List of Benchmark Queries

Contents

A. List of Queries

This chapter presents the complete set of benchmark queries designed for evaluating the latency characteristics of the Odysseus stream processing system. Each query focuses on a specific aspect of stream processing, such as ingestion latency, operator overhead, windowing impact, or multi-stream coordination. The queries were executed across multiple configurations to ensure a comprehensive evaluation of performance under varying conditions.

Benchmark Configurations

To systematically analyze the influence of stream rate and operator complexity on query latency, each query was executed under four distinct data rate scenarios. These configurations differ in total tuple volume, per-stream emission rates, and the auction duration, simulating environments from light to high-throughput conditions.

Scenario	**Tuples**	**Bid (t/s)**	**Auction (t/s)**	**Person (t/s)**	**Auction Duration (s)**
Controlled Low Rate	5000	500	100	20	8
Mid-Rate Streaming	25000	500	100	20	6
High-Rate Streaming	100000	3000	600	300	5
Peak Load Simulation	100000	6000	1200	600	3

Table A.1: Data rate configurations for each query scenario.

Queries involving windowed operators such as `AGGREGATE` and `JOIN` were further evaluated using time-based and element-based windowing strategies. These configurations determine the size and frequency of window activation and significantly influence processing latency.

Scenario	**Window Type**	**Window Size (ms)**	**Advance (ms)**
Controlled Low Rate	Tumbling	2000	2000
Mid-Rate Streaming	Tumbling	2000	2000
High-Rate Streaming	Sliding	2000	1000
Peak Load Simulation	Sliding	2000	500

Table A.2: Time-based windowing strategies across different stream rate scenarios.

Throughout the query listings, placeholders such as `STREAM_SIZE`, `WINDOW_SIZE_MS`, and `ADVANCE_SIZE_MS` should be substituted with the respective values from Tables A.1, A.2, and A.3 based on the scenario being tested. This structured

Scenario	Window Type	Window Size (elements)	Advance (elements)
Controlled Low Rate	Tumbling	1000	1000
Mid-Rate Streaming	Tumbling	2000	2000
High-Rate Streaming	Sliding	3000	1500
Peak Load Simulation	Sliding	3000	1500

Table A.3: Element-based windowing strategies used in stream query execution.

parametrization ensures consistent evaluation across different workloads and windowing strategies.

The remainder of this chapter presents the individual query definitions, grouped by operator type and transformation complexity.

A.1 Baseline Queries

Baseline queries are designed to capture the inherent system latency associated with data ingestion, parsing, and output in the absence of any transformation or filtering logic. These queries serve as reference points for evaluating the overhead introduced by more complex query operations.

Each baseline query uses the `CLOSESTREAM` operator to retrieve a fixed number of tuples from a specific stream. The placeholder `STREAM_SIZE` needs to be replaced by the tuple counts specified in Table A.1, corresponding to the different data rate scenarios under evaluation.

A.1.1 Query 1: Auction Stream Latency

This query captures the baseline latency of the `auction` stream by ingesting and outputting a fixed number of tuples without applying any transformation or filtering.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
output = CLOSESTREAM({
           count = STREAM_SIZE
           }, nexmark:auction)
```

Query A.1: Baseline latency of the `auction` stream

A.1.2 Query 2: Bid Stream Latency

This query captures the baseline latency of the `bid` stream by ingesting and outputting a fixed number of tuples without applying any transformation or filtering.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
output = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)
```

Query A.2: Baseline latency of the `bid` stream

A.1.3 Query 3: Person Stream Latency

This query captures the baseline latency of the `person` stream by ingesting and outputting a fixed number of tuples without applying any transformation or filtering.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
output = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:person)
```

Query A.3: Baseline latency of the `person` stream

A.2 Selection Queries

This section evaluates the impact of applying the `SELECT` operator with different predicate conditions. By varying the selectivity of the predicate (e.g., filtering more or fewer tuples), we analyze how selection affects the system's processing latency.

Each query begins with the `CLOSESTREAM` operator to retrieve a fixed number of tuples from the bid stream. The placeholder `STREAM_SIZE` needs to be replaced by the tuple counts specified in Table A.1, corresponding to the different data rate scenarios under evaluation.

A.2.1 Query 4: High Selectivity Predicate on Bid Stream

This query applies a selection predicate that filters a larger portion of the stream (`price > 300`), helping to observe latency under high selectivity conditions.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

output = SELECT({
            predicate = 'price > 300'
            }, bidStream)
```

Query A.4: Select bids with `price > 300` from the `bid` stream

A.2.2 Query 5: Low Selectivity Predicate on Bid Stream

This query applies a predicate that allows most tuples to pass through (`price > 1`), simulating low selectivity to evaluate its effect on latency.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY

bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

output = SELECT({
            predicate = 'price > 1'
            }, bidStream)
```

Query A.5: Select bids with `price > 1` from the `bid` stream

A.2.3 Query 6: High Selectivity on Auction Stream

This query applies a selection predicate to the `auction` stream, filtering auctions where the `initialbid` is greater than 150, to evaluate latency under high selectivity.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

output = SELECT({
            predicate = 'initialbid > 150'
```

```
            }, auctionStream)
```

Query A.6: Select auctions with `initialbid > 150` from the `auction` stream

A.2.4 Query 7: Low Selectivity on Auction Stream

This query applies a low selectivity predicate (`initialbid > 1`) to observe system behavior when most tuples pass through.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

output = SELECT({
            predicate = 'initialbid > 1'
            }, auctionStream)
```

Query A.7: Select auctions with `initialbid > 1` from the `auction` stream

A.3 Projection Queries

This section evaluates the effect of applying the `PROJECT` operator on query latency. By reducing the number of attributes emitted from the stream, these queries help assess how data reduction impacts system performance.

Each query begins with the `CLOSESTREAM` operator to retrieve a fixed number of tuples. The placeholder `STREAM_SIZE` needs to be replaced by the tuple counts specified in Table A.1, corresponding to the different data rate scenarios under evaluation.

A.3.1 Query 8: Projection on Bid Stream

This query projects only the `price` attribute from the `bid` stream to measure the latency effect of emitting fewer attributes.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY

bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

output = PROJECT({
            attributes = ['price']
```

```
            }, bidStream)
```

Query A.8: Project `price` from the `bid` stream

A.3.2 Query 9: Projection on Auction Stream

This query projects a subset of attributes `id`, `itemname`, and `initialbid`, from the `auction` stream to study partial projection effects on latency.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

output = PROJECT({
            attributes = ['id', 'itemname', 'initialbid']
            }, auctionStream)
```

Query A.9: Project `id`, `itemname`, and `initialbid` from the `auction` stream

A.4 Projection along with selection Queries

This section evaluates the performance impact of combining `SELECT` and `PROJECT` operations. These queries simulate realistic workloads where filtering and attribute reduction are applied together. The queries are also useful for analyzing query plan optimization in Odysseus, which may internally rewrite operations for efficiency.

Each query begins with the `CLOSESTREAM` operator to retrieve a fixed number of tuples. The placeholder `STREAM_SIZE` needs to be replaced by the tuple counts specified in Table A.1, corresponding to the different data rate scenarios under evaluation.

A.4.1 Query 10: Selection then Projection on Auction Stream (High Selectivity)

This query first filters auctions with `initialbid > 150`, then projects selected attributes to evaluate latency under high selectivity.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

filteredAuctions = SELECT({
```

```
            predicate = 'initialbid > 150'
            }, auctionStream)

output = PROJECT({
            attributes = ['id', 'itemname', 'initialbid']
            }, filteredAuctions)
```

Query A.10: Select and project auctions with `initialbid > 150`

A.4.2 Query 11: Selection then Projection on Auction Stream (Low Selectivity)

This query filters auctions with `initialbid > 1`, then projects the same attributes to examine latency under low selectivity.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

filteredAuctions = SELECT({
            predicate = 'initialbid > 1'
            }, auctionStream)

output = PROJECT({
            attributes = ['id', 'itemname', 'initialbid']
            }, filteredAuctions)
```

Query A.11: Select and project auctions with `initialbid > 1`

A.4.3 Query 12: Selection then Projection on Bid Stream (High Selectivity)

This query applies a high-selectivity filter `price > 300` on the `bid` stream, followed by projection.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

filteredBids = SELECT({
            predicate = 'price > 300'
            }, bidStream)
```

```
output = PROJECT({
            attributes = ['price']
            }, filteredBids)
```

Query A.12: Select and project bids with `price > 300`

A.4.4 Query 13: Selection then Projection on Bid Stream (Low Selectivity)

This query filters bids with `price > 1`, then projects the `price` attribute.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

filteredBids = SELECT({
            predicate = 'price > 1'
            }, bidStream)

output = PROJECT({
            attributes = ['price']
            }, filteredBids)
```

Query A.13: Select and project bids with `price > 1`

A.4.5 Query 14: Projection First, Then Selection on Auction Stream

This query first projects attributes from the `auction` stream, then applies a filter.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

projectedAuctions = PROJECT({
            attributes = ['id', 'itemname', 'initialbid']
            }, auctionStream)

output = SELECT({
            predicate = 'initialbid > 150'
            }, projectedAuctions)
```

Query A.14: Project then select auctions with `initialbid > 150`

A.4.6 Query 15: Projection First, Then Selection on Bid Stream

Similar to Query 14, this query begins with projection followed by selection.

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

projectedBids = PROJECT({
            attributes = ['auction', 'price']
            }, bidStream)

output = SELECT({
            predicate = 'price > 300'
            }, projectedBids)
```

Query A.15: Project then select bids with `price > 300`

A.5 Aggregation Queries

This section evaluates the performance of aggregation operations over both the `auction` and `bid` streams. Queries apply various aggregate functions, such as `max`, `min`, `sum`, `avg`, and `count`, with and without grouping. Aggregations are performed over time-based and element-based windows.

The `CLOSESTREAM` operator is used to ingest a fixed number of tuples, denoted by `STREAM_SIZE`, which should be set according to the data rate configurations in Table A.1.

Window sizes and advances are determined by the type of windowing strategy:

- For time-based windowing, refer to Table A.2.
- For element-based windowing, refer to Table A.3.

A.5.1 Query 16: Max Aggregation on Auction Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='MaxAggregation',
            aggregations=[
                ['max', 'initialbid', 'max_bid', 'Double']
            ]
            }, windowed_input)
```

Query A.16: Compute max `initialbid` in auction stream

A.5.2 Query 17: Max Aggregation on Bid Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='MaxBidAggregation',
            aggregations=[
                ['max', 'price', 'max_price', 'Double']
            ]
            }, windowed_input)
```

Query A.17: Compute max `price` in bid stream

A.5.3 Query 18: Min Aggregation on Auction Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='MinAggregation',
            aggregations=[
                ['min', 'initialbid', 'min_bid', 'Double']
            ]
            }, windowed_input)
```

Query A.18: Compute min `initialbid` in auction stream

A.5.4 Query 19: Min Aggregation on Bid Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='MinBidAggregation',
            aggregations=[
                ['min', 'price', 'min_price', 'Double']
            ]
            }, windowed_input)
```

Query A.19: Compute min `price` in bid stream

A.5.5 Query 20: Sum Aggregation on Auction Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='SumAggregation',
            aggregations=[
                ['sum', 'initialbid', 'sum_bid', 'Double']
            ]
            }, windowed_input)
```

Query A.20: Sum of `initialbid` in auction stream

A.5.6 Query 21: Sum Aggregation on Bid Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='SumBidAggregation',
            aggregations=[
                ['sum', 'price', 'sum_price', 'Double']
            ]
            }, windowed_input)
```

Query A.21: Sum of `price` in bid stream

A.5.7 Query 22: Count Aggregation on Auction Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='CountAggregation',
            aggregations=[
                ['count', '*', 'bid_count', 'Integer']
            ]
            }, windowed_input)
```

Query A.22: Count all tuples in auction stream

A.5.8 Query 23: Count Aggregation on Bid Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='CountBidAggregation',
            aggregations=[
                ['count', '*', 'bid_count', 'Integer']
            ]
            }, windowed_input)
```

Query A.23: Count all tuples in bid stream

A.5.9 Query 24: Average Aggregation on Auction Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='AvgAggregation',
            aggregations=[
                ['avg', 'initialbid', 'avg_bid', 'Double']
            ]
            }, windowed_input)
```

Query A.24: Average `initialbid` in auction stream

A.5.10 Query 25: Average Aggregation on Bid Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='AvgBidAggregation',
            aggregations=[
                ['avg', 'price', 'avg_price', 'Double']
            ]
            }, windowed_input)
```

Query A.25: Average `price` in bid stream

A.5.11 Query 26: GroupBy Aggregation on Auction Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='CategoryWiseAggregation',
            group_by=['category'],
            aggregations=[
                ['sum', 'initialbid', 'total_bid', 'Double'],
                ['count', '*', 'bid_count', 'Integer']
            ]
            }, windowed_input)
```

Query A.26: Category-wise aggregation on auction stream

A.5.12 Query 27: GroupBy Aggregation on Bid Stream

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
input = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

windowed_input = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, input)

output = AGGREGATE({
            name='AuctionWiseBidAggregation',
            group_by=['auction'],
            aggregations=[
                ['sum', 'price', 'total_bid', 'Double'],
                ['count', '*', 'bid_count', 'Integer']
            ]
```

```
            }, windowed_input)
```

Query A.27: Auction-wise aggregation on bid stream

A.6 Joining Queries

This section investigates latency trends for stream joins using the `JOIN` operator. It includes two-stream and three-stream joins, with and without projections. Queries operate over time-based or element-based windowed streams, simulating realistic stream integration scenarios.

The `CLOSESTREAM` operator is used to ingest a fixed number of tuples defined by `STREAM_SIZE`, which corresponds to the stream rate scenarios in Table A.1.

Window configurations are scenario-dependent:

- For time-based windows, refer to Table A.2.
- For element-based windows, refer to Table A.3.

A.6.1 Query 28: Two-Stream Join with Aggregation (Time-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

bidWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, bidStream)

auctionWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, auctionStream)

joinedStream = JOIN({
            predicate = 'auction = id',
            card = 'MANY_ONE'
            }, bidWindow, auctionWindow)
```

```
output = AGGREGATE({
            name='AuctionBidAggregation',
            group_by=['auction'],
            aggregations=[
                ['sum', 'price', 'total_bid', 'Double'],
                ['count', 'id', 'bid_count', 'Integer']
            ]
            }, joinedStream)
```

Query A.28: Join `bid` and `auction` streams and aggregate over time-based windows

A.6.2 Query 29: Two-Stream Join with Aggregation (Element-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

bidWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, bidStream)

auctionWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, auctionStream)

joinedStream = JOIN({
            predicate = 'auction = id',
            card = 'MANY_ONE'
            }, bidWindow, auctionWindow)

output = AGGREGATE({
            name='AuctionBidAggregation',
            group_by=['auction'],
            aggregations=[
                ['sum', 'price', 'total_bid', 'Double'],
                ['count', 'id', 'bid_count', 'Integer']
```

```
            ]
            }, joinedStream)
```

Query A.29: Join `bid` and `auction` streams and aggregate over element-based windows

A.6.3 Query 30: Two-Stream Join with Projection (Time-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

bidWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, bidStream)

auctionWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, auctionStream)

bidProj = PROJECT({
            attributes = ['auction', 'price']
            }, bidWindow)

auctionProj = PROJECT({
            attributes = ['id', 'itemname']
            }, auctionWindow)

output = JOIN({
            predicate = 'auction = id',
            card = 'MANY_ONE'
            }, bidProj, auctionProj)
```

Query A.30: Project before joining `bid` and `auction` streams (time-based)

A.6.4 Query 31: Two-Stream Join with Projection (Element-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

bidWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, bidStream)

auctionWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, auctionStream)

bidProj = PROJECT({
            attributes = ['auction', 'price']
            }, bidWindow)

auctionProj = PROJECT({
            attributes = ['id', 'itemname']
            }, auctionWindow)

output = JOIN({
            predicate = 'auction = id',
            card = 'MANY_ONE'
            }, bidProj, auctionProj)
```

Query A.31: Project before joining `bid` and `auction` streams (element-based)

A.6.5 Query 32: Two-Stream Join Without Projection (Time-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

bidWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, bidStream)

auctionWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, auctionStream)

output = JOIN({
            predicate = 'auction = id',
            card = 'MANY_ONE'
            }, bidWindow, auctionWindow)
```

Query A.32: Join `bid` and `auction` streams without projection (time-based)

A.6.6 Query 33: Two-Stream Join Without Projection (Element-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

bidWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, bidStream)

auctionWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, auctionStream)

output = JOIN({
            predicate = 'auction = id',
            card = 'MANY_ONE'
            }, bidWindow, auctionWindow)
```

Query A.33: Join `bid` and `auction` streams without projection (element-based)

A.6.7 Query 34: Three-Stream Join with Projection (Time-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

personStream = CLOSESTREAM({
            count = STREAM_SIZE
```

```
            }, nexmark:person)

bidWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, bidStream)

auctionWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, auctionStream)

personWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, personStream)

bidProj = PROJECT({
            attributes = ['auction', 'price', 'bidder']
            }, bidWindow)

personProj = PROJECT({
            attributes = ['id', 'name', 'state']
            }, personWindow)

auctionRenamed = MAP({
            expressions = [
                ['id', 'auction_id'],
                'itemname',
                'seller'
            ]
            }, auctionWindow)

bidAuctionJoin = JOIN({
            predicate = 'auction = auction_id',
            card = 'MANY_ONE'
            }, bidProj, auctionRenamed)

output = JOIN({
            predicate = 'bidder = id',
            card = 'MANY_ONE'
            }, bidAuctionJoin, personProj)
```

Query A.34: Join `bid`, `auction`, and `person` streams with projection (time-based)

A.6.8 Query 35: Three-Stream Join with Projection (Element-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

personStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:person)

bidWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, bidStream)

auctionWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, auctionStream)

personWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, personStream)

bidProj = PROJECT({
            attributes = ['auction', 'price', 'bidder']
            }, bidWindow)

personProj = PROJECT({
            attributes = ['id', 'name', 'state']
            }, personWindow)

auctionRenamed = MAP({
            expressions = [
                ['id', 'auction_id'],
                'itemname',
                'seller'
            ]
            }, auctionWindow)
```

```
bidAuctionJoin = JOIN({
            predicate = 'auction = auction_id',
            card = 'MANY_ONE'
            }, bidProj, auctionRenamed)

output = JOIN({
            predicate = 'bidder = id',
            card = 'MANY_ONE'
            }, bidAuctionJoin, personProj)
```

Query A.35: Join `bid`, `auction`, and `person` streams with projection (element-based)

A.6.9 Query 36: Three-Stream Join without Projection (Time-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

personStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:person)

bidWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, bidStream)

auctionWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, auctionStream)

personWindow = TIMEWINDOW({
            size = [WINDOW_SIZE_MS, 'MILLISECONDS'],
```

```
            advance = [ADVANCE_SIZE_MS, 'MILLISECONDS']
            }, personStream)

auctionRenamed = MAP({
            expressions = [
                ['id', 'auction_id'],
                'itemname',
                'description',
                'initialbid',
                'reserve',
                'expires',
                'seller',
                'category'
            ]
            }, auctionWindow)

bidAuctionJoin = JOIN({
            predicate = 'auction = auction_id',
            card = 'MANY_ONE'
            }, bidWindow, auctionRenamed)

output = JOIN({
            predicate = 'bidder = id',
            card = 'MANY_ONE'
            }, bidAuctionJoin, personWindow)
```

Query A.36: Join all streams without projection (time-based)

A.6.10 Query 37: Three-Stream Join without Projection (Element-Based)

```
#PARSER PQL
#METADATA TimeInterval
#ADDQUERY
bidStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:bid)

auctionStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:auction)

personStream = CLOSESTREAM({
            count = STREAM_SIZE
            }, nexmark:person)

bidWindow = ELEMENTWINDOW({
```

```
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, bidStream)

auctionWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, auctionStream)

personWindow = ELEMENTWINDOW({
            size = WINDOW_SIZE,
            advance = ADVANCE_SIZE
            }, personStream)

auctionRenamed = MAP({
            expressions = [
                ['id', 'auction_id'],
                'itemname',
                'description',
                'initialbid',
                'reserve',
                'expires',
                'seller',
                'category'
            ]
            }, auctionWindow)

bidAuctionJoin = JOIN({
            predicate = 'auction = auction_id',
            card = 'MANY_ONE'
            }, bidWindow, auctionRenamed)

output = JOIN({
            predicate = 'bidder = id',
            card = 'MANY_ONE'
            }, bidAuctionJoin, personWindow)
```

Query A.37: Join all streams without projection (element-based)