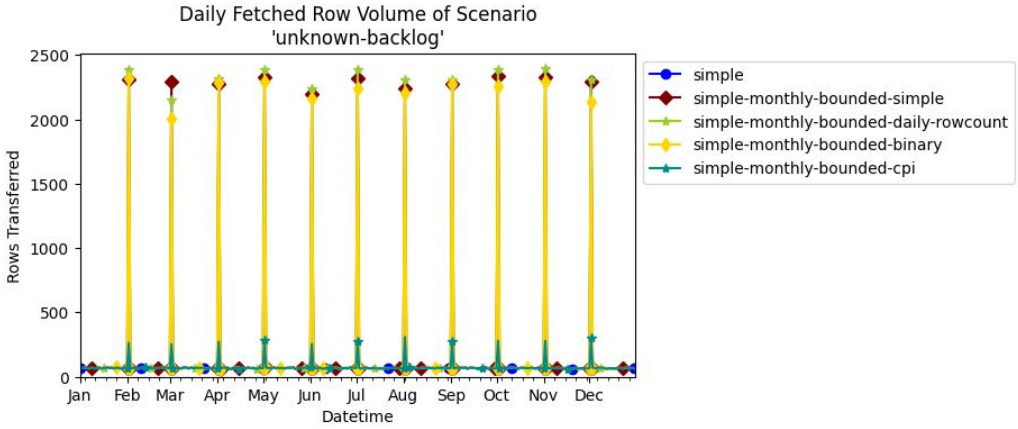


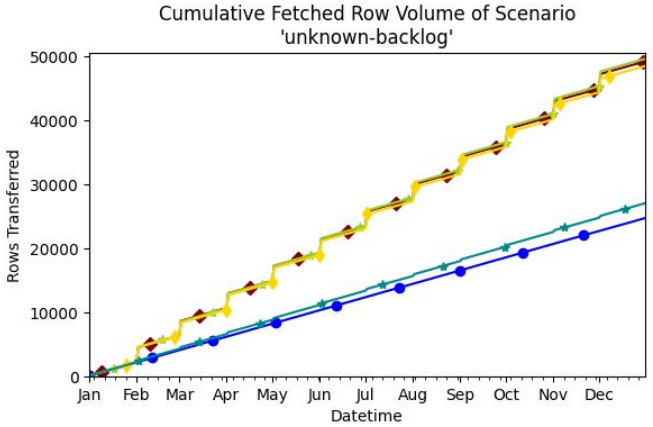


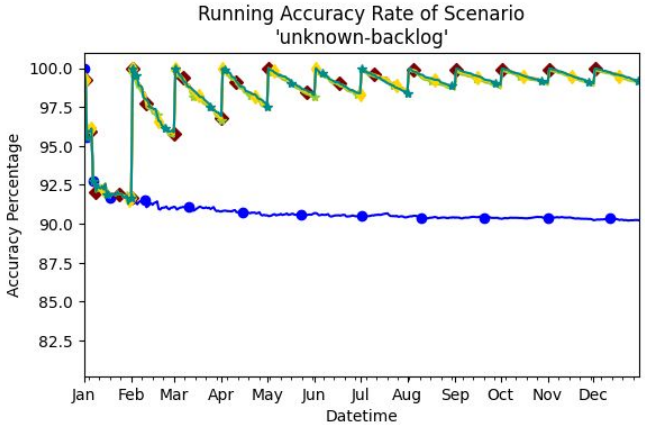
Corrective Syncs Daily Performance (bounded)

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.



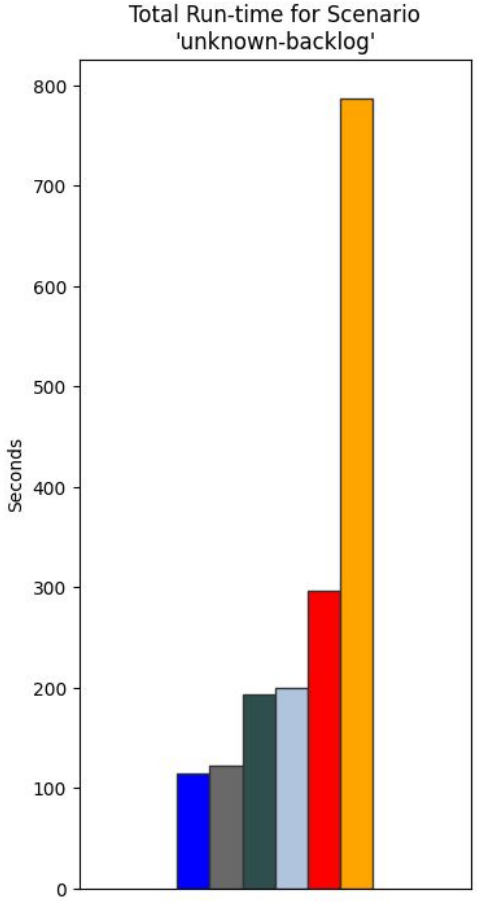


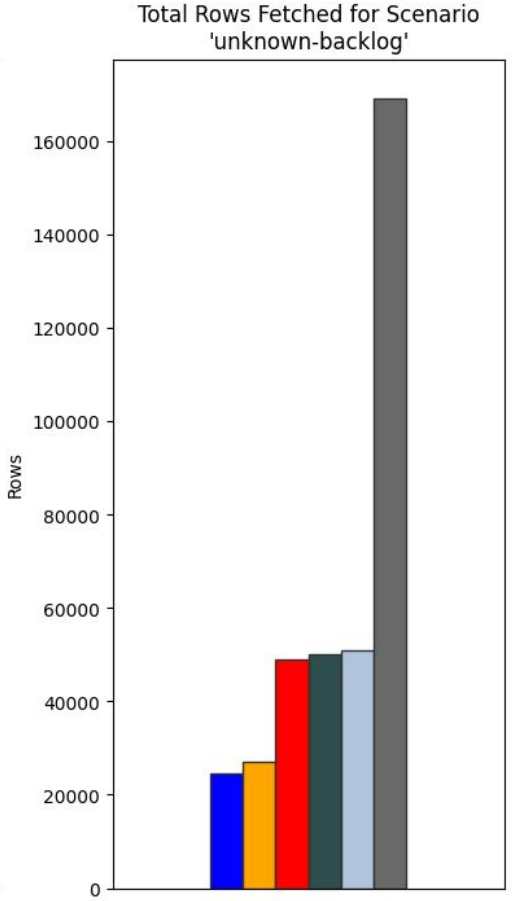


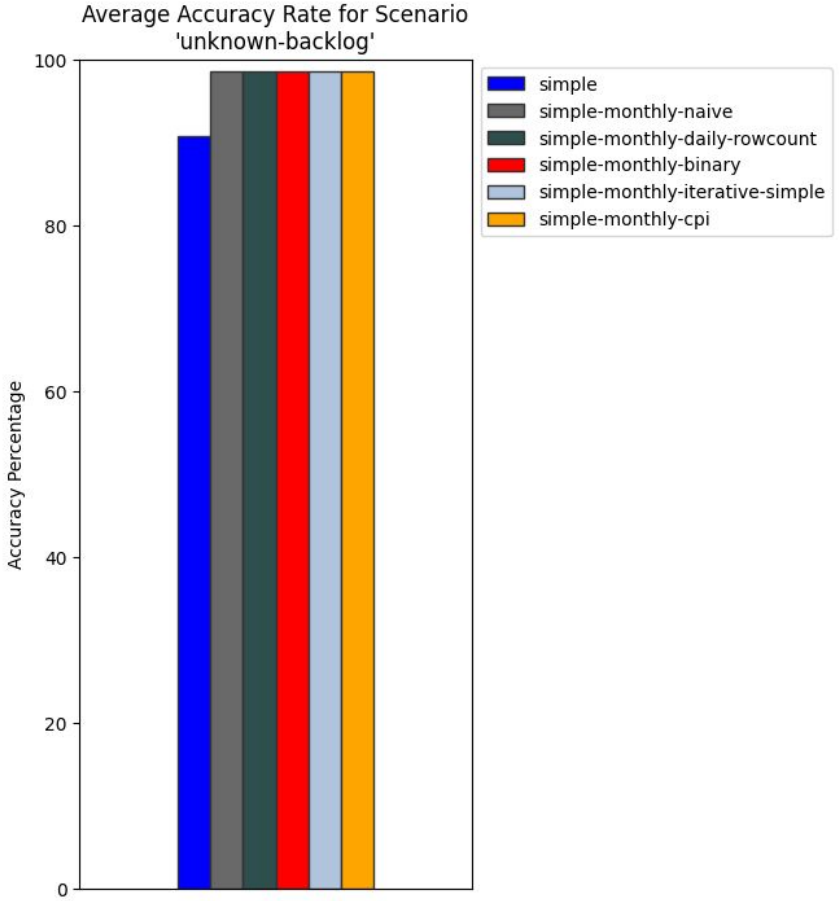


Corrective Syncs Performance Summary

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.

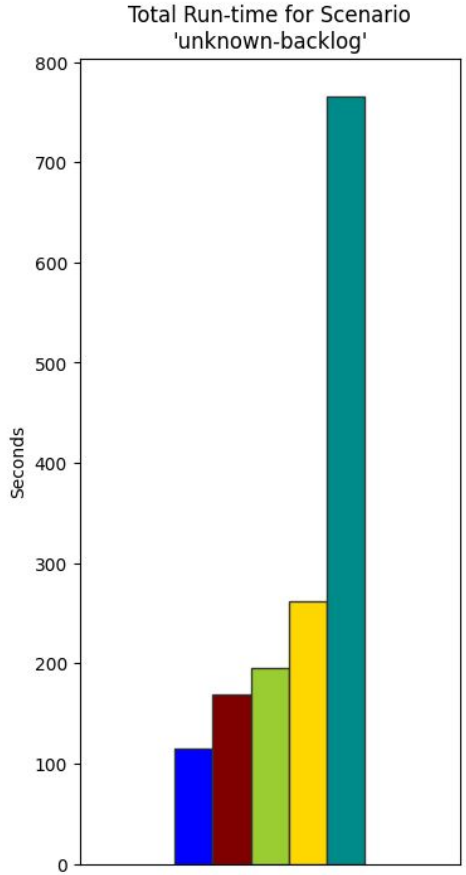


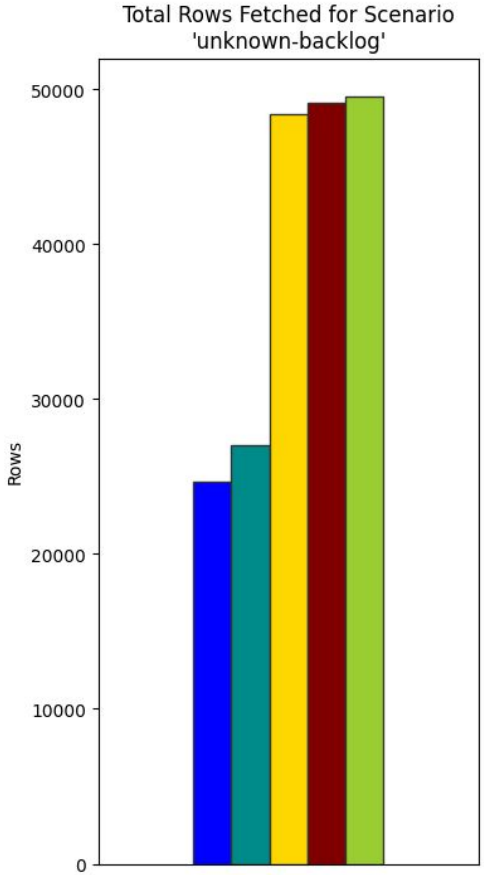


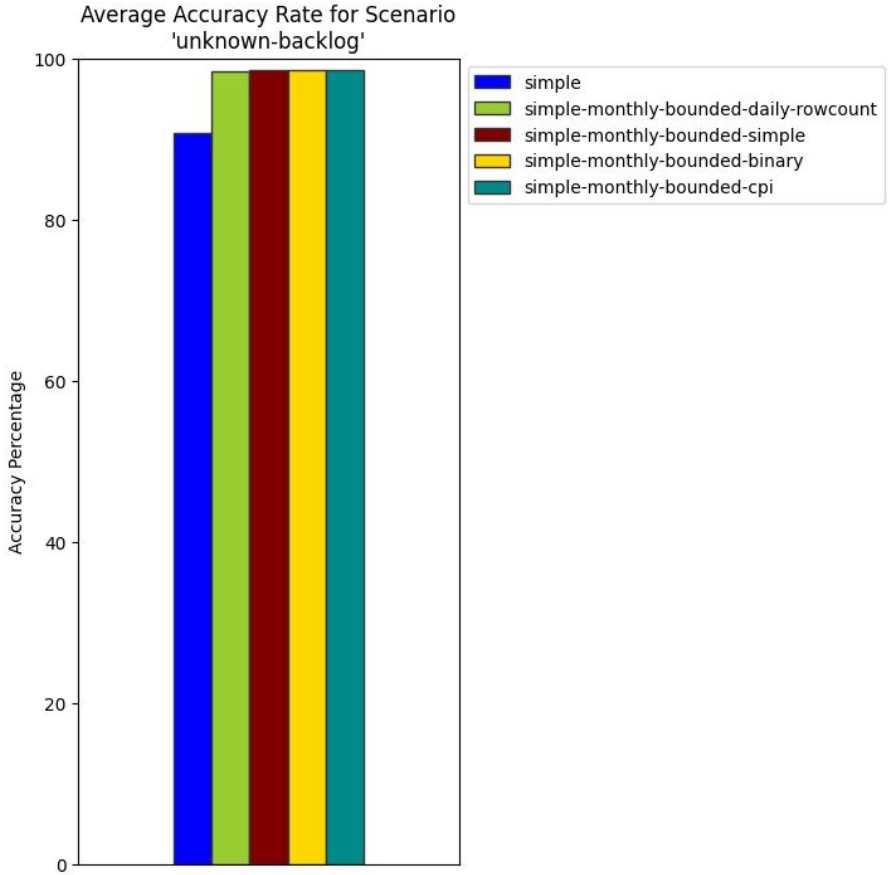


Corrective Syncs Performance Summary (bounded)

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.

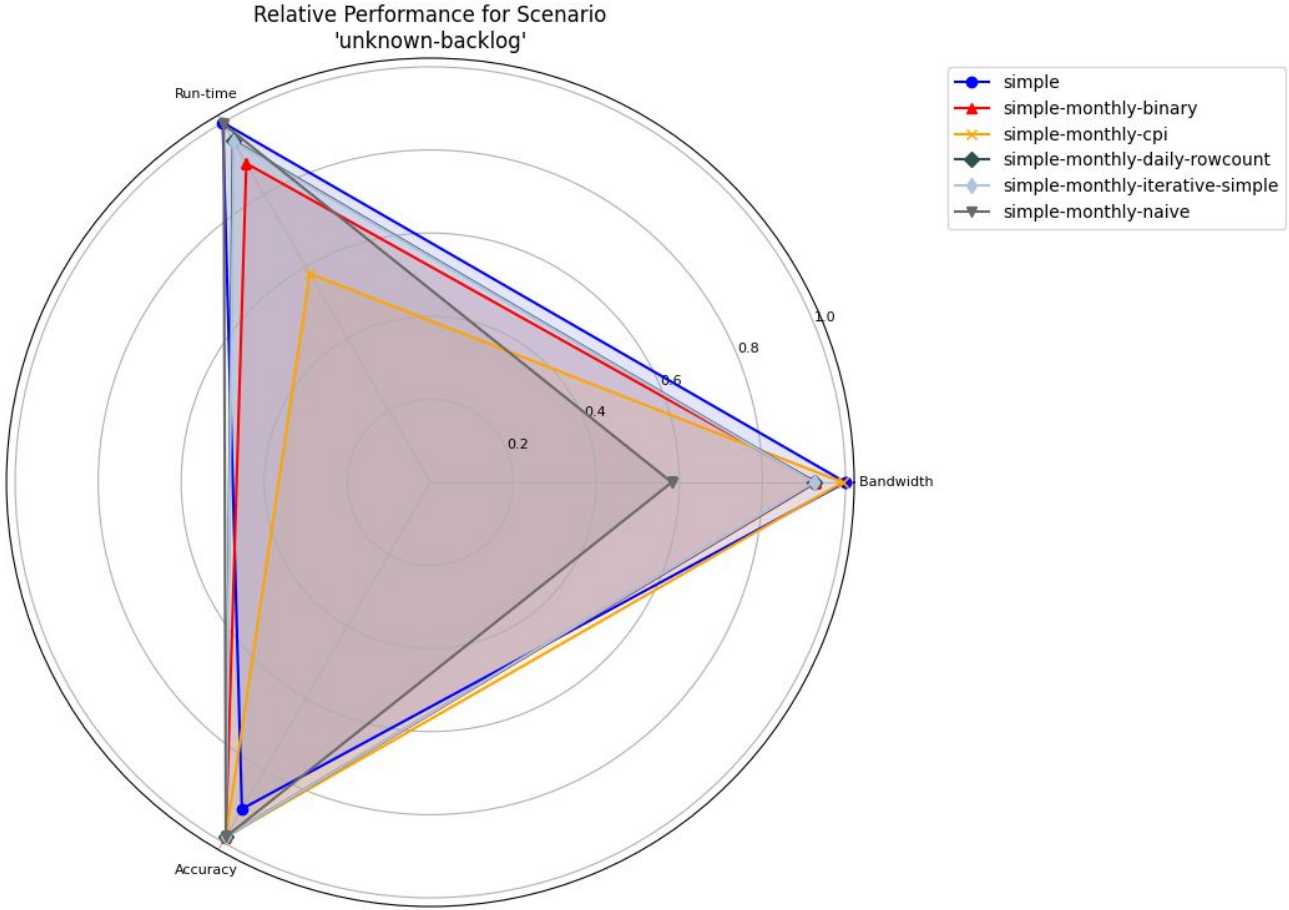






Corrective Syncs Relative Performances

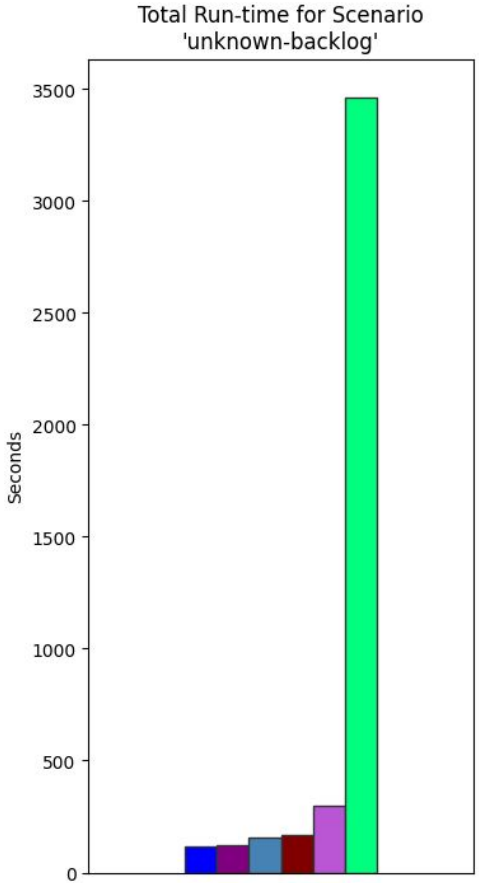
"Higher is better" for every metric.

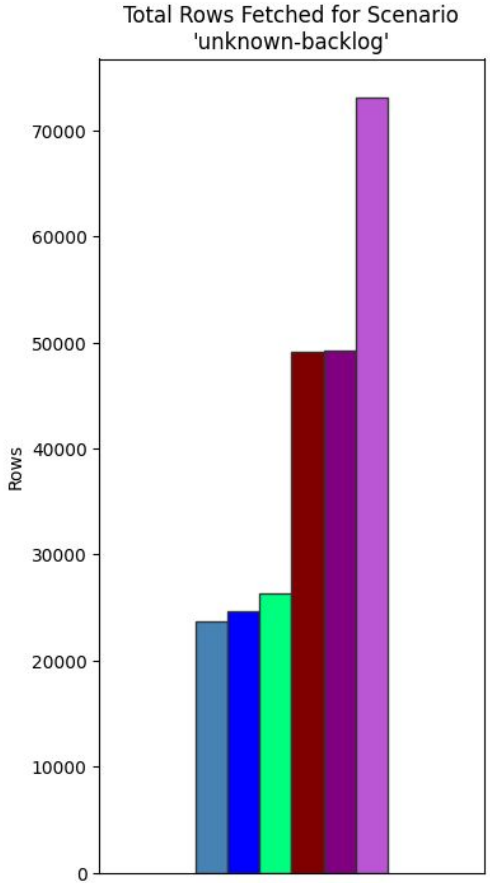


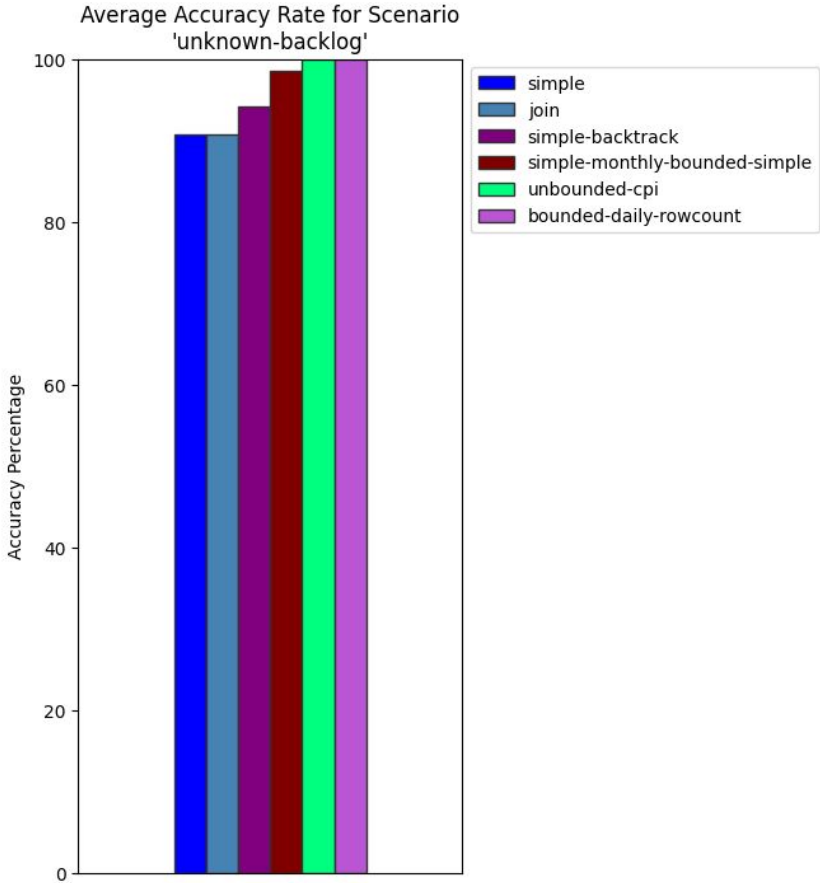
4.2 Ranking Strategies

Limitations of the Summary Bar Charts

"Lower is better" for run-time and bandwidth.
"Higher is better" for accuracy.

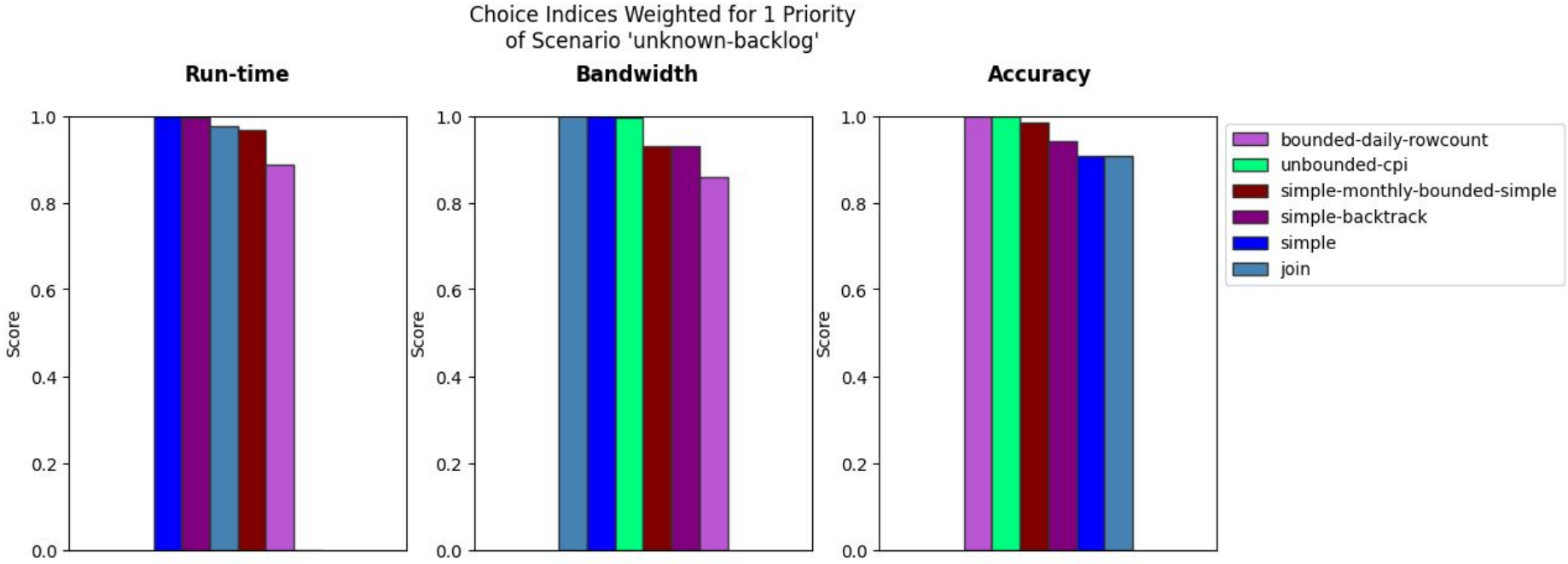






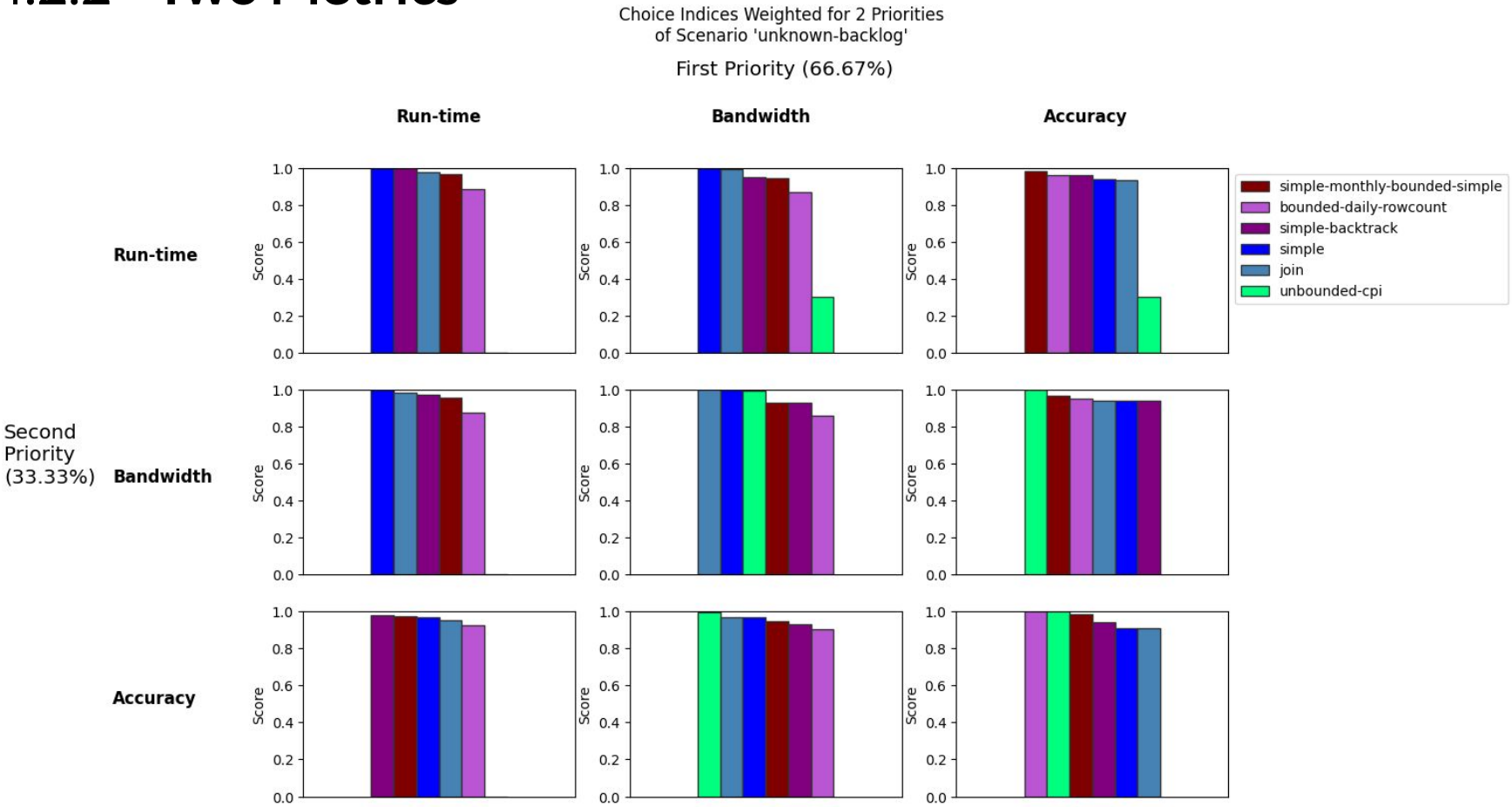
4.2.1 One Metric

"Higher is better" for every metric.



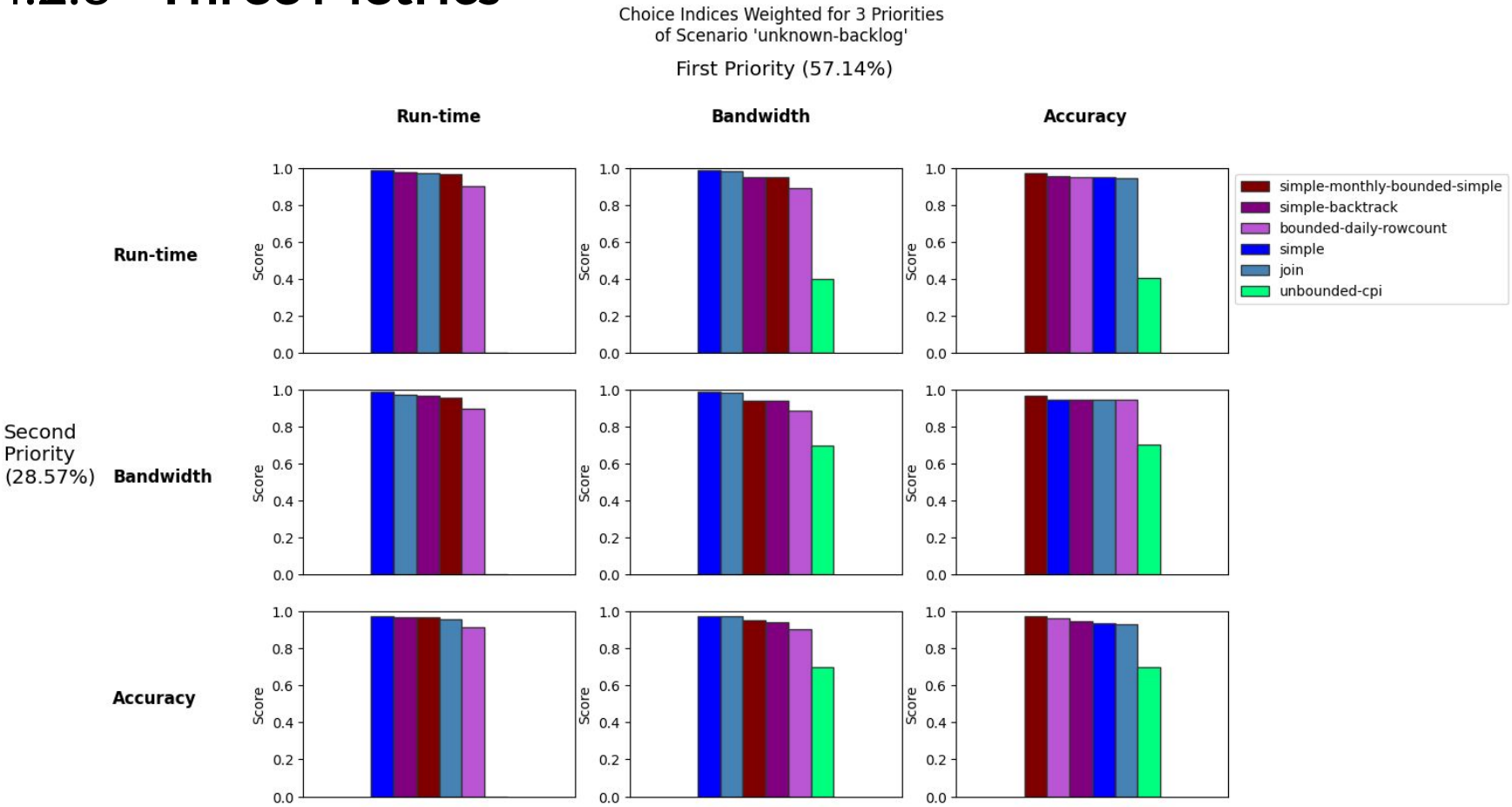
4.2.2 Two Metrics

"Higher is better" for every metric.



4.2.3 Three Metrics

"Higher is better" for every metric.



Chapter 5
Conclusion

Summarizing our findings and looking forward

Choosing a Strategy

Identify the Scenario

1. *How many IDs do you have?*
2. *Do you know when data are backlogged, if at all?*
3. *What is the frequency of source database?*

- Append Only
  - Single ID: *Simple Sync*
  - A few IDs: *Simple Join Sync*
  - Many IDs: *Simple Sync*
- Known Backlog
  - *Simple Backtrack Sync*
- Unknown Backlog
  - It depends.

Rank Your Priorities

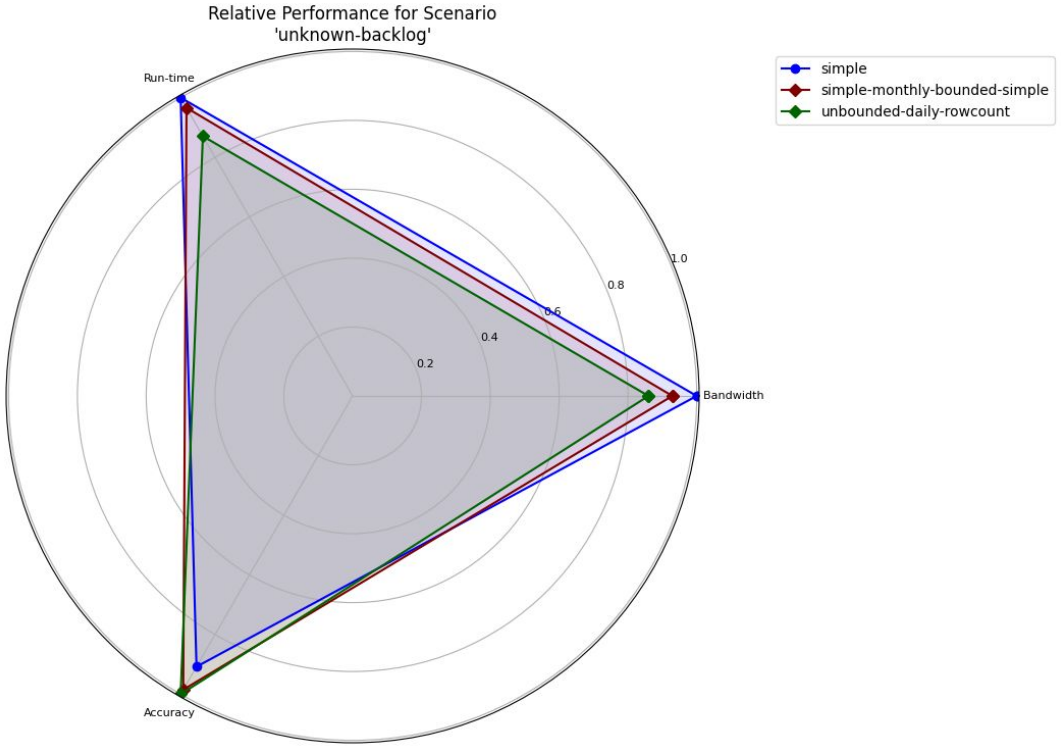
1. *Is perfect accuracy a requirement?*
2. *How cheap is bandwidth?*
3. *Will you be synchronizing often?*

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

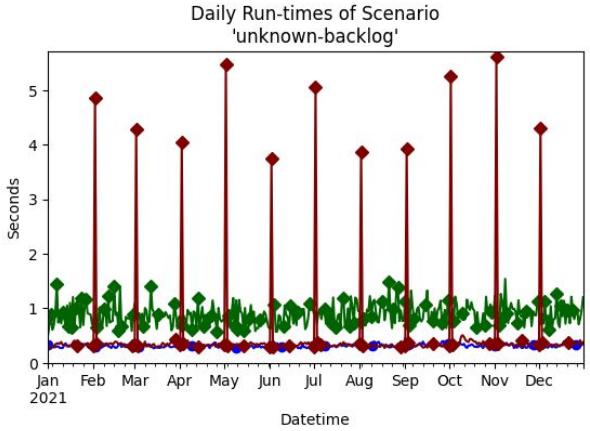
Table 5.1: Strategy recommendation chart for situations similar to *unknown-backlog*

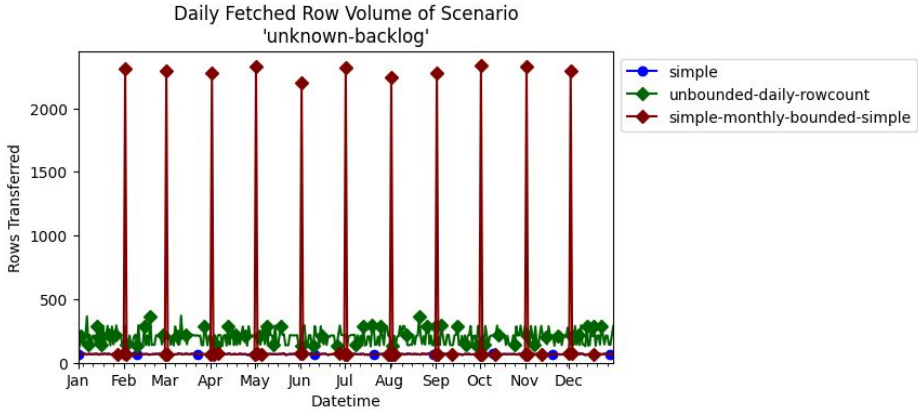
Summary

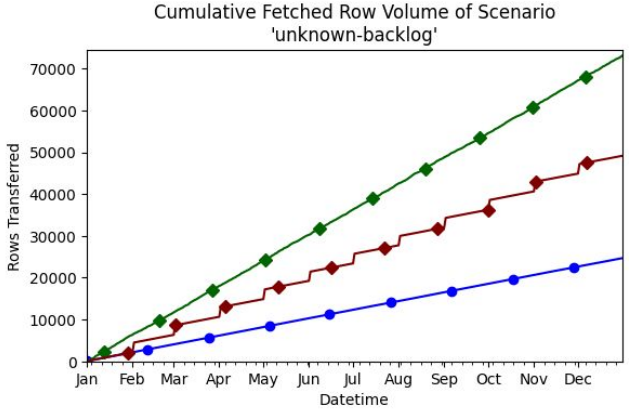
- ***Simple Sync***
  - Minimal run-time and bandwidth
  - Decent accuracy
- ***Daily Row-Count Sync***
  - Minimal run-time and bandwidth
  - Perfect accuracy
- ***Simple Monthly Iterative Simple Sync* (bounded)**
  - Balance between run-time, bandwidth, and accuracy

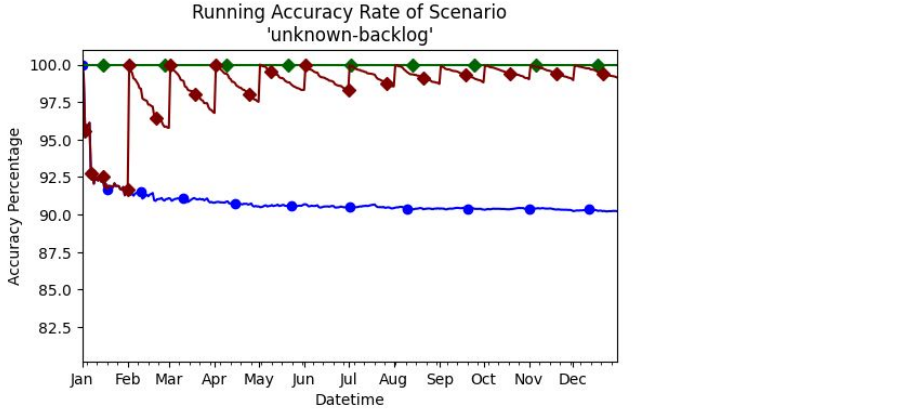


Summary (visualized)









Future Work

- Additional Strategies
  - Fetch Strategies
  - Filter Strategies
  - Insert Strategies
  - Further Optimizations
- Time-Series Properties
  - Mutability
  - Frequency, Timescale, Resolution
- Experiment Design
  - Evaluation Metrics
  - Scenario Simulations