NUMA, Streams & touch-based exploration

Erietta Liarou

CS165 - 04/09/2014
Part I

Adaptive Transaction processing on hardware Islands

Danica Porobic, Erietta Liarou, Pinar Tözün, Anastasia Ailamaki
Scaling up OLTP on multisockets

Throughput

# of sockets

Wednesday, April 9, 14
Scaling up OLTP on multisockets

# of sockets

Throughput

1 2 4 8

# of sockets
Scaling up OLTP on multisockets

Multisocket servers are severely underutilized
Communication latencies vary by an order of magnitude
Communication latencies vary by an order of magnitude
Hardware parallelism: a fact of life

ILP
pipelining
multithreading

multisocket multicores
(CMP)

heterogeneous CMP
interconnect topologies can become very complicated ...

2 sockets

4 sockets - a

4 sockets - b

8 sockets
OLTP on Hardware Islands

Shared-everything  Island shared-nothing  Shared-nothing
Scaling-up on a 8-socket machine

Throughput (MTPS) vs Number of sockets

- **Shared-nothing**
- **Island shared-nothing**
- **Shared-everything**

8 socket x 10 core
800K row dataset
Probing one row

- Islands significantly challenge scalability
Physical partitioning for Islands

No configuration is optimal for all environments

Throughput (KTps)

% multisite transactions

- Shared-nothing
- Island shared-nothing
- Shared-everything

4 socket x 6 core
240K row dataset
Updating 10 rows
OLTP on Hardware Islands

Shared-everything
- stable
- no optimal

Island shared-nothing
- Robust middle ground

Shared-nothing
- Fast
- Sensitive to workload

challenges

* optimal configuration depends on workload and hardware

* expensive repartitioning due to physical data movement
ATraPos: Adaptive Transaction Processing

- No unnecessary inter-socket synchronization
- Workload- & hardware-aware partitioning
- Lightweight monitoring & repartitioning

Hardware and workload-aware shared-everything adaptive system
Critical path of transaction execution

Many accesses to shared data structures
PLP: Physiological partitioning

System state is still shared
Perfectly partitionable workload

Throughput (MTPS) vs. Number of sockets

- Shared-nothing
- PLP
- Centralized shared-everything

8 socket x 10 core
800K row dataset
Probing one row

Inter-socket accesses to system state are a bottleneck
ATraPos: Island-aware SE
Perfectly partitionable workload

Throughput (MTPS)

- Shared-nothing
- ATraPos
- PLP
- Centralized shared-everything

Island awareness brings scalability

8 socket x 10 core
800K row dataset
Probing one row
Naive partitioning and placement

8 socket x 10 core
800K rows per table
Probing 1 row each from A and B

Throughput (KTPS)

Cores are overloaded with contending threads
ATraPos partitioning and placement

8 socket x 10 core
800K rows per table
Probing 1 row each from A and B

Throughput (KTPS)

<table>
<thead>
<tr>
<th></th>
<th>PLP</th>
<th>ATraPos HW-aware</th>
<th>ATraPos Load balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
<td>600</td>
<td>2100</td>
</tr>
</tbody>
</table>

4.4x

Ignoring Islands brings synchronization overhead

Wednesday, April 9, 14
ATraPos partitioning and placement

Throughput (KTPS)

<table>
<thead>
<tr>
<th></th>
<th>PLP</th>
<th>ATraPos HW-aware</th>
<th>ATraPos Load balanced</th>
<th>ATraPos</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>900</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.8x

ATraPos: load balance and reduce synchronization

8 socket x 10 core
800K rows per table

Probing 1 row each from A and B

Probe A

Probe B
A TraPos monitoring and repartitioning

Goal:
* balance the load
* minimize synchronization
Repartitioning multi-rooted B-trees
Repartitioning multi-rooted B-trees

Splitting and merging B-trees access few pages
ATraPos repartitioning cost

- **merge**
- **split**
- **rearrange (split+merge)**

8 socket x 10 core
800K row table

Number of repartitioning actions

Repartitioning cost (ms)

Wednesday, April 9, 14
ATraPos repartitioning cost

Repartitioning takes < 200msec
Adapting to workload skew

**Graph Description:**
- **Y-axis:** Throughput (MTPS)
- **X-axis:** Time (s)
- **Lines:**
  - Red dashed line: Static
  - Black line: ATRAPOS
- **Annotations:**
  - Monitoring
  - Repartitioning
  - 50% requests to 20% data

**Note:**
- 8 socket x 10 core
- 800K subscribers
- TATP GetSubData

**Time Points:**
- 0 to 20 seconds: Monitoring phase
- 20 to 50 seconds: Repartitioning phase

**Additional Information:**
- Tuesday, April 9, 14
Adapting to workload skew

ATraPos detects skew and quickly adapts
Adapting to any workload type

Throughput (KTPS) vs Time (s)

- Static
- ATraPos

Monitoring
Repartitioning
GetNewDest
UpdSubData
Mix

8 socket x 10 core
800K subscribers
TATP

Wednesday, April 9, 14
Adapting to any workload type

ATraPos gracefully adapts to any changes

Throughput (KTPS)

Time (s)

Static

ATraPos

Monitoring

Repartitioning

GetNewDest

UpdSubData

Mix

8 socket x 10 core
800K subscribers
TATP

ATraPos gracefully adapts to any changes
Adaptive OLTP for Islands

challenges

* optimal configuration depends on workload and hardware
* expensive repartitioning due to physical data movement

ATraPos

* minimal intersocket access to the critical path
* Workload & hardware-aware partitioning & placement
* Lightweight monitoring and repartitioning
Part 2

Enhanced Stream Processing in a DBMS Kernel

Erietta Liarou, Stratos Idreos, Martin Kersten, Romulo Goncalves, Stefan Manegold

order of things in DBMSs
order of things in DBMSSs
order of things in DBMSs

1. Store incoming data
order of things in DBMSs

1. Store incoming data
2. Submit one-time query
order of things in DBMSs

1. Store incoming data
2. Submit one-time query
3. Query processing on the already stored data
order of things in DBMSs

1. Store incoming data
2. Submit one-time query
3. Query processing on the already stored data
4. Create answer
order of things in DSMSs
order of things in DSMSs
a data stream is a never ending sequence of tuples
a data stream is a never ending sequence of tuples
order of things in DSMSs

1. Submit continuous queries

a data stream is a never ending sequence of tuples
Submit continuous queries
Incoming streams

A data stream is a never ending sequence of tuples
A data stream is a never ending sequence of tuples. incoming streams. Input stream is processed on the fly. Submit continuous queries.
Submit continuous queries
Incoming streams
Input stream is processed on the fly
The produced results are continuously delivered to the clients

A data stream is a never ending sequence of tuples
One-time Queries vs. Continuous Queries

arrival time of query

\[ t_n \]

data time
One-time Queries vs. Continuous Queries

- Evaluated *once* over the already stored tuples
One-time Queries vs. Continuous Queries

- Evaluated *once* over the already stored tuples
One-time Queries vs. Continuous Queries

- **one-time query**
  - Evaluated *once* over the already stored tuples

- **continuous query**
  - Waits for *future* incoming tuples
  - Evaluated *continuously* as new tuples arrive
LSST

> than 30 TBytes of image data each night
a 150 Petabytes database

300GByte/s of data -> 300MByte/s
15 petabytes of data annually

THINGS THAT HAPPEN ON INTERNET EVERY 60 SECONDS

1 bone segment = 30G
1 day at the lab 1 Terabyte

THINGS THAT HAPPEN ON INTERNET EVERY 60 SECONDS
A new processing paradigm is born

streaming data + existing data

current DBMS technology is inefficient to handle the big data overflow
* Streaming functionalities on top of a modern DBMS kernel

* Exploit database techniques, query optimization and operators
the basic Idea

- **Trick** the database kernel to consider a continuous query as a normal one-time query
  - Schedule the trigger conditions
  - Wait to collect a few tuples and then evaluate the query

- Use **monetdb** columns to temporarily hold streaming data
  - Once a tuple is seen, it is *dropped*
The MonetDB stack

SQL Query

Query parser

Query Optimizer

MAL Interpreter

Query Executor

SQL

MAL
The MonetDB/DataCell stack

SQL Query

Query parser + CQ

Query Optimizer + DC opt

Continuous Query Scheduler

MAL Interpreter

Query Executor

SQL

Wednesday, April 9, 14
DataCell components

- **Receptor** ↔ Listens to a stream
- **Emitter** ↔ Delivers events to the clients
- **Factory** ↔ Continuous query
- **Basket** ↔ Holds events

Input Stream

... → R → Q → E → Output Stream
Baskets

- Hold a portion of the stream data
- Behave almost like relational tables
  - Once a tuple is seen, it is *dropped* from its basket.
- Batch processing
- Out of order processing
Factory

- Contains complete or partial *query plan*
- Its execution state is *saved* between the calls

Factory (INPUT, OUTPUT, BODY, DELETE)

CQ1:
INSERT INTO Y (y1)
SELECT x1
FROM X
WHERE v1 ≤ X.x1 ≤ v2

Factory_CQ1()

```plaintext
input = basket.bind(X,x1);
output = basket.bind(Y,y1);

WHILE true DO
  basket.lock(input);
  basket.lock(output);
  result = monetdb.select(input,v1,v2);
  basket.append(output,result);
  basket.empty(input);
  basket.unlock(input);
  basket.unlock(output);
  suspend();
END WHILE
```

Wednesday, April 9, 14
The Processing Model

The computational model is based on Petri-net
The Processing Model

The computational model is based on Petri-net

Diagram:

- Place
- Transition
- Token

Wednesday, April 9, 14
The computational model is based on Petri-net.
The Processing Model

The computational model is based on Petri-net

```
Select * from
(select * from X) as A,
(select * from
 (select * from Y) as B
 where B.a > 100
 ) as C
```

---

**Transition**

- **Factory**, **Emitter**, **Receptor**

**Place**

- **Baskets**

**Tokens**

- **Tuple**
The Processing Model

The computational model is based on *Petri-net*

```sql
SELECT *
FROM (SELECT *
      FROM X)
AS A,
(SELECT *
     FROM (SELECT *
           FROM Y)
    AS B
WHERE B.a > 100)
AS C
(SELECT *
 FROM Y)
AS B
(SELECT *
 FROM (SELECT *
       FROM Y)
  AS B
WHERE B.a > 100)
AS C
```

<table>
<thead>
<tr>
<th>transition</th>
<th>place</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory, Receptor, Emitter</td>
<td>Baskets</td>
<td>Tuple</td>
</tr>
</tbody>
</table>
Real-time/Incremental processing

- Window queries make possible the continuous query evaluation

- DBMSs do *not* handle sliding window queries
  - expired data
  - incremental processing
Real-time/Incremental processing

- Window queries make possible the continuous query evaluation

- DBMSs do not handle sliding window queries
  - expired data
  - incremental processing

Goal
Extend the DBMS to support sliding window queries

Solution
High level scheduling and dynamic query plan rewriting
Sliding window processing

```
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(6,2);
```
Sliding window processing

\[
\text{INSERT into output (y1)}
\]
\[
\text{SELECT avg(x1)}
\]
\[
\text{FROM input}
\]
\[
\text{WHERE x1>5 and window(6,2)};
\]
Sliding window processing

Re-evaluation: repeatedly access/process *partially the same* data

```
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(6,2);
```
Sliding window processing

- **Re-evaluation**: repeatedly access/process *partially the same* data

```sql
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(6,2);
```
Sliding window processing

- **Re-evaluation**: repeatedly access/process *partially the same* data
- **Incremental evaluation**: build the answer of a window snapshot by *exploiting partial intermediate results*

```sql
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(6,2);
```
Operator vs. Plan-level Incremental Processing

- *Basic window* strategy for sliding window processing
Operator vs. Plan-level Incremental Processing

- *Basic window* strategy for sliding window processing
Operator vs. Plan-level Incremental Processing

- *Basic window* strategy for sliding window processing
Operator vs. Plan-level Incremental Processing

- **Basic window** strategy for sliding window processing

![Diagram of window processing]

- expired data
- already processed data
- new data

Q
Operator vs. Plan-level Incremental Processing

- **Basic window** strategy for sliding window processing

- Typically the incremental logic is all the way down to the operators
  - specialized stream operators

- DataCell develops the incremental logic at the query plan level:
  - leaves the lower level intact
  - reuses the complete storage and execution engine of a DBMS kernel
  - inherits all the good properties of the DBMS regarding *scalability* and *robustness* in heavy workloads
Incremental evaluation of sliding windows

INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W,bw);

Original query plan
Incremental evaluation of sliding windows

INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W,bw);

Original query plan

[Diagram of the query plan with select and avg nodes and W1 window]

Wednesday, April 9, 14
Incremental evaluation of sliding windows

```
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W,bw);
```

**Original query plan**
Incremental evaluation of sliding windows

```
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W,bw);
```

Original query plan
Incremental evaluation of sliding windows

1) \textit{Split} the input stream into \( n \) basic windows

\begin{verbatim}
INSERT into output (y1) 
SELECT avg(x1) 
FROM input 
WHERE x1>5 and window(W,bw);
\end{verbatim}

Original query plan

![Diagram showing the original query plan with input W1 and select, avg, and results nodes.](image-url)
Incremental evaluation of sliding windows

1) *Split* the input stream into *n* basic windows
2) *Process* each basic window *separately*
3) *Merge* partial results

```
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W,bw);
```

**Original query plan**

**Incremental query plan**
Incremental evaluation of sliding windows

1) **Split** the input stream into \( n \) basic windows
2) **Process** each basic window *separately*
3) **Merge** partial results
4) **Slide** to prepare for next basic window

---

**Original query plan**

```
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W,bw);
```

---

**Incremental query plan**

```
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W,bw);
```
Incremental evaluation of sliding windows

1) **Split** the input stream into $n$ basic windows
2) **Process** each basic window *separately*
3) **Merge** partial results
4) **Slide** to prepare for next basic window

### Original query plan

```
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W, bw);
```

### Incremental query plan

**Transition phase**

- **Select** operation is performed for each basic window.
- Partial results are accumulated.
- Concatenation and sum operations are applied to intermediate results.
- Final results are obtained through division.
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1 > 5 and window(W, bw);

**Original query plan**

**Incremental query plan**

1) *Split* the input stream into *n* basic windows
2) *Process* each basic window *separately*
3) *Merge* partial results
4) *Slide* to prepare for next basic window

**transition phase**
Incremental evaluation of sliding windows

- **Split** the input stream into $n$ basic windows
- **Process** each basic window *separately*
- **Merge** partial results
- **Slide** to prepare for next basic window

Original query plan

Incremental query plan

INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1 > 5 and window(W, bw);

Transition phase
1) **Split** the input stream into \( n \) basic windows
2) **Process** each basic window *separately*
3) **Merge** partial results
4) **Slide** to prepare for next basic window

**Original query plan**

```sql
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W, bw);
```

**Incremental query plan**

**Insert** into output (y1)
```
SELECT avg(x1)
FROM input
WHERE x1>5 and window(W, bw);
```

**Process** each basic window *separately*.

**Merge** partial results.

**Slide** to prepare for next basic window.
Set of optimizer rules in DataCell
Continuous Query
transforms a function (one-time query) to a factory (continuous query)

CQ1:
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1 > 5 and window(W, bw);

window = W tuples,
slide = bw tuples

function user.q1(A0:=5):void{

  in:=sql.bind("sys","input","x1",0);
  a2 := algebra.select(in,A0,">");
  a3 := aggr.sum(a2);
  a4 := aggr.count(a2);
  a5 := calc./(a3,a4);
  sql.append("sys","output","y1",a5);
}
end q1;
Continuous Query

transforms a function (one-time query) to a factory (continuous query)

CQ1:
INSERT into output (y1)
SELECT avg(x1)
FROM input
WHERE x1 > 5 and window(W,bw);

window = W tuples,
slide = bw tuples

Factory (Input, Output, Body, Delete)

Factory_CQ1(A0:=5)

input = basket.bind(input, x1);
output = basket.bind(output, y1);

WHILE true DO
  basket.lock(input);
  basket.lock(output);
  a1 := algebra.slice(input, W);
  a2 := algebra.select(a1, A0, ">");
  a3 := aggr.sum(a2);
  a4 := aggr.count(a2);
  a5 := calc./(a3, a4);
  basket.append(output, a5);
  basket.empty(input, bw);
  basket.unlock(input);
  basket.unlock(output);
  suspend();
END WHILE

function user.q1(A0:=5):void{
  in:=sql.bind("sys","input","x1",0);
  a2 := algebra.select(in,A0,">");
  a3 := aggr.sum(a2);
  a4 := aggr.count(a2);
  a5 := calc./(a3,a4);
  sql.append("sys","output","y1",a5);
} end q1;
**Slicer**: cut the input stream into pieces

```plaintext
Factory_CQ1(A0:=5) {

  in = basket.bind(input, x1);
  out = basket.bind(output, y1);

  WHILE true DO
    basket.lock(input);
    basket.lock(output);

    a1 := algebra.slice(in, W);
    a2 := algebra.select(a1, A0, ">");
    a3 := aggr.sum(a2);
    a4 := aggr.count(a2);
    a5 := calc./(a3, a4);
    basket.append(output, a5);

    basket.empty(in, bw);
    basket.unlock(input);
    basket.unlock(output);
    suspend();

  END WHILE
}
```
Set of Optimizer Rules

**Slicer:** cut the input stream into pieces

```
Factory_CQ1(A0:=5){
  in = basket.bind(input,x1);
  out= basket.bind(output,y1);

  WHILE true DO
    basket.lock(input);
    basket.lock(output);
    a1 := algebra.slice(in, W);
    a2 := algebra.select(a1,A0,">");
    a3 := aggr.sum(a2);
    a4 := aggr.count(a2);
    a5 := calc./(a3,a4);
    basket.append(output,a5);
    basket.empty(in,bw);
    basket.unlock(input);
    basket.unlock(output);
  suspend();
  END WHILE
}
```
Slicer: cut the input stream into pieces

Factory_CQ1(A0:=5) {
  in  := basket.bind(input,x1);
  out := basket.bind(output,y1);
  WHILE true DO
    basket.lock(input);
    basket.lock(output);
    a1 := algebra.slice(in,W);
    a2 := algebra.select(a1,A0,">");
    a3 := aggr.sum(a2);
    a4 := aggr.count(a2);
    a5 := calc./(a3,a4);
    basket.append(output,a5);
    basket.empty(in,bw);
  END WHILE
}

CQ1:
INSERT into output (y1)
SELECT avg(x1) FROM input
WHERE x1>5 and window(W,bw);

Factory_CQ1(A0:=5) {
  ...
  in1 := algebra.slice(a1,bw1);
  in2 := algebra.slice(a1,bw2);
  in_i := algebra.slice(a1,...);
  in_n := algebra.slice(a1,bw_n);
  a1:=mat.new(in1,in2,in3,in4);
  ...
  END WHILE
}
Set of Optimizer Rules

Mitosis + merge

\[
\begin{align*}
  & a_1 := \text{algebra.slice}(\text{in}, W); \\
  & a_2 := \text{algebra.select}(a_1, A_0, ">"); \\
  & a_3 := \text{aggr.sum}(a_2); \\
  & a_4 := \text{aggr.count}(a_2); \\
  & a_5 := \text{calc.} / (a_3, a_4);
\end{align*}
\]

\[
\begin{align*}
  & \text{in}_1 := \text{algebra.slice}(a_1, b_{w1}); \\
  & \text{in}_2 := \text{algebra.slice}(a_1, b_{w2}); \\
  & \text{in}_3 := \text{algebra.slice}(a_1, b_{w3}); \\
  & \quad \ldots \\
  & \text{in}_n := \text{algebra.slice}(a_1, b_{w_n}); \\
  & a_1 := \text{mat.new}(\text{in}_1, \text{in}_2, \ldots, \text{in}_n);
\end{align*}
\]

\[
\begin{align*}
  & \text{Factory}_{\text{CQ}1}(A_0 := 5) \{ \\
  & \quad \text{WHILE true DO} \\
  & \quad \quad \ldots \\
  & \quad \quad \text{in}_1 := \text{algebra.slice}(a_1, 0, 1); \\
  & \quad \quad z_1 := \text{algebra.select}(\text{in}_1, A_0, ">"); \\
  & \quad \quad \text{sum}_1 := \text{aggr.sum}(z_1); \\
  & \quad \quad \text{in}_2 := \text{algebra.slice}(a_1, 3, 2); \\
  & \quad \quad z_2 := \text{algebra.select}(\text{in}_2, A_0, ">"); \\
  & \quad \quad \text{sum}_2 := \text{aggr.sum}(z_2); \\
  & \quad \quad \ldots \\
  & \quad \quad \text{sumPack} := \text{mat.pack}(\text{sum}_1, \text{sum}_2, \ldots, \text{sum}_n) \\
  & \quad \quad \text{cntPack} := \text{mat.pack}(z_1, z_2, \ldots, z_n) \\
  & \quad \quad a_4 := \text{aggr.count}(\text{cntPack}); \\
  & \quad \quad a_5 := \text{calc.} / (\text{sumPack}, \text{cntPack}); \\
  & \quad \quad \text{basket.append}(\text{output}, a_5); \\
  & \quad \quad \text{basket.empty}(\text{in}, b_{w1}); \\
  & \quad \quad \text{suspend}(); \\
  & \quad \text{END WHILE} \\
  & \}
\]
Set of Optimizer Rules

Incrementalist: generates the differential query plan + the transition state

Factory_CQ1(A0:=5){

WHILE true DO
  in1 := algebra.slice(a1,0,1);
  ...  
  in4 := algebra.slice(a1,6,7);
  
  z1 := algebra.select(in1,A0,">");
  sum1 := aggr.sum(z1);
  ... 
  z4 := algebra.select(in4,A0,">");
  sum4 := aggr.sum(z4);

  sumPack:=mat.pack(sum1,sum2,sum3,sum4)
  cntPack:=mat.pack(z1,z2,z3,z4)
  a4 := aggr.count(cntPack);
  a5 := calc./(sumPack,a4);
  basket.append(output,a5);
  basket.empty(in,2);
  suspend();

END WHILE
}
Incrementalist: generates the differential query plan + the transition state

```
Factory_CQ1(A0:=5) {

    WHILE true DO
        in1 := algebra.slice(a1,0,1);
        ...
        in4 := algebra.slice(a1,6,7);
        z1 := algebra.select(in1,A0,">");
        sum1 := aggr.sum(z1);
        ...
        z4 := algebra.select(in4,A0,">");
        sum4 := aggr.sum(z4);
        sumPack:=mat.pack(sum1,sum2,sum3,sum4)
        cntPack:=mat.pack(z1,z2,z3,z4)
        a4 := aggr.count(cntPack);
        a5 := calc./(sumPack,a4);
        basket.append(output,a5);
        basket.empty(in,2);
        suspend();
    END WHILE
}
```
Incrementalist: generates the **differential** query plan + the **transition** state

```python
Factory_CQ1(A0:=5){

  WHILE true DO
    in1 := algebra.slice(a1,0,1);
    ...
    in4 := algebra.slice(a1,6,7);
    z1 := algebra.select(in1,A0,">");
    sum1 := aggr.sum(z1);
    ...
    z4 := algebra.select(in4,A0,">");
    sum4 := aggr.sum(z4);
    sumPack:=mat.pack(sum1,sum2,sum3,sum4)
    cntPack:=mat.pack(z1,z2,z3,z4)
    a4 := aggr.count(cntPack);
    a5 := calc./(sumPack,a4);
    basket.append(output,a5);
  }
  basket.empty(in,2);
  suspend();
  END WHILE
}
Set of Optimizer Rules

Incrementalist: generates the \textit{differential} query plan + the \textit{transition} state

**Factory\textsubscript{CQ1}(A0:=5)**

```plaintext
WHILE true DO
    in\textsubscript{1} := algebra.slice(a1,0,1);
    ...
    in\textsubscript{4} := algebra.slice(a1,6,7);
    z\textsubscript{1} := algebra.select(in\textsubscript{1},A0,">" );
    sum\textsubscript{1} := aggr.sum(z\textsubscript{1});
    ...
    z\textsubscript{4} := algebra.select(in\textsubscript{4},A0,">" );
    sum\textsubscript{4} := aggr.sum(z\textsubscript{4});
    sum\textsubscript{Pack}:=mat.pack(sum\textsubscript{1},sum\textsubscript{2},...,sum\textsubscript{n-1})
    cnt\textsubscript{Pack}:=mat.pack(z\textsubscript{1},z\textsubscript{2},...,z\textsubscript{n})
    a\textsubscript{4} := aggr.count(cnt\textsubscript{Pack});
    a\textsubscript{5} := calc.\slash{(sum\textsubscript{Pack},a\textsubscript{4})};
    basket.append(output,a\textsubscript{5});
    suspend();
END WHILE
```

**Factory\textsubscript{CQ1}(A0:=5)**

```plaintext
WHILE true DO
    in\textsubscript{1} := algebra.slice(a1,0,1);
    ...
    in\textsubscript{4} := algebra.slice(a1,6,7);
    z\textsubscript{1} := algebra.select(in\textsubscript{1},A0,">" );
    sum\textsubscript{1} := aggr.sum(z\textsubscript{1});
    ...
    z\textsubscript{4} := algebra.select(in\textsubscript{4},A0,">" );
    sum\textsubscript{4} := aggr.sum(z\textsubscript{4});
    sum\textsubscript{Pack}:=mat.pack(sum\textsubscript{1},sum\textsubscript{2},...,sum\textsubscript{n-1})
    cnt\textsubscript{Pack}:=mat.pack(z\textsubscript{1},z\textsubscript{2},...,z\textsubscript{n})
    a\textsubscript{4} := aggr.count(cnt\textsubscript{Pack});
    a\textsubscript{5} := calc.\slash{(sum\textsubscript{Pack},a\textsubscript{4})};
    basket.append(output,a\textsubscript{5});
    suspend();
END WHILE
```
Plan-level Incremental Processing in DataCell

- Discard useless input

- **Generic plan rewriting** at the optimization phase
  - join, aggregations, group-by, order-by
  - multi-stream queries
  - joins between streams and tables
  - split the plan as deep as possible

- Exploit column-store intermediates

- Row-store Processing

- Landmark Window Queries

- Time-based sliding windows
Basic Performance: response time for the first 20 windows

```
INSERT into output (y1,y2)
SELECT max(stream1.x1),
       avg(stream2.x1)
FROM stream1, stream2
WHERE stream1.x2=stream2.x2;
```

\[ W : 1024 \times 10^5 \]

\[ bw : 1600 \]
The most important issue though is that it does not functionalities of getting the performance metrics did not queries. System Streams seemed to work correctly but the crashes that occurred repeatedly when running continuous code. However, we did not manage to analyze and fix the crashes. For example, TelegraphCQ compiled on our contemporary comparing against any of these open source stream systems we are not aware of any stream papers projects and are not supported anymore making it very difficult to use them. In fact, we are not aware of any stream systems as they are very small anyway. However, as the overhead involved around the incremental logic in a stream system never starves waiting for tuples, representing the best behavior. In Figure, we see that plain DataCell is losing its edge for the very small sizes. This holds for both DataCell and SystemX, with the latter even outperforming the stream solutions in the smaller sizes.

The amount of data to be processed is so small that simply the administrative cost of simply calling these operators required to make these combinations is attributed to the query processing cost which as we see is significantly more expensive. This cost is rather stable given the results of the above experiments, the response time of the group by window in this case. The query processing cost is used to consume a number of sliding windows and produce all intermediates to be combined and thus more operator calls are maintained. Their total size remains invariant. However, with more basic windows, a larger number of intermediates is even outperforming the stream solutions in the smaller sizes.

INSERT into output (y1, y2)
SELECT max(stream1.x1),
       avg(stream2.x1)
FROM stream1, stream2
WHERE stream1.x2 = stream2.x2;

WHERE stream1.x2 = stream2.x2;

For this experiment, we use the double stream Query 7. Overall, we test our DataCell prototype against a state-of-the-art commercial specialized engine. Due to license restrictions, we refrain from revealing the actual system and we will refer to it as SystemX. In addition, we tested a few open source systems but we were not successful in installing and using them to use them. These systems were academic systems but we were not successful in installing and using them. In addition, we tested a few open source systems.

Here we test the complete software stack of DataCell, i.e., smaller than Figure, for small windows, i.e., smaller than Figure, for bigger windows. For very small window size, the behavior than with Query 6. This time, the query processing cost becomes negligible while the merging cost is the one we see.

Optimization.

Notice also that there is a small rise in the total incremental cost with many basic windows. In Figure, w follows the same trend. What happens is that the total size of intermediates is invariant with invariant window size, regardless of the step size.

The metric reported is the total time needed for the system to parse the file and load the proper column baskets for each file is organized in rows, i.e., a typical csv file. DataCell has it is passed into the system for query processing. The input data is read from an input file in chunks. It is parsed and then tuples in the most demanding case. The most lightweight case and with in total, we feed the system to consume a number of sliding windows and produce all.

<table>
<thead>
<tr>
<th>Window size (tuples*1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Window size (tuples*1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Window size (tuples*1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Here, we test our DataCell prototype against a state-of-the-art commercial engine. We use the double stream Query 7. Overall, we test our DataCell prototype against a state-of-the-art commercial engine. Due to license restrictions, we refrain from revealing the actual system and we will refer to it as SystemX. In addition, we tested a few open source systems but we were not successful in installing and using them to use them.
database

strength

maximum speed

short-term distance
stream engine

lightweight

long-term distance

forgets to stop

queries of data
database

stream engines
database  DataCell  stream engines
database  DataCell  stream engines

approximate query processing

adaptivity

hybrid
Part 3

dbTouch
analytics at your fingertips

Stratos Idreos, Erietta Liarou

CIDR 2013, ICDE 2014