Streaming SQL with PipelineDB
What is PipelineDB?
What is PipelineDB?

- Continuous SQL on streams (continuous views)
What is **PipelineDB**?

- Continuous SQL on streams (continuous views)
- High-throughput, incremental materialized views
What is **PipelineDB**?

- Continuous SQL on streams (continuous views)
- High-throughput, incremental materialized views
- Based on PostgreSQL 9.5
What is **PipelineDB**?

- Continuous SQL on streams (continuous views)
- High-throughput, incremental materialized views
- Based on PostgreSQL 9.5
- No special client libraries
What is **PipelineDB**?

- Continuous SQL on streams (continuous views)
- High-throughput, incremental materialized views
- Based on PostgreSQL 9.5
- No special client libraries
- Free and open-source (GPLv3)
What is **PipelineDB**?

- Continuous SQL on streams (continuous views)
- High-throughput, incremental materialized views
- Based on PostgreSQL 9.5
- No special client libraries
- Free and open-source (GPLv3)

*(30-second demo)*
When is PipelineDB not useful?
When is PipelineDB not useful?

- SQL isn’t a fit
- Ad-hoc on granular data
When is PipelineDB useful?
When is PipelineDB **useful**?

- High throughput aggregations *(realtime reporting/analytics)*
- Computations over sliding windows *(continuous monitoring/ops)*
- Queries are known in advance
100,000 feet
Produce SQL

Process

Consume
100,000 feet

Process

SQL

Produce

Consume

Aggregation
Filtering
Sliding windows
100,000 feet

Process

SQL

Aggregation
Filtering
Sliding windows

= Reduction
Why did we build it?
Produce

Consume

Pipeline

DB

continuous view
Simplicity is nice, but what else?
Benefits of continuous SQL on streams

- Aggregate before writing to disk
Benefits of continuous SQL on streams

- Aggregate before writing to disk

CREATE CONTINUOUS VIEW v AS
SELECT COUNT(*) FROM stream
Benefits of continuous SQL on streams

- Aggregate before writing to disk

```sql
CREATE CONTINUOUS VIEW v AS
SELECT COUNT(*) FROM stream
```
Benefits of continuous SQL on streams

- Sliding window queries

```sql
CREATE CONTINUOUS VIEW v WITH (max_age = '1 hour') AS
SELECT COUNT(*) FROM stream
```

- Any information outside of the window is excluded from results and deleted from disk

- Essentially automatic TTL
Benefits of continuous SQL on streams

- Probabilistic computations on infinite inputs

```
CREATE CONTINUOUS VIEW v AS
  SELECT COUNT(DISTINCT x) FROM never_ending_stream
```

- Streaming **Top-K, Percentiles, distincts**, large set **cardinalities**
- Constant space
- No sorting
- Small margin of error
Streams

- Internally, a stream is **Foreign Table**

```sql
CREATE STREAM stream (x int, y int, z int);
INSERT INTO stream (x, y, z) VALUES (0, 1, 2);
```
Streams

- Internally, a stream is **Foreign Table**

  ```
  CREATE STREAM stream (x int, y int, z int);
  INSERT INTO stream (x, y, z) VALUES (0, 1, 2);
  ```

- System-wide **Foreign Server** called **pipeline Streams**

- **stream_fdw** reads from/writes to the **Stream Buffer**

- No permanent storage

- Stream rows only exist until they’ve been fully read
stream buffer → query on microbatch → incremental table update
stream buffer ➔ query on microbatch ➔ incremental table update

- INSERT INTO ...

<table>
<thead>
<tr>
<th>HeapTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeapTuple</td>
</tr>
<tr>
<td>HeapTuple</td>
</tr>
<tr>
<td>HeapTuple</td>
</tr>
<tr>
<td>HeapTuple</td>
</tr>
</tbody>
</table>
- **INSERT INTO** ...
- **Concurrent circular buffer**

```
<table>
<thead>
<tr>
<th>HeapTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeapTuple</td>
</tr>
<tr>
<td>HeapTuple</td>
</tr>
<tr>
<td>HeapTuple</td>
</tr>
</tbody>
</table>
```

- **stream buffer** → **query on microbatch** → **incremental table update**
- INSERT INTO ...
- Concurrent circular buffer
- Preallocated block of shared memory
stream buffer → query on microbatch → incremental table update

- INSERT INTO ...
- Concurrent circular buffer
- Preallocated block of shared memory

HeapTuple

HeapTuple

HeapTuple

HeapTuple

HeapTuple

HeapTuple

{0,1,0,1,1}
- INSERT INTO ...
- Concurrent circular buffer
- Preallocated block of shared memory

Stream buffer \rightarrow \text{query on microbatch} \rightarrow \text{incremental table update}

- HeapTuple

- HeapTuple \{0,1,0,1,1\}

- HeapTuple \{1,1,1,1,1\}
stream buffer \rightarrow \text{query on microbatch} \rightarrow \text{incremental table update}

- **INSERT INTO ...**
- **Concurrent circular buffer**
- **Preallocated block of shared memory**

```
<table>
<thead>
<tr>
<th>HeapTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeapTuple</td>
</tr>
<tr>
<td>HeapTuple</td>
</tr>
<tr>
<td>HeapTuple</td>
</tr>
<tr>
<td>HeapTuple</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>HeapTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0,1,0,1,1}</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>HeapTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,1,1,1,1}</td>
</tr>
</tbody>
</table>
```
/* At Postmaster startup time ... */
worker.bgw_main = any_function;
worker.bgw_main_arg = (Datum) arg;

RegisterDynamicBackgroundWorker(&worker, &handle);
Stream buffer → **query on microbatch** → incremental table update

```
<table>
<thead>
<tr>
<th>HeapTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeapTuple</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>HeapTuple</td>
</tr>
</tbody>
</table>
```

```
SELECT count(*), avg(x) FROM stream
```
SELECT count(*), avg(x) FROM stream

<table>
<thead>
<tr>
<th>microbatch_result</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
</tr>
<tr>
<td>1000</td>
</tr>
</tbody>
</table>

stream_fdw#GetStreamScanPlan

while (!BatchDone(node))
{
  tup = PinNext(buf)
  yield MarkAsRead(tup)
}
Worker process parallelism

tuples round-robin'd across n worker procs

Stream buffer

Worker proc 0
Worker proc 1
Worker proc ...
Worker proc n
transition_state = combine(microbatch_tstate, existing_tstate)
- transition_state = combine(microbatch_tstate, existing_tstate)

- pipeline_combine catalog table maps combine functions to aggregates
transition_state = combine(microbatch_tstate, existing_tstate)

pipeline_combine catalog table maps combine functions to aggregates

No changes to pg_aggregate catalog table or existing aggregate functions
stream buffer → query on microbatch → incremental table update

- transition_state = combine(microbatch_tstate, existing_tstate)
- `pipeline_combine` catalog table maps combine functions to aggregates
- No changes to `pg_aggregate` catalog table or existing aggregate functions
- User-defined aggregates just need a `combinefunc` to be combinable

```sql
CREATE AGGREGATE combinable_agg(x)
(
    sfunc=sfunc,
    finalfunc=finalfunc,
    combinefunc=combinefunc,
);
```
transition_state = \texttt{combine}(\texttt{microbatch\_tstate}, \texttt{existing\_tstate})

\textbf{pipeline\_combine} catalog table maps combine functions to aggregates

<table>
<thead>
<tr>
<th>microbatch_result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{count}</td>
</tr>
<tr>
<td>\textbf{1000}</td>
</tr>
</tbody>
</table>
stream buffer → query on microbatch → incremental table update

- transition_state = combine(microbatch_tstate, existing_tstate)
- pipeline_combine catalog table maps combine functions to aggregates

<table>
<thead>
<tr>
<th>microbatch_result</th>
<th>combine()</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>avg</td>
</tr>
<tr>
<td>1000</td>
<td>{1000, 4000}</td>
</tr>
<tr>
<td>5000</td>
<td></td>
</tr>
</tbody>
</table>
• transition_state = `combine(microbatch_tstate, existing_tstate)`

• `pipeline_combine` catalog table maps combine functions to aggregates

```
<table>
<thead>
<tr>
<th>microbatch_result</th>
<th>combine()</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>avg</td>
</tr>
<tr>
<td>1000</td>
<td><code>{1000, 4000}</code></td>
</tr>
<tr>
<td>5000</td>
<td><code>{5000, 10000}</code></td>
</tr>
</tbody>
</table>
```

existing on-disk row
- transition_state = combine(microbatch_tstate, existing_tstate)

- pipeline_combine catalog table maps combine functions to aggregates

<table>
<thead>
<tr>
<th>microbatch_result</th>
<th>combine()</th>
<th>updated_result</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>avg</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>{1000, 4000}</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>{5000, 10000}</td>
<td></td>
</tr>
</tbody>
</table>
lookup_plan = get_plan(SELECT * FROM matrel WHERE hash_group(x, y, z) IN (...))
lookup_plan = get_plan(SELECT * FROM matrel WHERE hash_group(x, y, z) IN (...))

/* dynamically generate a VALUES node */
foreach(row, microbatch)
    values = lappend(values, hash_group(row));
lookup_plan = get_plan(SELECT * FROM matrel WHERE hash_group(x, y, z) IN (...))

/* dynamically generate a VALUES node */
foreach(row, microbatch)
  values = lappend(values, hash_group(row));

set_values(lookup_plan, values)
lookup_plan = get_plan(SELECT * FROM matrel WHERE hash_group(x, y, z) IN (...))

/* dynamically generate a VALUES node */
foreach(row, microbatch)
  values = lappend(values, hash_group(row));

set_values(lookup_plan, values)

existing = PortalRun(lookup_plan, ...)

/* now we’re reading to combine these on-disk tuples with the incoming batch result */
SELECT * FROM matrel WHERE hash_group(x, y, z) IN (hash(microbatch group), ...)

- This query needs to be as fast as possible
- Continuous views indexed on a 32-bit hash of grouping
- **Pro**: maximize cardinality of the index keyspace, great for random perf
- **Con**: must deal with collisions programmatically
stream buffer → query on microbatch → incremental table update

SELECT * FROM matrel WHERE hash_group(x, y, z) IN (hash(microbatch group), ...)

- If the grouping contains a **time-based column**, we can do better

  CREATE ... AS SELECT day(timestamp), count(*) FROM stream GROUP BY day
stream buffer \rightarrow \text{query on microbatch} \rightarrow \text{incremental table update}

\[
\text{SELECT * FROM matrel WHERE } \text{hash\_group}(x, y, z) \text{ IN (hash(microbatch group), ...)}
\]

- If the grouping contains a \textit{time-based column}, we can do better

\[
\text{CREATE ... AS SELECT day(timestamp), count(*) FROM stream GROUP BY day}
\]

- These continuous views are indexed with \textbf{64 bits}: hash of grouping

| Timestamp from group (32 bits) | Regular 32-bit grouping hash |
SELECT * FROM matrel WHERE hash_group(x, y, z) IN (hash(microbatch group), ...)

- If the grouping contains a time-based column, we can do better

CREATE ... AS SELECT day(timestamp), count(*) FROM stream GROUP BY day

- These continuous views are indexed with 64 bits: hash of grouping

<table>
<thead>
<tr>
<th>Timestamp from group (32 bits)</th>
<th>Regular 32-bit grouping hash</th>
</tr>
</thead>
</table>

- **Pro**: most incoming groups will have a similar timestamp, so better index caching

- **Con**: larger index footprint
stream buffer $\rightarrow$ query on microbatch $\rightarrow$ incremental table update

✔ microbatch result generated from stream by worker
stream buffer ➞ query on microbatch ➞ incremental table update

- ✔ microbatch result generated from stream by worker
- ✔ existing result retrieved from disk
combine_plan = get_plan(SELECT group, combine(count), combine(avg) 
   FROM microbatch_result UNION existing GROUP BY group);

combined = PortalRun(combine_plan, ...)

foreach(row, combined) {
   if (new_tuple(row))
      heap_insert(row, ...);
   else
      heap_update(row, ...);
}
Combiner process parallelism

On-disk groupings are sharded over combiners by group

Each row is guaranteed to only ever be updated by one combiner process

Continuous view

- grouping (a, b, c)
- grouping (d, e, f)
- grouping (g, h, i)
- grouping (j, k, l)
Just released! Continuous transforms
Just released! **Continuous transforms**

- Worker-only continuous queries
Just released! **Continuous transforms**

- Worker-only continuous queries
- Arbitrary procedure called on its output rows
Just released! **Continuous transforms**

- Worker-only continuous queries
- Arbitrary procedure called on its output rows
- Enable work sharing between continuous views
Just released! Continuous transforms

- Worker-only continuous queries
- Arbitrary procedure called on its output rows
- Enable work sharing between continuous views

```sql
CREATE CONTINUOUS TRANSFORM xform AS
  SELECT foo(col), bar(col) FROM raw_stream
  THEN EXECUTE PROCEDURE pipeline_stream_insert('normalized_stream')
```
Just released! **Continuous transforms**

- Worker-only continuous queries
- Arbitrary procedure called on its output rows
- Enable work sharing between continuous views

```sql
CREATE CONTINUOUS TRANSFORM xform AS
  SELECT foo(col), bar(col) FROM raw_stream
  THEN EXECUTE PROCEDURE pipeline_stream_insert('normalized_stream')
```

```sql
CREATE CONTINUOUS VIEW v0 AS SELECT ... FROM normalized_stream;
CREATE CONTINUOUS VIEW v1 AS SELECT ... FROM normalized_stream;
```
Thanks!

- derek@pipelinedb.com
- pipelinedb.com
- docs.pipelinedb.com
- github.com/pipelinedb