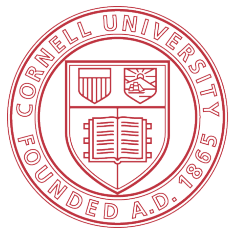
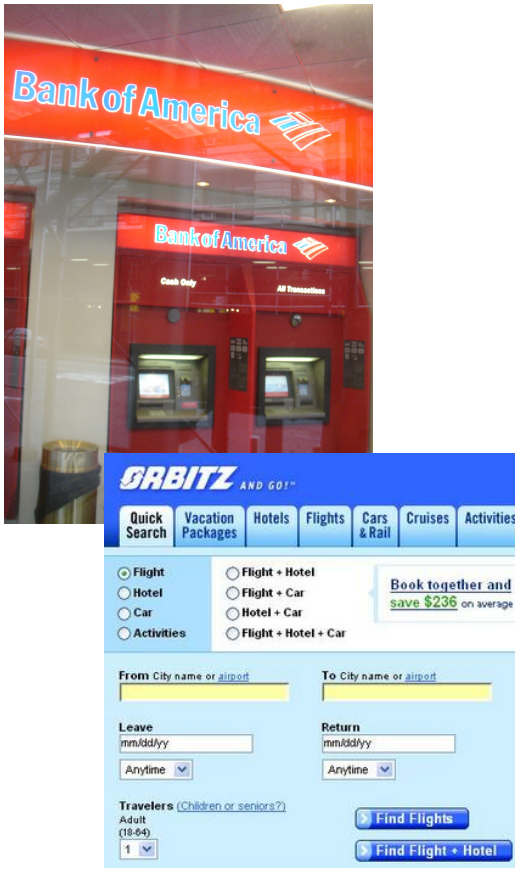
Lightweight Incremental Query Processing for Update-Intensive Applications

Yanif Ahmad
yanif@cs.cornell.edu
Database Group, Cornell University



DBMS Apps in the 1960s

- Online transaction processing (OLTP)
  - Transaction: unit of work, with ACID properties
  - Applied in: banking, airlines, e-commerce, etc.
  - Example queries:
    - what were the withdrawals I made from my checking account yesterday?
    - what direct flights are available on March 15th from New York to Rome?

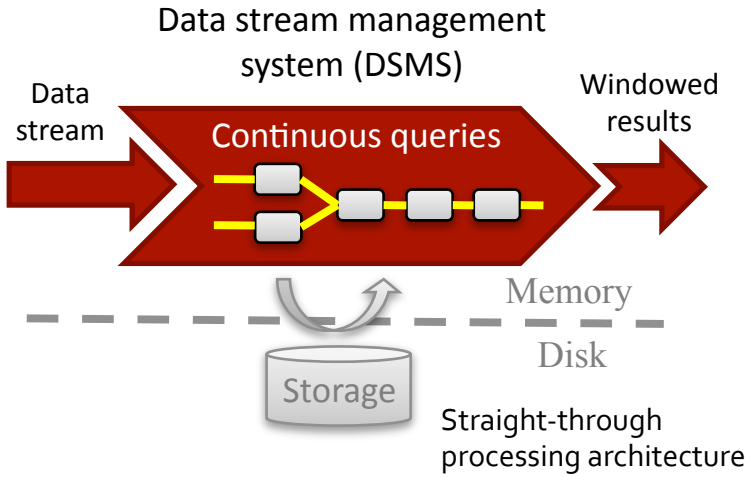


DBMS Apps in the 1980s

- Online analytical processing (OLAP)
  - Analytical queries compute categorized statistics (i.e. aggregates)
  - Used for business intelligence (sales, marketing, logistics, etc.)
  - Example queries:
    - what is the total number of games consoles purchased by females between 19 and 45?
    - what were the total sales of digital cameras in NY state in 2008 and 2009?

DBMS Apps in the Mid-2000s

- Data stream processing:
  - Continuously arriving data, continuously evaluated queries
  - Applications:
    - Computer network monitoring
    - Environmental monitoring
  - Example query:
    - which weather stations detected wind gusts above 40 mph and 20 mph in the last 2 hours?



Apps in the 2010s: Algorithmic Trading

- High-frequency algorithmic trading on order books
  - Q1/2009: 73% of all equities trades in the US
  - ~15% annual profit margin industry-wide

- Order book trading
  - Actions:
    - Algos: insertions, deletions of bid and ask orders
    - Exchange: matching (potentially partial) of bids and asks
  - Queries:
    - Algo strategies
    - Exchange simulation for backtesting

t = timestamp oid = order id bid = broker id p = price v = volume

Bid order book (buyers)

t	oid	bid	p	v
2526035	36721	NITE	184000	1500
2526690	36909	MASH	183200	200
2527543	37001	MSCO	182700	3000
2528321	37008	GSCO	182500	500
2529032	37011	FBCO	181900	600

Electronic exchange
(e.g., NYSE, NASDAQ)

Ask order book (sellers)

t	oid	bid	p	v
2526345	36750	GSCO	184000	1000
2527389	37002	MSCO	185200	200
2527928	37006	GSCO	186100	500
2528894	37020	NITE	186800	500
2529758	37032	MASH	187900	700

Apps in the 2010s: Algorithmic Trading

- High-frequency algorithmic trading on order books
  - Q1/2009: 73% of all equities trades in the US
  - ~15% annual profit margin industry-wide

- Order book trading
  - Actions:
    - Algos: insertions, deletions of bid and ask orders
    - Exchange: matching (potentially partial) of bids and asks
  - Queries:
    - Algo strategies
    - Exchange simulation for backtesting

t = timestamp oid = order id bid = broker id p = price v = volume

Bid order book (buyers)

t	oid	bid	p	v
2526035	36721	NITE	184000	1500
2526690	36909	MASH	183200	200
2530574	**37055**	**NITE**	**183000**	**300**
[illegible]				
2528321	37008	GSCO	182500	500
2529032	37011	FBCO	181900	600

Electronic exchange
(e.g., NYSE, NASDAQ)

Ask order book (sellers)

t	oid	bid	p	v
2526345	36750	GSCO	184000	1000
2527389	37002	MSCO	185200	200
2527928	37006	GSCO	186100	500
2531230	**37075**	**FBCO**	**186400**	**1000**
2528894	37020	NITE	186800	500
2529758	37032	MASH	187900	700

Apps in the 2010s: Algorithmic Trading

- High-frequency algorithmic trading on order books
  - Q1/2009: 73% of all equities trades in the US
  - ~15% annual profit margin industry-wide

- Order book trading
  - Actions:
    - Algos: insertions, deletions of bid and ask orders
    - Exchange: matching (potentially partial) of bids and asks
  - Queries:
    - Algo strategies
    - Exchange simulation for backtesting

t = timestamp oid = order id bid = broker id p = price v = volume

Bid order book (buyers)

t	oid	bid	p	v
2526035	36721	NITE	184000	1500
2526690	36909	MASH	183200	200
2530574	37055	NITE	183000	300
2528321	37008	GSCO	182500	500
2529032	37011	FBCO	181900	600

Electronic exchange
(e.g., NYSE, NASDAQ)

Ask order book (sellers)

t	oid	bid	p	v
2526345	36750	GSCO	184000	1000
2527389	37002	MSCO	185200	200
2527928	37006	GSCO	186100	500
2531230	37075	FBCO	186400	1000
2528894	37020	NITE	186800	500
2529758	37032	MASH	187900	700

Apps in the 2010s: Algorithmic Trading

- High-frequency algorithmic trading on order books
  - Q1/2009: 73% of all equities trades in the US
  - ~15% annual profit margin industry-wide

- Order book trading
  - Actions:
    - Algos: insertions, deletions of bid and ask orders
    - Exchange: matching (potentially partial) of bids and asks
  - Queries:
    - Algo strategies
    - Exchange simulation for backtesting

t = timestamp oid = order id bid = broker id p = price v = volume

Bid order book (buyers)

t	oid	bid	p	v
2526035	36721	NITE	184000	500
2526690	36909	MASH	183200	200
2530574	37055	NITE	183000	300
2528321	37008	GSCO	182500	500
2529032	37011	FBCO	181900	600

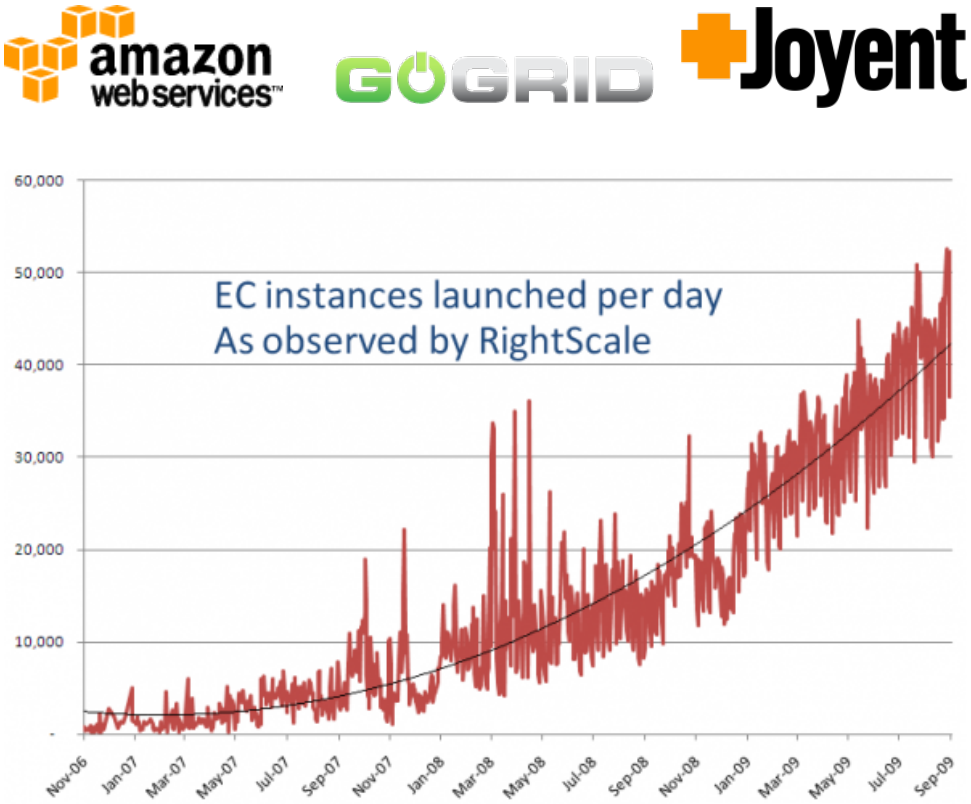
Electronic exchange
(e.g., NYSE, NASDAQ)

Ask order book (sellers)

t	oid	bid	p	v
2527389	37002	MSCO	185200	200
2527928	37006	GSCO	186100	500
2531230	37075	FBCO	186400	1000
2528894	37020	NITE	186800	500
2529758	37032	MASH	187900	700

Apps in the 2010s: Cloud Management

- Data management for cloud infrastructure metadata
- Metadata updates:
  - VM instantiation and migration
  - Hardware replenishment
  - User management
- Queries:
  - Pricing model evaluation
  - Monitoring and billing



Apps in the 2010s: Microblogging, Personal Feeds

- Facebook, Twitter streams
- Status feed update examples:
  - Comments added to threads at any time
  - Posts removed from threads at any time
- Queries:
  - Hot trend analyses
  - Advertising

Update–Intensive Applications

- These are all examples of update-heavy apps!
  - Algorithmic trading updates: order insertions & deletions
  - Microblogging updates: status updates, comments, edits
  - Cloud management updates: slice instantiation, migration
  - Other examples: location-based services, clickstreams, MMOs
- Relational databases are notoriously poor at handling updates
- Update-intensive apps follow a recent trend, namely…
  - When commercial databases cannot handle app's needs…

The Rise of Lightweight Databases

- Communities avoid relational databases
  - Large web companies (Google, Amazon, eBay)
  - NoSQL crowd (Facebook, Digg)
  - Scientific applications (LHC/SLAC, NCAR/NOAA)
- Communities roll their own ***lightweight*** systems
  - Trade off expressiveness and consistency for scalability
- Many examples of this trend:
  - Streams: Streambase, IBM Infosphere Streams, MS StreamInsight
  - Analytics & cloud DBMS: mapreduce, Vertica, Greenplum, HadoopDB
  - Key-value stores: Bigtable, HBase, Dynamo

The DBToaster Project

- Project vision:
  - Develop techniques for generating nimble, robust, lightweight systems for data management applications
  - Reason about query properties from an online (i.e. incremental) perspective, to discover deep properties to exploit for scalability
  - Support an established declarative query language, establish that query evaluation can match hand-coded programs

Update Processing Example

- Algorithmic trading schema

```
Bids(time, order_id, broker_id, price, volume)

Asks(time, order_id, broker_id, price, volume)
```

```
select sum((A.price*A.volume
            - B.price*B.volume)
        * (A.time - B.time))
        as holds
from    Bids B, Asks A
where   B.broker_id = A.broker_id;
```

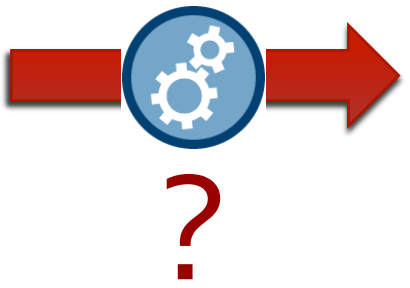
Update-Intensive Application Characteristics

- Database must maintain results for continuous queries
- Database must support arbitrary updates

```
select  sum((A.price*A.volume
            - B.price*B.volume) *
        (A.time - B.time)) as holds
from    Bids B, Asks A
where   B.broker_id = A.broker_id;
```

t	oid	bid	p	v
5	4	2	100	50
3	2	1	90	100
7	6	1	70	25

t	oid	bid	p	v
6	5	2	105	70
2	1	1	110	60
4	3	1	115	50



holds
-325

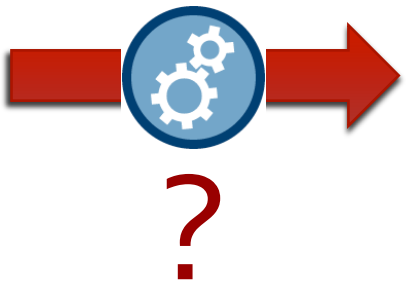
Update-Intensive Application Characteristics

- Database must maintain results for continuous queries
- Database must support arbitrary updates

```
select  sum((A.price*A.volume
            - B.price*B.volume) *
        (A.time - B.time)) as holds
from    Bids B, Asks A
where   B.broker_id = A.broker_id;
```

t	oid	bid	p	v
9	**4**	**2**	**100**	**100**
3	2	1	90	100
7	6	1	70	25

t	oid	bid	p	v
6	5	2	105	70
2	1	1	110	60
4	3	1	115	50



holds
-29150

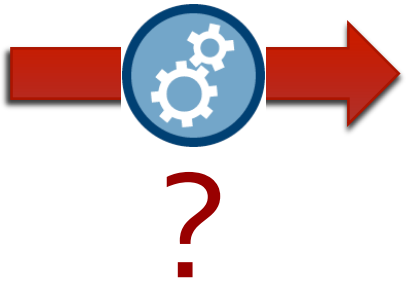
Update-Intensive Application Characteristics

- Database must maintain results for continuous queries
- Database must support arbitrary updates

```
select  sum((A.price*A.volume
            - B.price*B.volume) *
        (A.time - B.time)) as holds
from    Bids B, Asks A
where   B.broker_id = A.broker_id;
```

t	oid	bid	p	v
9	**4**	**2**	**100**	**100**
3	2	1	90	100
7	6	1	70	25

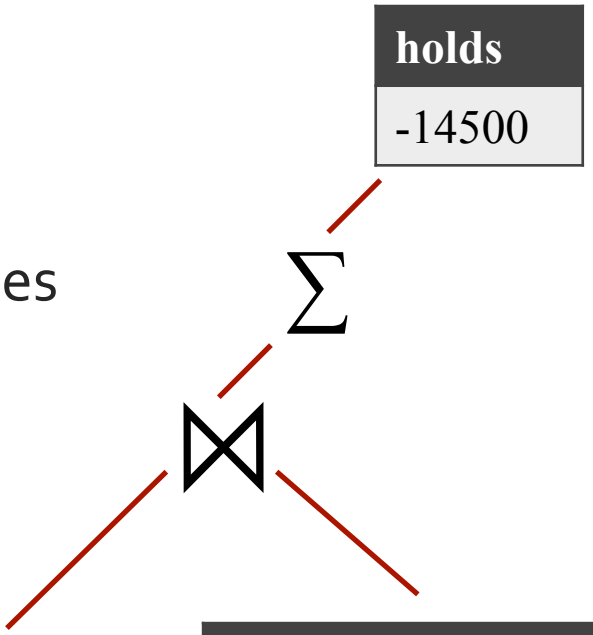
t	oid	bid	p	v
11	**5**	**2**	**105**	**120**
2	1	1	110	60
4	3	1	115	50



holds
-31900

Update-Intensive Application Characteristics

- Database must maintain results for continuous queries
- Database must support arbitrary updates
- Plan-based techniques:
  - Naïve: repeat the query on the whole input, whenever it changes
  - State-of-the-art: view maintenance and stream processing

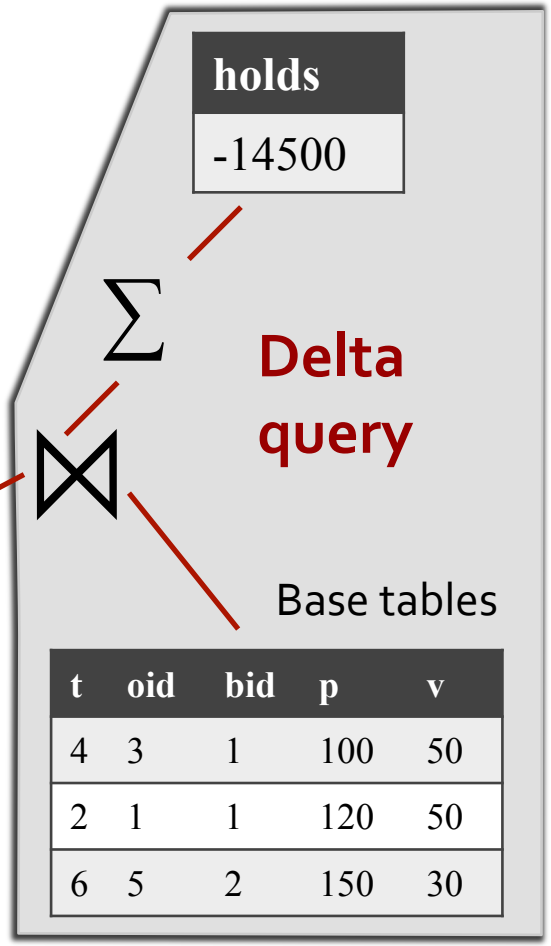


t	oid	bid	p	v
5	4	2	100	50
3	2	1	80	100
7	6	1	80	50

t	oid	bid	p	v
4	3	1	100	50
2	1	1	120	50
6	5	2	150	30

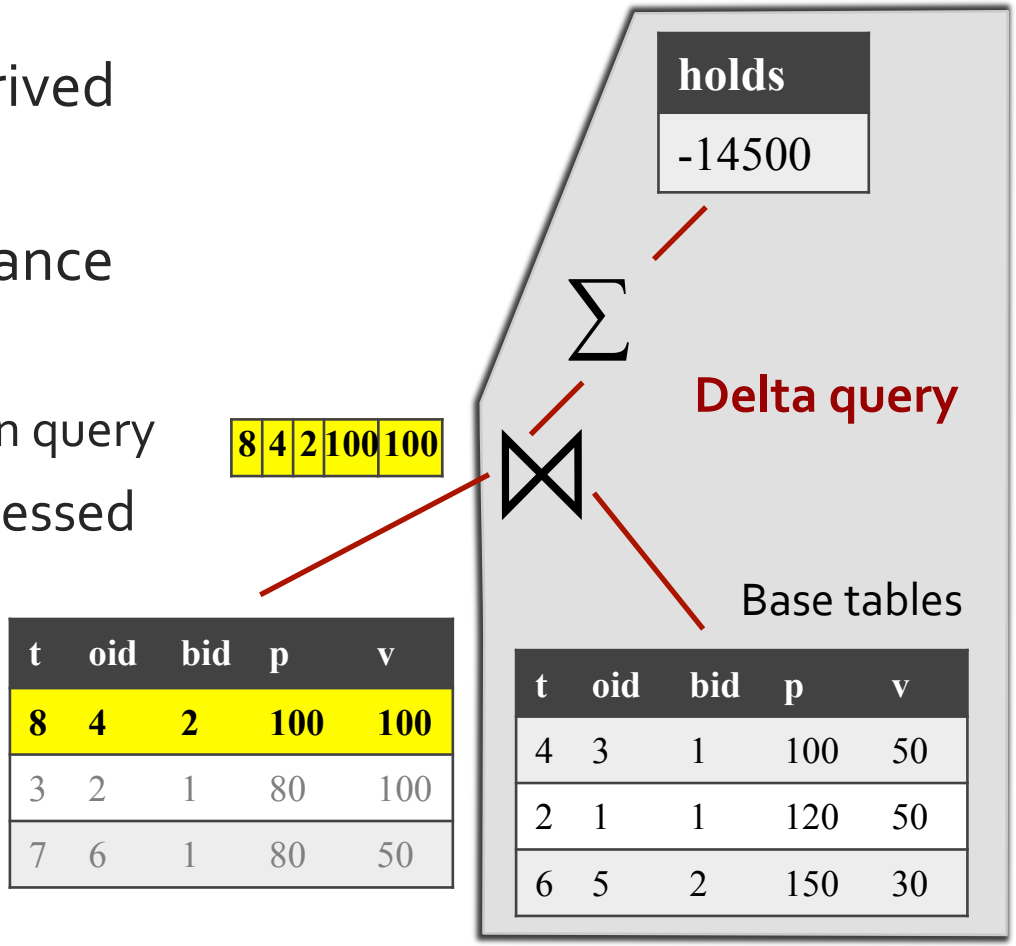
The State of the Art in Update Processing

- Views: logical relations derived from a query's results
- Incremental view maintenance
  - Mechanism: delta queries
    - Simpler than view definition query
  - But… delta queries still processed with classical QP engine
- Well-studied through 1980-today:
  - [Roussopoulos, 1991; Yan and Larson, 1995; Colby et al, 1996; Kotidis and Roussopoulos, 2001; Zhou et al, 2007]



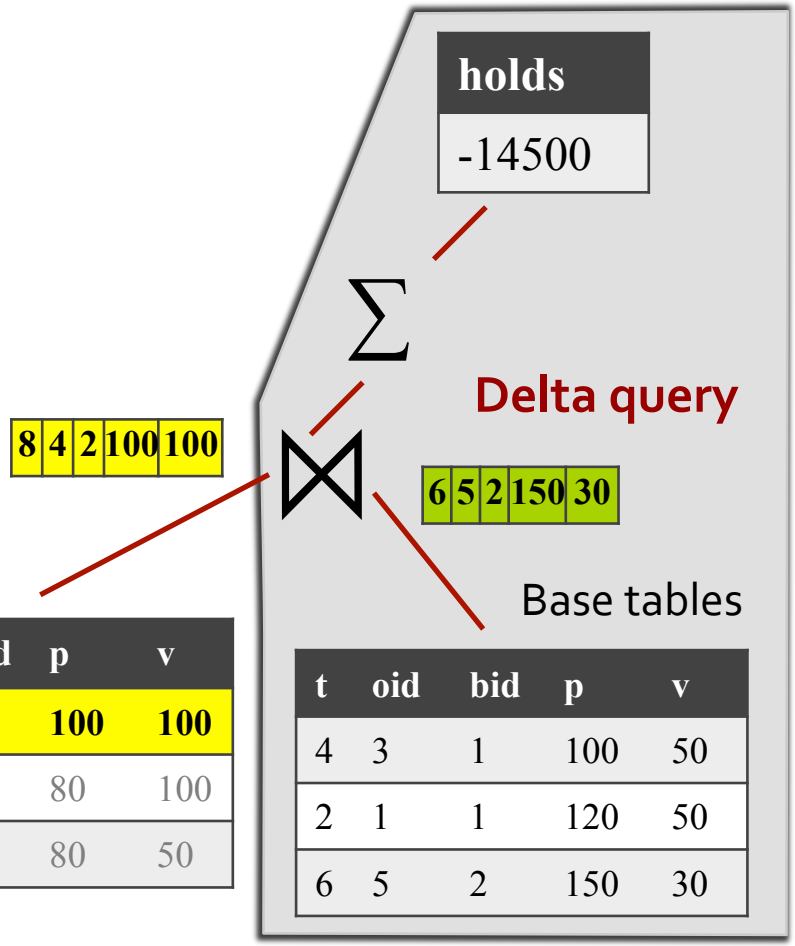
The State of the Art in Update Processing

- Views: logical relations derived from a query's results
- Incremental view maintenance
  - Mechanism: delta queries
    - Simpler than view definition query
  - But… delta queries still processed with classical QP engine
- Well-studied through 1980-today:
  - [Roussopoulos, 1991; Yan and Larson, 1995; Colby et al, 1996; Kotidis and Roussopoulos, 2001; Zhou et al, 2007]



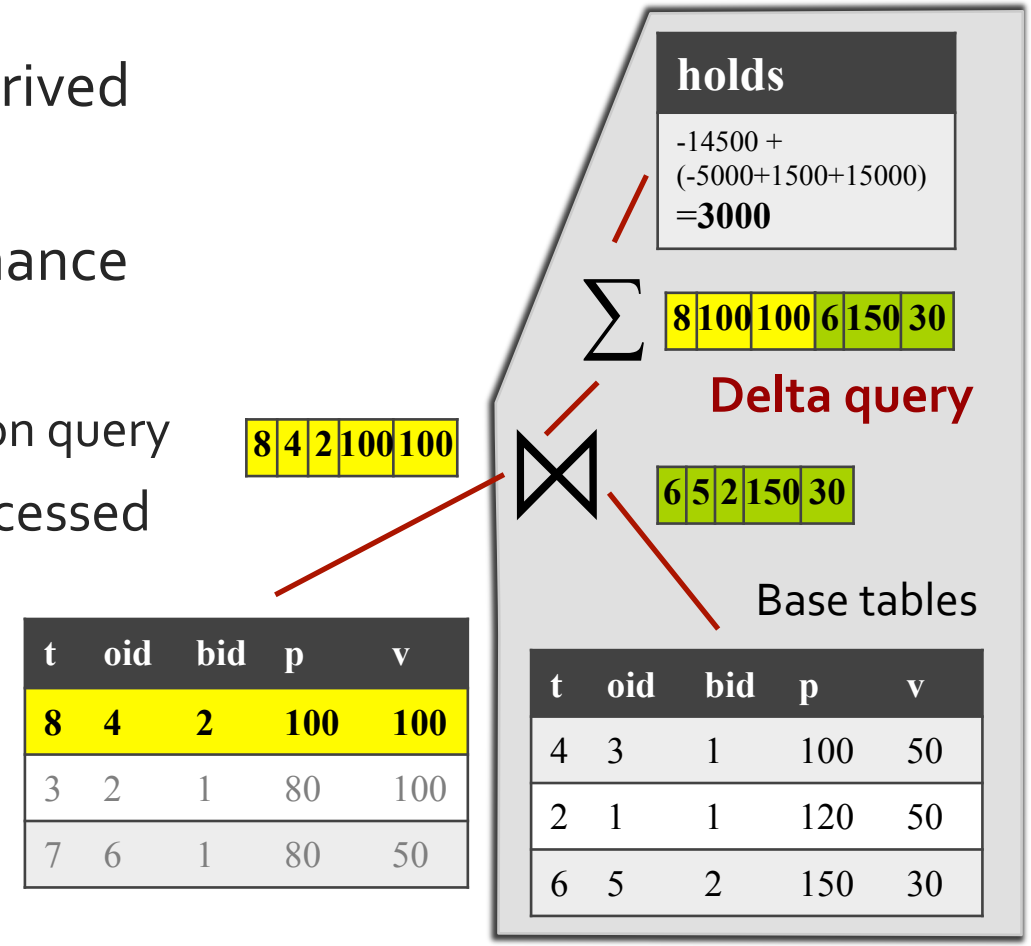
The State of the Art in Update Processing

- Views: logical relations derived from a query's results
- Incremental view maintenance
  - Mechanism: delta queries
    - Simpler than view definition query
  - But… delta queries still processed with classical QP engine
- Well-studied through 1980-today:
  - [Roussopoulos, 1991; Yan and Larson, 1995; Colby et al, 1996; Kotidis and Roussopoulos, 2001; Zhou et al, 2007]



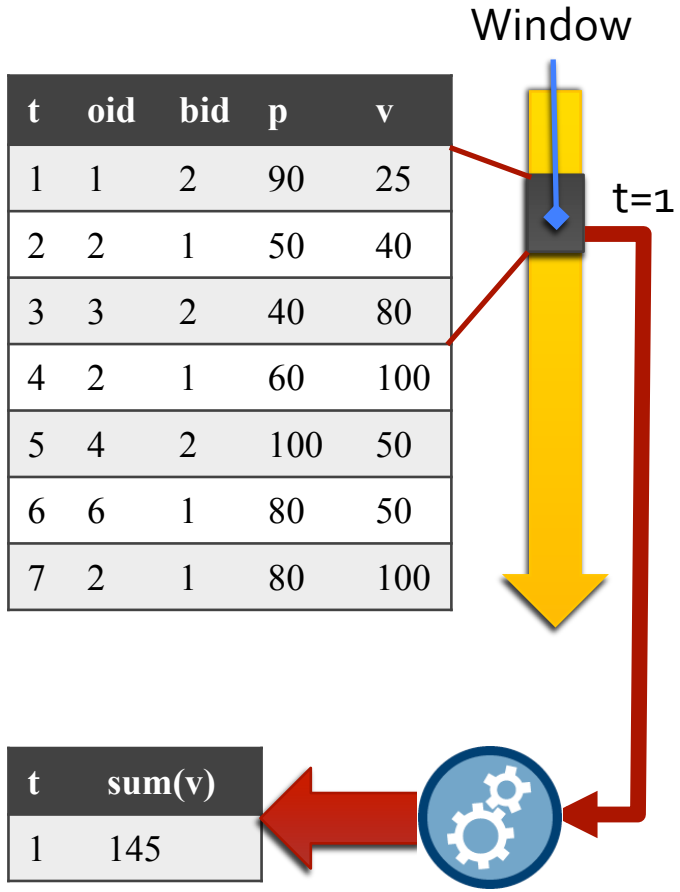
The State of the Art in Update Processing

- Views: logical relations derived from a query's results
- Incremental view maintenance
  - Mechanism: delta queries
    - Simpler than view definition query
  - But… delta queries still processed with classical QP engine
- Well-studied through 1980-today:
  - [Roussopoulos, 1991; Yan and Larson, 1995; Colby et al, 1996; Kotidis and Roussopoulos, 2001; Zhou et al, 2007]



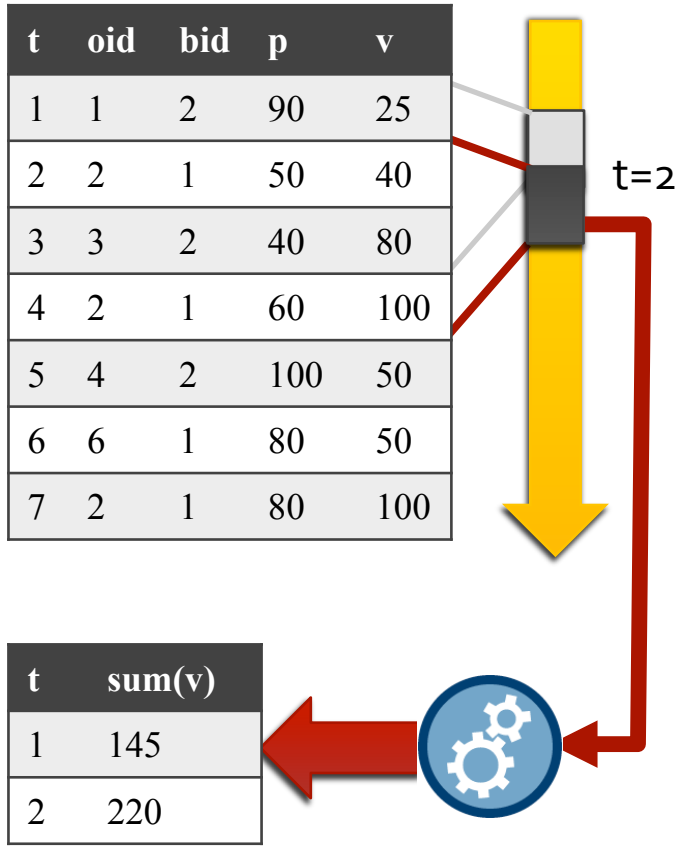
The State of the Art in Update Processing

- Stream processing engines (SPEs)
  - Assumes append-only ordered inputs
  - Processes queries over ***windows*** of input data
  - Windows advance over the input stream, with results produced for each window



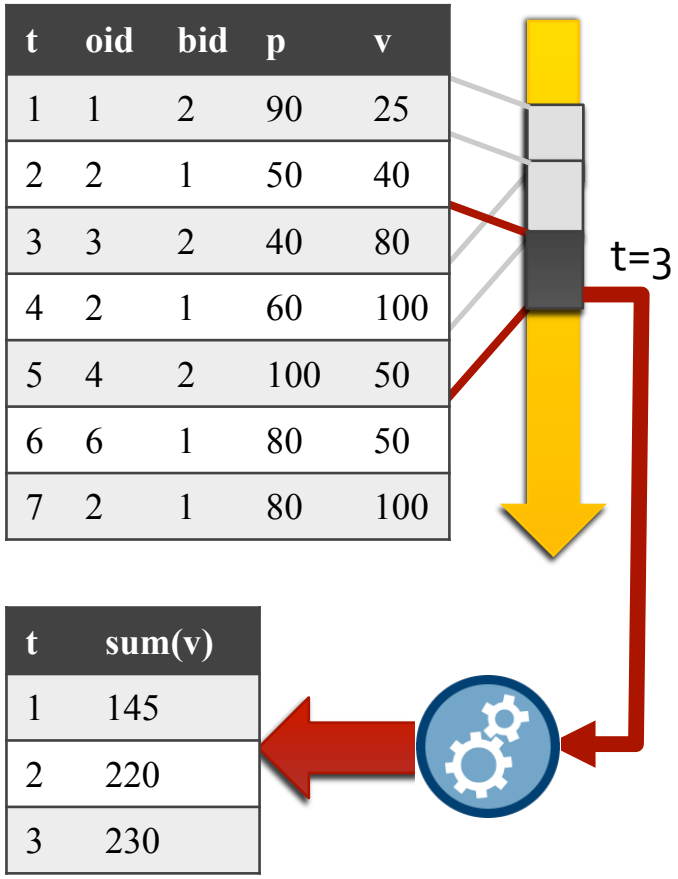
The State of the Art in Update Processing

- Stream processing engines (SPEs)
  - Assumes append-only ordered inputs
  - Processes queries over ***windows*** of input data
  - Windows advance over the input stream, with results produced for each window



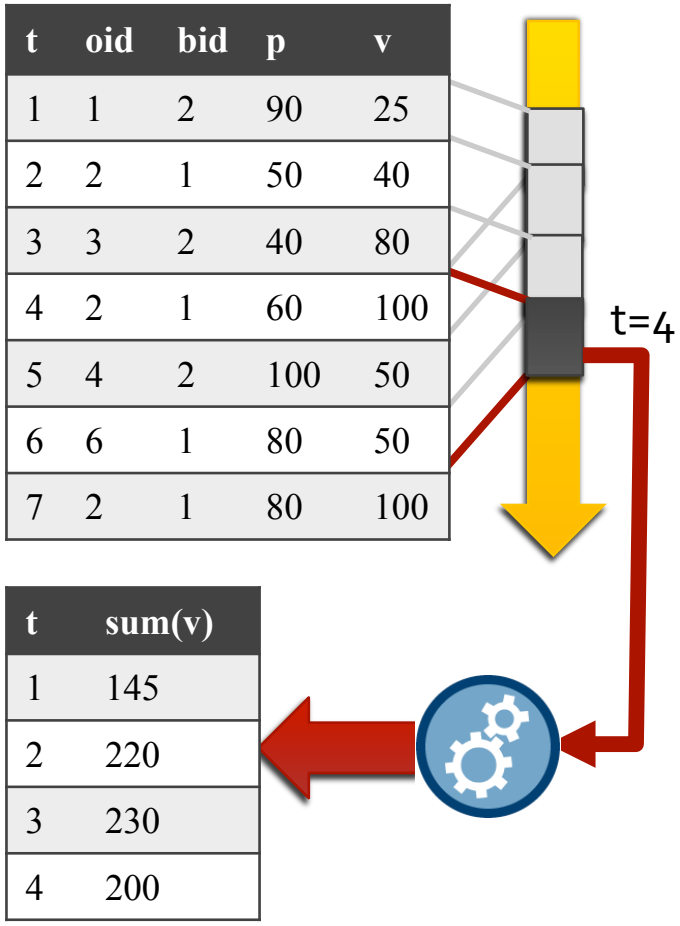
The State of the Art in Update Processing

- Stream processing engines (SPEs)
  - Assumes append-only ordered inputs
  - Processes queries over ***windows*** of input data
  - Windows advance over the input stream, with results produced for each window



The State of the Art in Update Processing

- Stream processing engines (SPEs)
  - Assumes append-only ordered inputs
  - Processes queries over ***windows*** of input data
  - Windows advance over the input stream, with results produced for each window

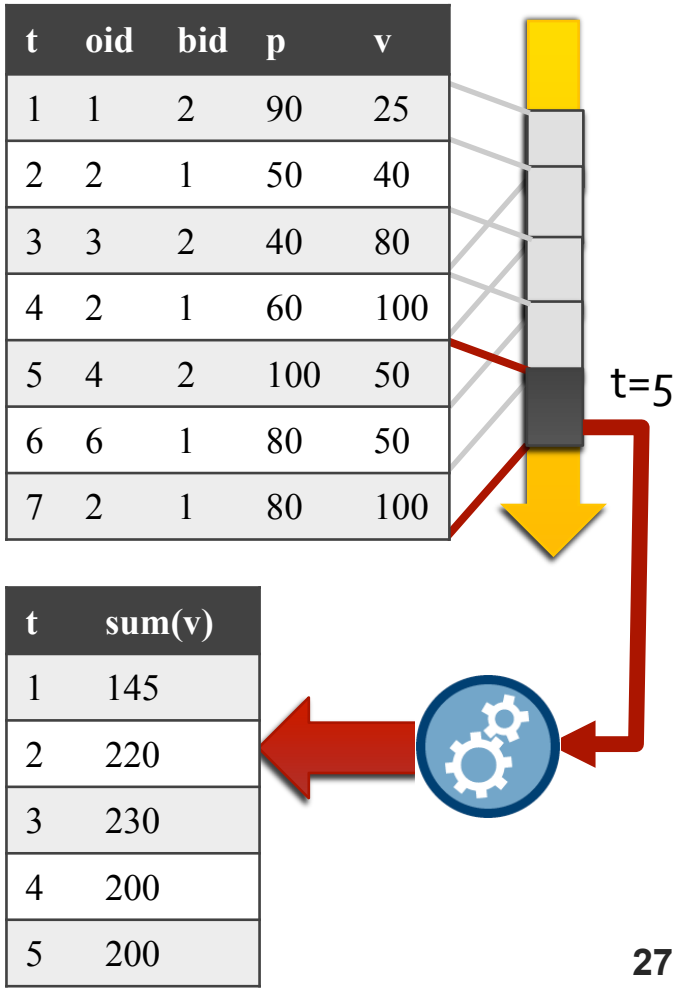


t	oid	bid	p	v
1	1	2	90	25
2	2	1	50	40
3	3	2	40	80
4	2	1	60	100
5	4	2	100	50
6	6	1	80	50
7	2	1	80	100

t	sum(v)
1	145
2	220
3	230
4	200

The State of the Art in Update Processing

- Stream processing engines (SPEs)
  - Assumes append-only ordered inputs
  - Processes queries over ***windows*** of input data
  - Windows advance over the input stream, with results produced for each window

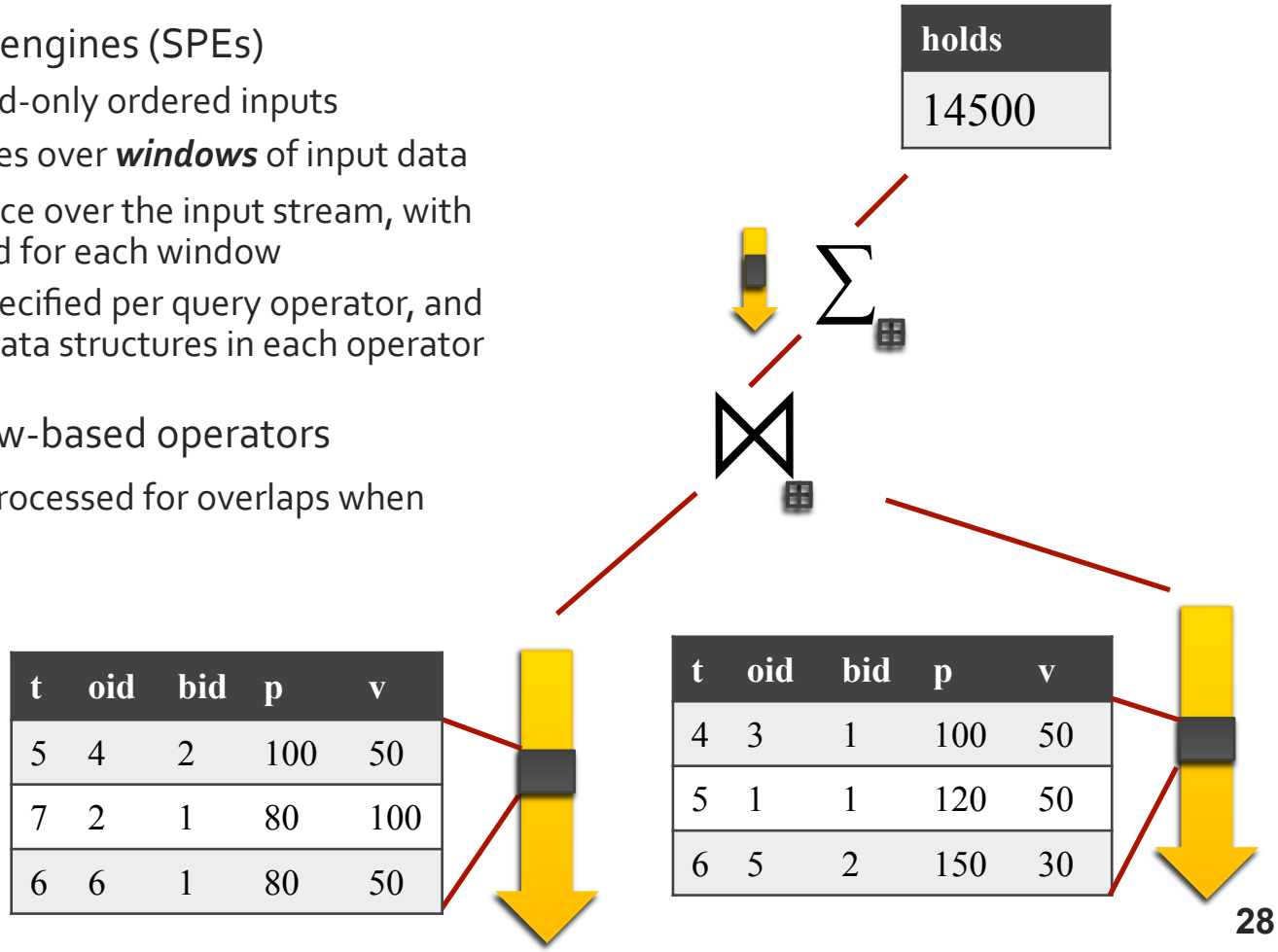


t	oid	bid	p	v
1	1	2	90	25
2	2	1	50	40
3	3	2	40	80
4	2	1	60	100
5	4	2	100	50
6	6	1	80	50
7	2	1	80	100

t	sum(v)
1	145
2	220
3	230
4	200
5	200

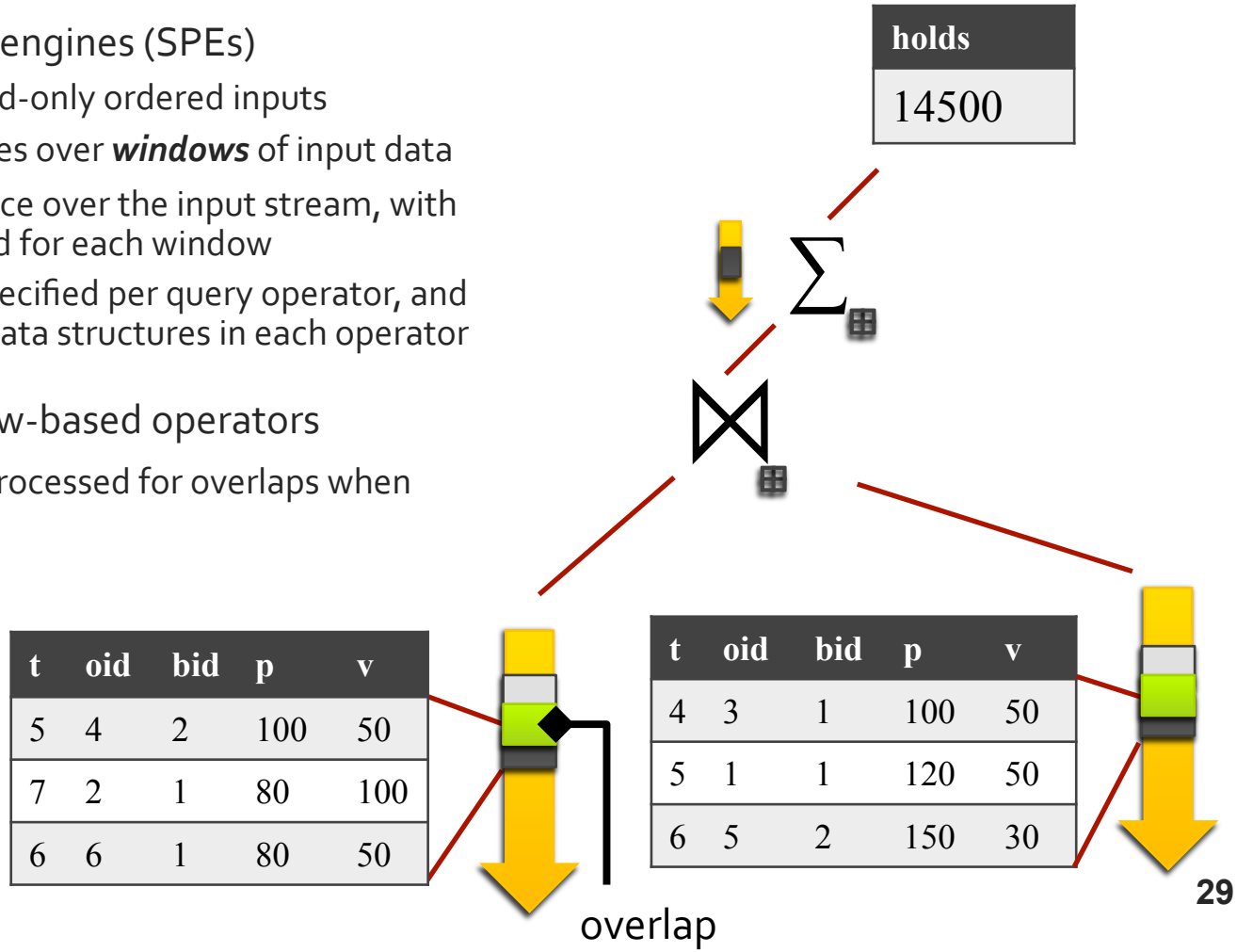
The State of the Art in Update Processing

- Stream processing engines (SPEs)
  - Assumes append-only ordered inputs
  - Processes queries over ***windows*** of input data
  - Windows advance over the input stream, with results produced for each window
  - Windows are specified per query operator, and maintained as data structures in each operator
- Mechanism: window-based operators
  - Incrementally processed for overlaps when windows slide



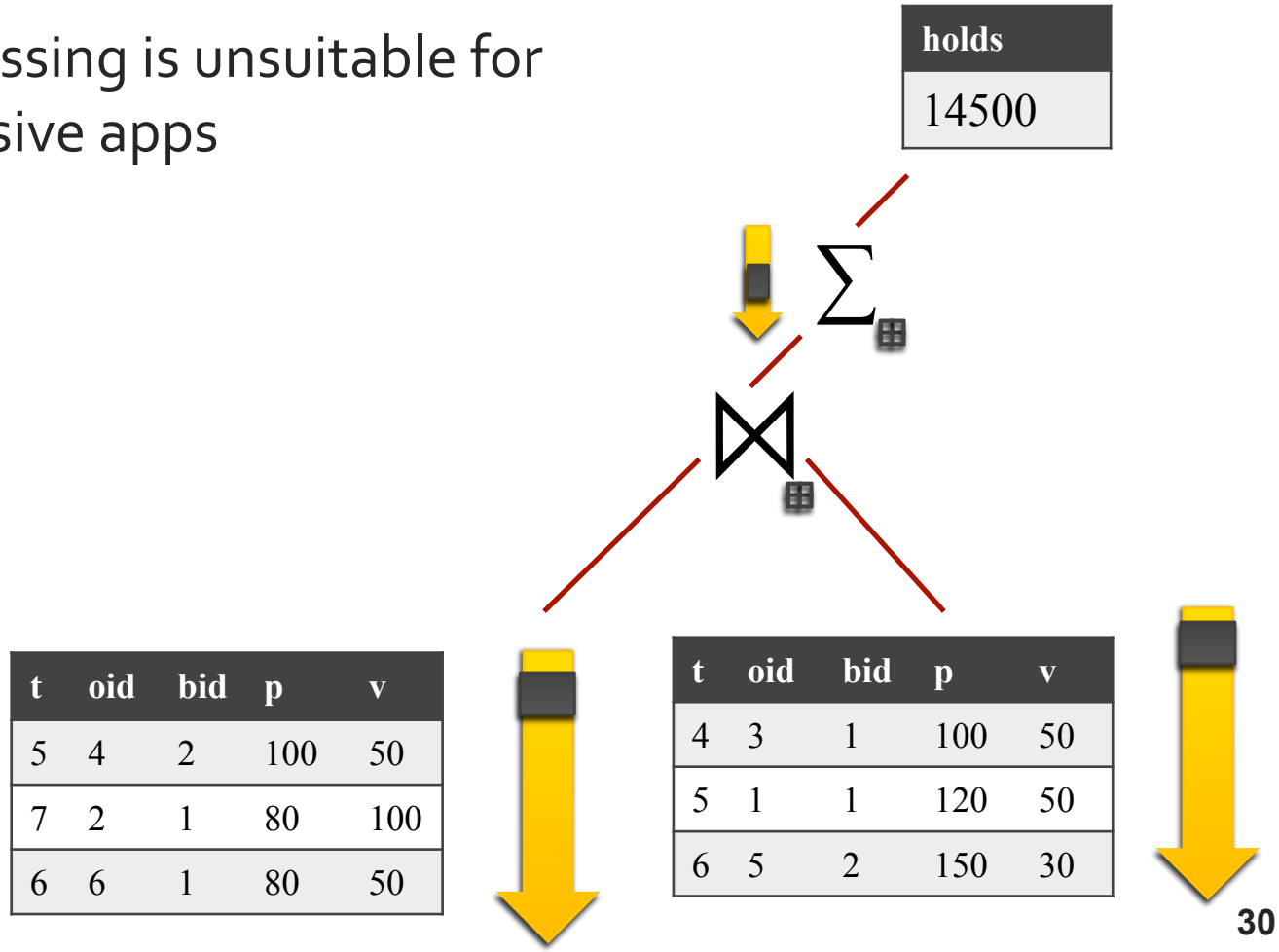
The State of the Art in Update Processing

- Stream processing engines (SPEs)
  - Assumes append-only ordered inputs
  - Processes queries over ***windows*** of input data
  - Windows advance over the input stream, with results produced for each window
  - Windows are specified per query operator, and maintained as data structures in each operator
- Mechanism: window-based operators
  - Incrementally processed for overlaps when windows slide



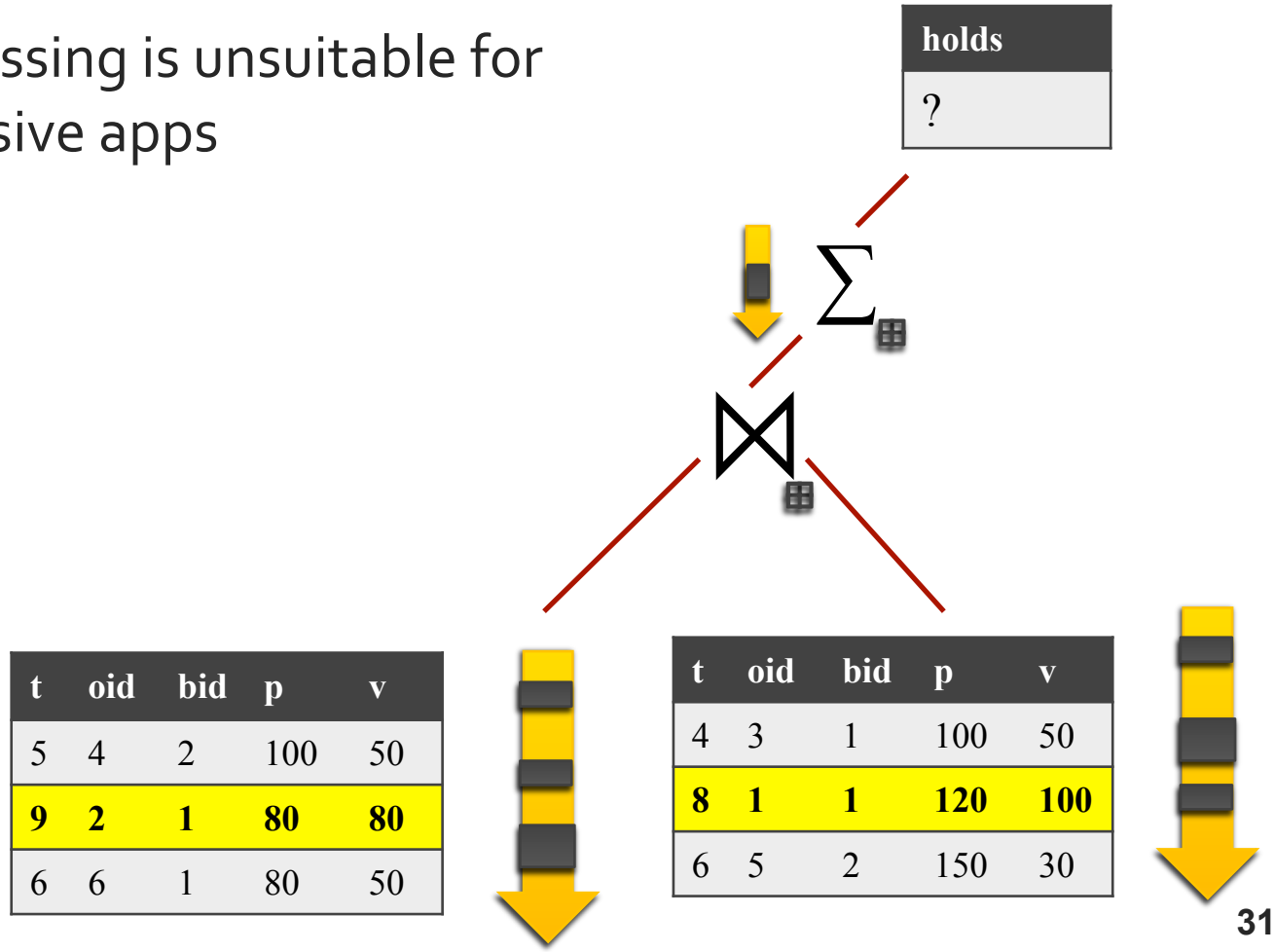
The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps



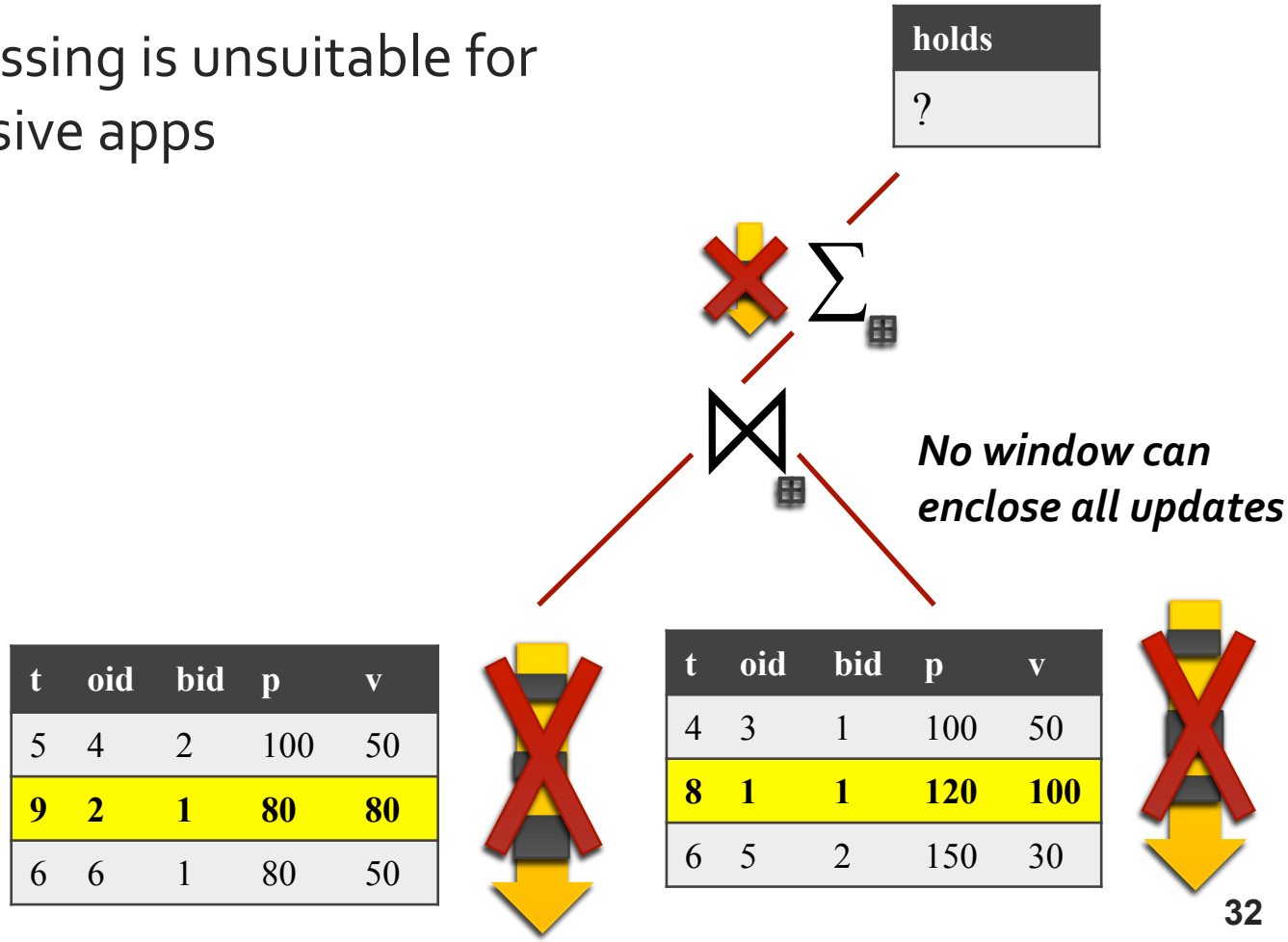
The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps



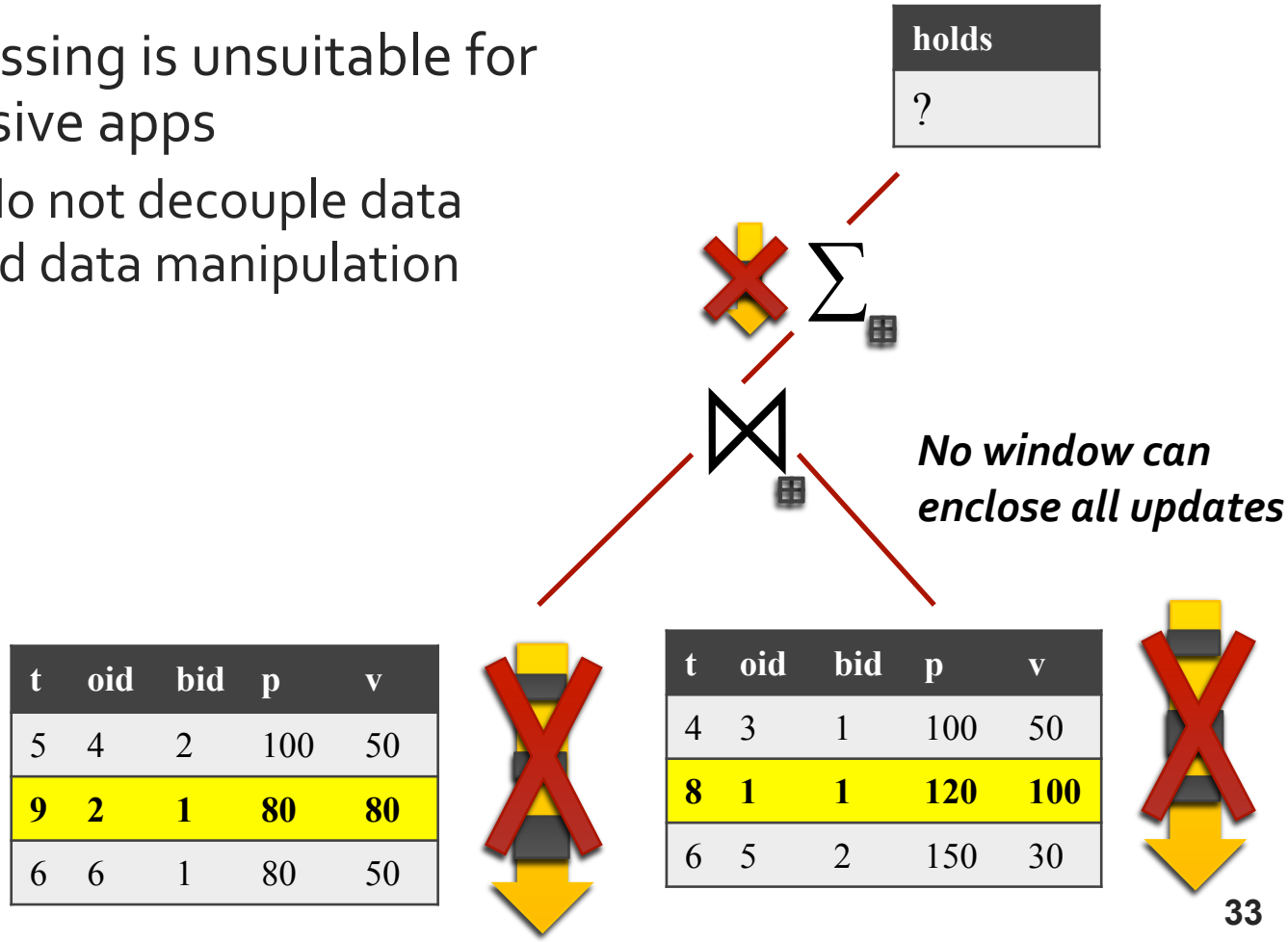
The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps



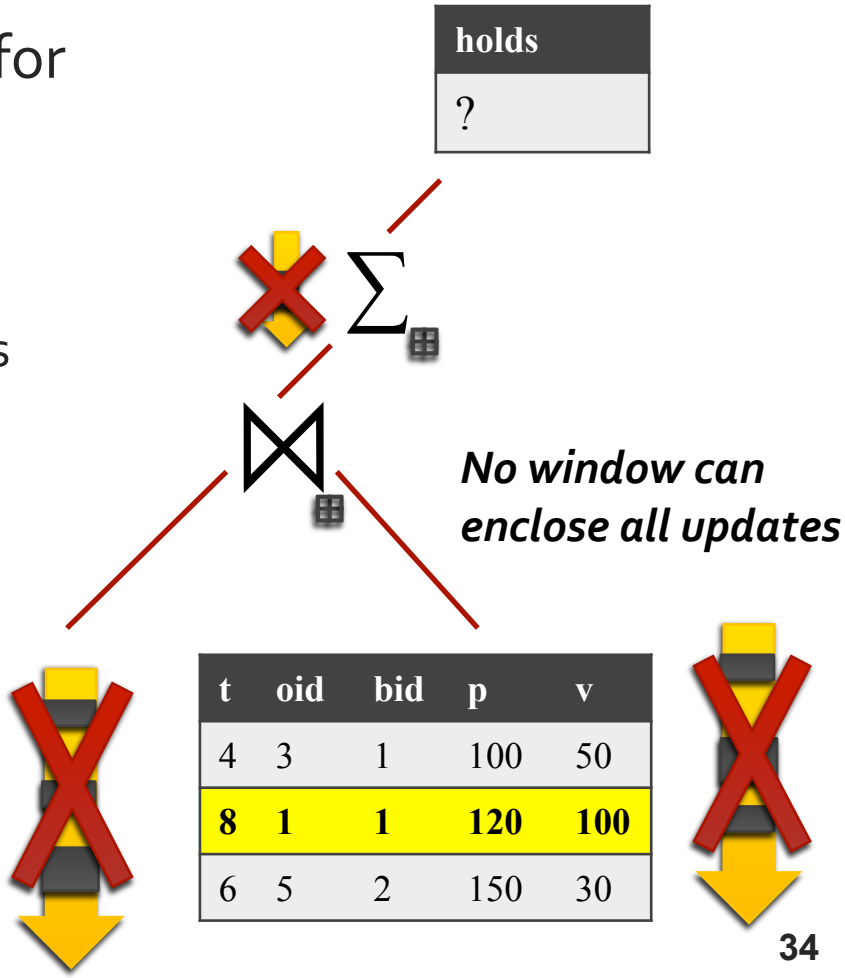
The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps
  - Windows do not decouple data scoping and data manipulation



The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps
  - Windows do not decouple data scoping and data manipulation
    - No "state-of-the-world" queries
    - No OLAP or nested aggregates
  - For general processing, stream engines resort to repetition

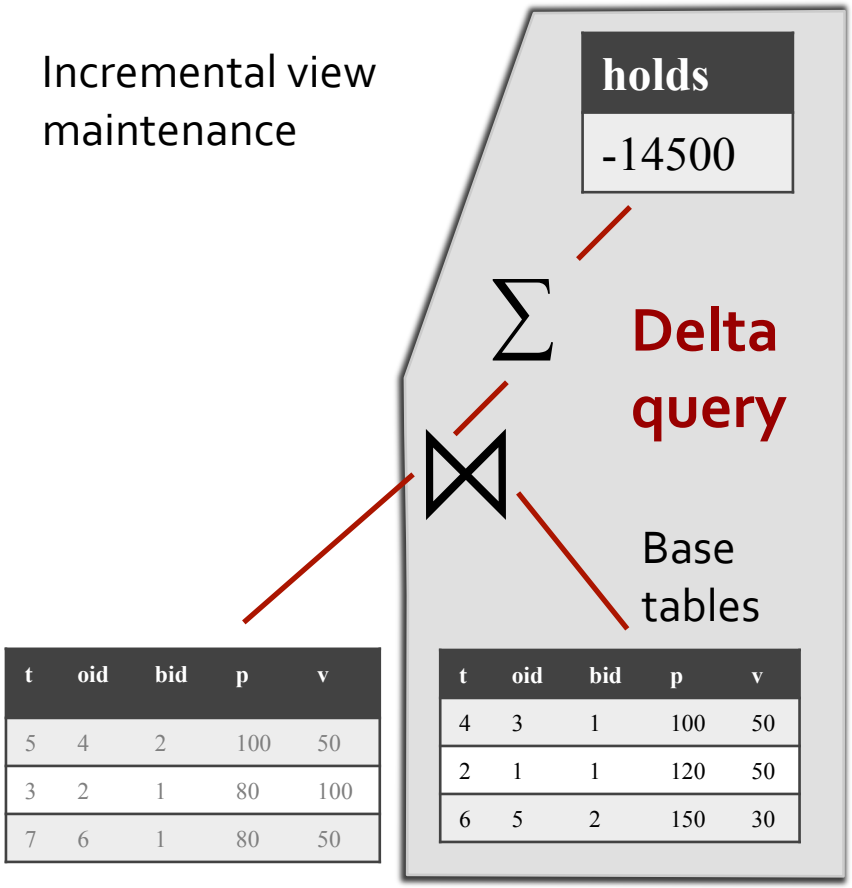


t	oid	bid	p	v
5	4	2	100	50
9	**2**	**1**	**80**	**80**
6	6	1	80	50

t	oid	bid	p	v
4	3	1	100	50
8	**1**	**1**	**120**	**100**
6	5	2	150	30

The State of the Art in Update Processing

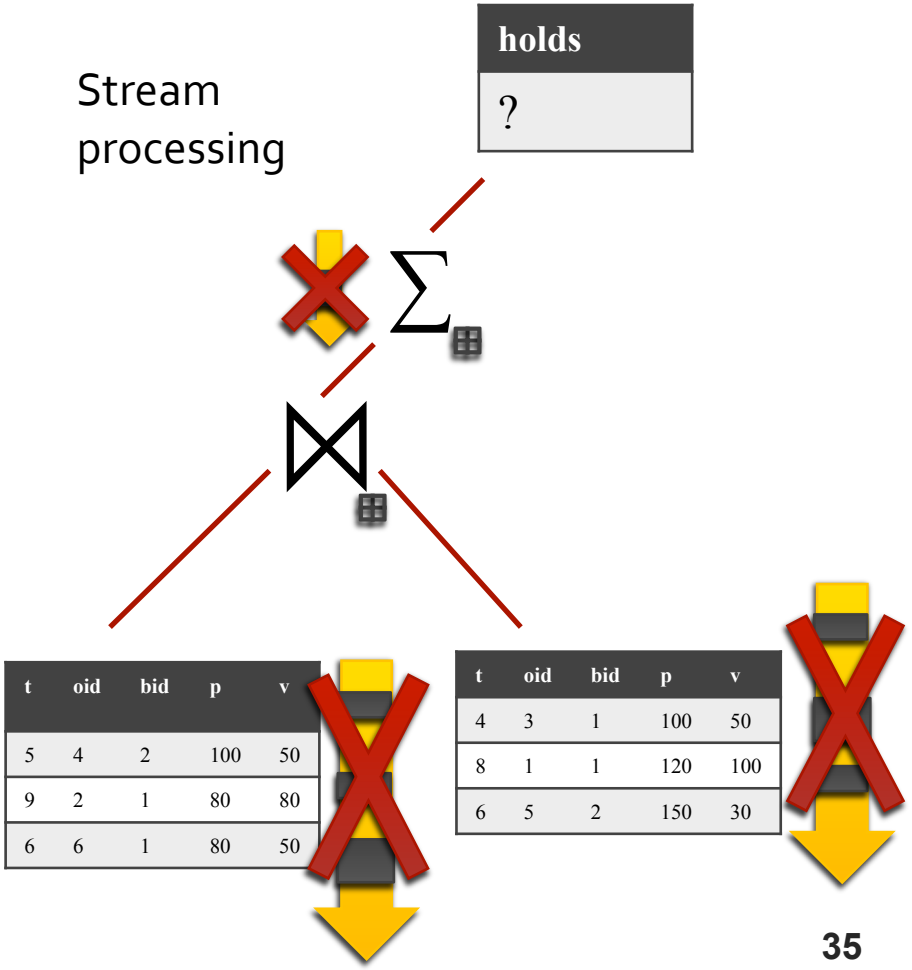
Incremental view maintenance



t	oid	bid	p	v
5	4	2	100	50
3	2	1	80	100
7	6	1	80	50

t	oid	bid	p	v
4	3	1	100	50
2	1	1	120	50
6	5	2	150	30

Stream processing



t	oid	bid	p	v
5	4	2	100	50
9	2	1	80	80
6	6	1	80	50

t	oid	bid	p	v
4	3	1	100	50
8	1	1	120	100
6	5	2	150	30

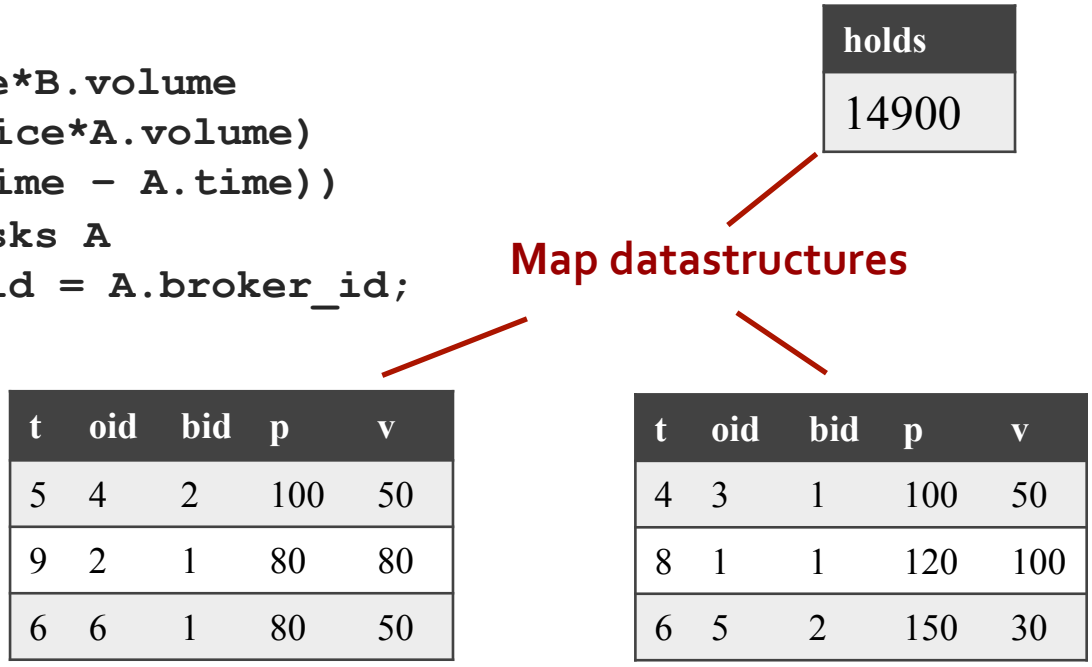
DBToaster: Technical Focus of This Talk

"How do we efficiently, incrementally, process queries on a rapidly-changing, general, database?"

DBToaster: Technical Focus of This Talk

"How do we efficiently, incrementally, process queries on a rapidly-changing, general, database?"

```
select sum((B.price*B.volume
            - A.price*A.volume)
            * (B.time - A.time))
from     Bids B, Asks A
where    B.broker_id = A.broker_id;
```

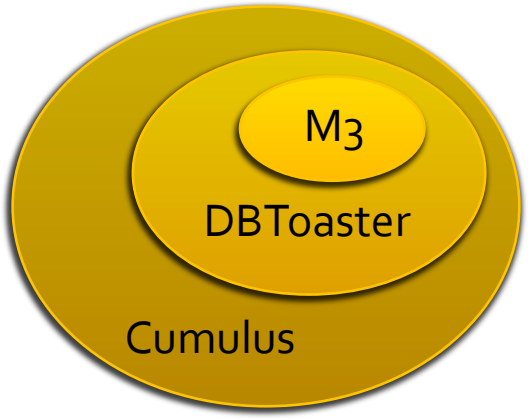


DBToaster: Our Results

- Extremely efficient, lightweight incremental query evaluation
  - Datastructure-oriented, not operator-oriented
  - For many practical queries, we evaluate queries asymptotically faster than any non-incremental algorithm
    - e.g. 2-way join & aggregate query in linear time!
  - Embarrassingly parallelizable
- Novel algorithm to compile SQL to this evaluation model
  - extends the class of queries that can be processed incrementally

Talk Outline

- Introduction and background
- A simple intermediate language: M3
  - Can be evaluated in constant time
  - Extremely parallelizable
- Compiling SQL aggregate queries to M3
  - Aggressive recursive compilation
  - Experimental evaluation against commercial DBMS
- Parallel evaluation of M3 programs
  - Large-scale main-memory OLAP M3 runtime
- Next steps: bulk processing



M3: Map Maintenance Intermediate Language

$$
\begin{aligned}
M3 &::= \text{on } event\ R(\vec{x}\vec{y})\ \{stmt^*\} \\
event &::= \text{insert} \mid \text{delete} \\
stmt &::= m[\vec{x}] \pm\!= expr \\
&\quad \mid \texttt{foreach}\ \vec{z}\ \texttt{in}\ m[\vec{x}\vec{z}]\ \texttt{do}\ m[\vec{x}\vec{z}] \pm\!= expr \\
expr &::= v \mid m[\vec{v}] \\
&\quad \mid expr \times expr \mid expr + expr \mid expr?\ expr : 0
\end{aligned}
$$

- Trigger statements
- Slice update statements
- Arithmetic map expressions

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

M3: Map Maintenance Intermediate Language

$$M3 ::= \text{on } event\ R(\vec{xy})\ \{stmt^*\}$$
$$event ::= \text{insert} \mid \text{delete}$$
$$stmt ::= m[\vec{x}] \pm\!= expr$$
$$\mid \texttt{foreach}\ \vec{z}\ \texttt{in}\ m[\vec{xz}]\ \texttt{do}\ m[\vec{xz}] \pm\!= expr$$
$$expr ::= v \mid m[\vec{v}]$$
$$\mid expr \times expr \mid expr + expr \mid expr?\ expr : 0$$

- Trigger statements
- Slice update statements
- Arithmetic map expressions

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Auxiliary maps:

q1: # customers per nation

q2: # customer, nation pairs

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC

Map: q1

nation

Map: q2

cid	nation

Map: q

cid

Trace: q

Δq

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC
1	+	1,US

Map: q1

nation

Map: q2

cid	nation

Map: q

cid	
1	1

Trace: q

Δq
q[1] += 1

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC
1	+	1,US

Map: q1

nation	
US	1

Map: q2

cid	nation	
1	US	1

Map: q

cid	
1	1

Trace: q

Δq
q[1] += 1

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC
1	+	1,US

Map: q1

nation	
US	1

Map: q2

cid	nation	
1	US	1

Map: q

cid	
1	1

Trace: q

Δq
q[1] += 1

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC
1	+	1,US
2	+	2,UK

Map: q1

nation	
US	1

Map: q2

cid	nation	
1	US	1

Map: q

cid	
1	1
2	**1**

Trace: q

Δq
q[1] += 1
q[2] += 1

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC
1	+	1,US
2	+	2,UK
3	+	3.UK

Map: q1

nation	
US	1
UK	1

Map: q2

cid	nation	
1	US	1
2	UK	1

Map: q

cid	
1	1
2	2
3	**2**

Trace: q

Δq
q[1] += 1
q[2] += 1
q[3] += q1[UK] + 1 q[2] += q2[2,UK]

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC
1	+	1,US
2	+	2,UK
3	+	3.UK

Map: q1

nation	
US	1
UK	1

Map: q2

cid	nation	
1	US	1
2	UK	1

Map: q

cid	
1	1
2	2
3	**2**

Trace: q

Δq
q[1] += 1
q[2] += 1
q[3] += q1[UK] + 1 q[2] += q2[2,UK]

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC
1	+	1,US
2	+	2,UK
3	+	3.UK
4	+	4,US

Map: q1

nation	
US	1
UK	2

Map: q2

cid	nation	
1	US	1
2	UK	1
3	UK	1

Map: q

cid	
1	2
2	2
3	2
4	**2**

Trace: q

Δq
q[1] += 1
q[2] += 1
q[3] += q1[UK] + 1 q[2] += q2[2,UK]
q[4] += q1[US] + 1 q[1] += q2[1,US]

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC
1	+	1,US
2	+	2,UK
3	+	3.UK
4	+	4,US
5	-	3,UK

Map: q1

nation	
US	2
UK	2

Map: q2

cid	nation	
1	US	1
2	UK	1
3	UK	1
4	US	1

Map: q

cid	
1	2
2	1
3	0
4	2

Trace: q

Δq
q[1] += 1
q[2] += 1
q[3] += q1[UK] + 1 q[2] += q2[2,UK]
q[4] += q1[US] + 1 q[1] += q2[1,US]
q[3] -= q1[UK] - 1 q[2] -= q2[2,UK]

M3: Evaluation Example

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

Trigger

t		ΔC
1	+	1,US
2	+	2,UK
3	+	3.UK
4	+	4,US
5	-	3,UK
6	+	3,US

Map: q1

nation	
US	2
UK	1

Map: q2

cid	nation	
1	US	1
2	UK	1
3	UK	0
4	US	1

Map: q

cid	
1	3
2	1
3	3
4	3

Trace: q

Δq
q[1] += 1
q[2] += 1
q[3] += q1[UK] + 1 q[2] += q2[2,UK]
q[4] += q1[US] + 1 q[1] += q2[1,US]
q[3] -= q1[UK] - 1 q[2] -= q2[2,UK]
q[3] += q1[US] + 1 q[1] += q2[1,US] q[4] += q2[4,US]

M3: Map Maintenance Intermediate Language

$$
\begin{aligned}
M3 &::= \text{on } event\ R(\vec{x}\vec{y})\ \{stmt^*\} \\
event &::= \text{insert} \mid \text{delete} \\
stmt &::= m[\vec{x}] \pm\!= expr \\
&\quad \mid \texttt{foreach}\ \vec{z}\ \texttt{in}\ m[\vec{x}\vec{z}]\ \texttt{do}\ m[\vec{x}\vec{z}] \pm\!= expr \\
expr &::= v \mid m[\vec{v}] \\
&\quad \mid expr \times expr \mid expr + expr \mid expr?\ expr : 0
\end{aligned}
$$

```
on insert into Customer(cid,nation) {
  q[cid] += q1[nation];
  foreach cid2 do
    q[cid2] += q2[cid2,nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid,nation] += 1;
}
```

- No operators, only map updates
- Constant time expressions (RHS), i.e. maintain each aggregate value in constant time!
- Other structural characteristics:
  - Topologically sorted w.r.t map updates (statements use "old" data)
  - "Atomic" triggers

Query Compilation

- DBToaster compiles SQL group-by aggregate queries
  - Queries specified on an arbitrary database as with views
- DBToaster uses ***aggressive recursive compilation***
  - Compute deltas of queries as in incremental view maintenance.
  - But: the delta is again a query: so maintain its view incrementally by delta processing.
  - Maintain the delta to the delta incrementally, etc.
  - Delta processing rules are an analogy to differentials in calculus, but applied to relational calculus.
- Related work: System R [Chamberlin et al. '81], Genesis [Batory et al. '88]

Example Schema (~TPC-H)

```
Order(OrderKey, CustomerKey, Date, ExchangeRate)
O(OK, CK, D, XCH)

LineItem(OrderKey, PartKey, Price)
LI(OK, PK, P)

q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;
```

Example Schema (~TPC-H)

```
Order(OrderKey, CustomerKey, Date, ExchangeRate)
O(OK, CK, D, XCH)

LineItem(OrderKey, PartKey, Price)
LI(OK, PK, P)

q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;
```

```
select   sum((A.price - B.price)
         * (A.time - B.time))
         as holds
from     Bids B, Asks A
where    B.broker_id = A.broker_id;
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] +=

      select sum(LI.P * O.XCH)
      from {<xOK, xCK, xD, xXCH>} O, LineItem LI
      where O.OK = LI.OK;

+LI(yOK, yPK, yP)       q[] += ...
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] +=

      select sum(LI.P *  xXCH)
      from LineItem LI
      where  xOK = LI.OK;

+LI(yOK, yPK, yP)       q[] += ...
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] += xXCH *

      select sum(LI.P)
      from LineItem LI
      where  xOK = LI.OK;
                                  } qO[xOK]

+LI(yOK, yPK, yP)       q[] += ...
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
                       foreach xOK: qO[xOK] =
      select sum(LI.P)
      from LineItem LI
      where  xOK = LI.OK;

+LI(yOK, yPK, yP)      q[] += ...
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)       foreach xOK: qO[xOK] +=
      select sum(LI.P)
      from {<yOK, yPK, yP>} LI
      where  xOK = LI.OK;

+LI(yOK, yPK, yP)       q[] += ...
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)       foreach xOK: qO[xOK] +=
      select yP

      where  xOK = yOK;

+LI(yOK, yPK, yP)       q[] += ...
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;
```

```
+O(xOK, xCK, xD, xXCH)  q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)       qO[yOK] += yP;
```

```
+LI(yOK, yPK, yP)       q[] += ...
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)      qO[yOK] += yP;
+LI(yOK, yPK, yP)      q[] +=

      select sum(LI.P * O.XCH)
      from Order O, {<yOK, yPK, yP>} LI
      where O.OK = LI.OK;
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)      qO[yOK] += yP;
+LI(yOK, yPK, yP)      q[] +=

      select sum(  yP * O.XCH)
      from Order O
      where O.OK =   yOK;
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)      qO[yOK] += yP;
+LI(yOK, yPK, yP)      q[] += yP *

      select sum(       O.XCH)
      from Order O
      where O.OK =   yOK;
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;
```

```
+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)      qO[yOK] += yP;
+LI(yOK, yPK, yP)      q[] += yP * qLI[yOK];
```

```
      select sum(       O.XCH)
      from Order O
      where O.OK =   yOK;
```

qLI[yOK]

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;
```

```
+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)      qO[yOK] += yP;
+LI(yOK, yPK, yP)      q[] += yP * qLI[yOK];
+O(xOK, xCK, xD, xXCH) foreach yOK: qLI[yOK] +=
      select sum(        O.XCH)
      from {<xOK, xCK, xD, xXCH>} O
      where O.OK =   yOK;
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;
```

```
+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)      qO[yOK] += yP;
+LI(yOK, yPK, yP)      q[] += yP * qLI[yOK];
+O(xOK, xCK, xD, xXCH) foreach yOK: qLI[yOK] +=
      select             xXCH

      where  xOK =   yOK;
```

Compilation Example

```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)      qO[yOK] += yP;
+LI(yOK, yPK, yP)      q[] += yP * qLI[yOK];
+O(xOK, xCK, xD, xXCH) qLI[xOK] += xXCH;
```

The triggers for incrementally maintaining all the maps run in constant time!

Compilation Algorithm

Given a query Q,

1. Compute delta of Q for tuple insertion/deletion.
2. Simplify delta query.
3. Extract aggregate queries.
4. Recursively compile extracted aggregate subqueries.

```
algorithm Compile(map_name: string,
                  map_args: var list, t: term)
outputs an M3 program
begin
for each relation R in the schema, pm in {+, −} do
  trigger_args := turn columns names of R into list
                  of new argument variable names;
  for each t_i in RecMonomials(Δ_{pmR(trigger_args)} t) do
    bound_vars := trigger_args ∪ map_args;
    (t'_i, Θ_i) := ExtractAggregates(
      Simplify(t_i, bound_vars), bound_vars);
    s := SimplifyArgs((foreach map_args do
      map_name[map_args] pm= t'_i), trigger_args);
    if pm='+' then
      output on insert into R(trigger_args) {s};
    else
      output on delete from R(trigger_args) {s};
  Θ := ⋃_i Θ_i; /* eliminates duplicates */
  for each (m[x⃗] ↦ t') in Θ do Compile(m, x⃗, t');
end
```

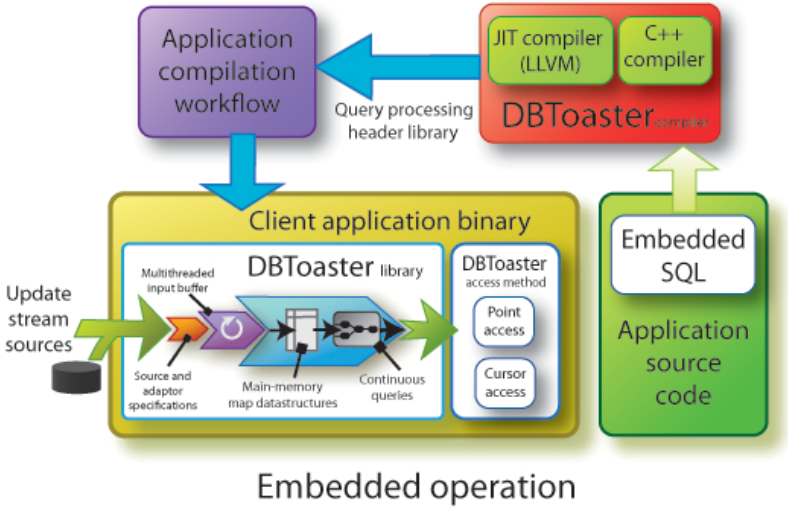
Nested Aggregate Queries

```
select avg(b0.p * b0.v) from B b0
where 0.25 * (select sum(b1.v) from B b1) >
      (select sum(b2.v) from B b2
       where b2.p > b0.p);
```

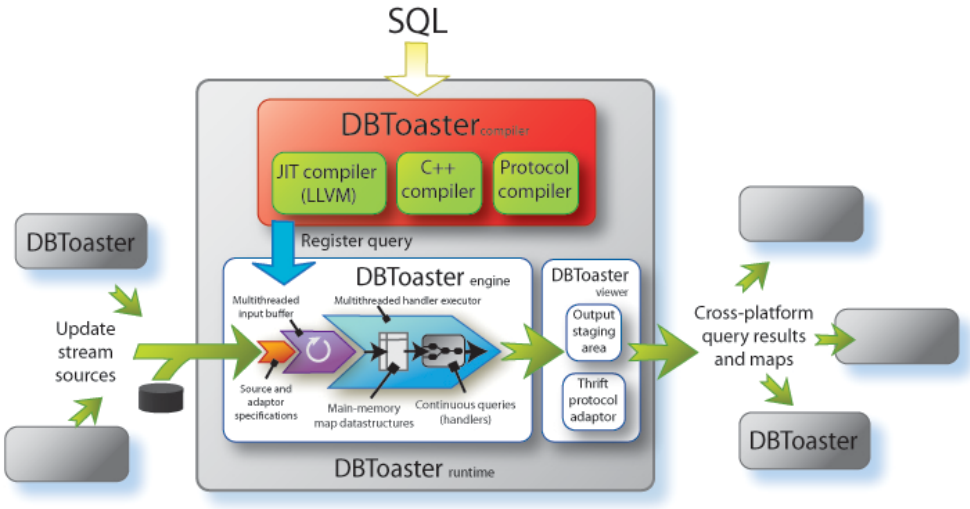
```
+B(xp, xv): q[] += sum_p(
      if (c[p] + Delta c[p] > 0) then Delta a[p] else 0
      + if (c[p] + Delta c[p] > 0) and (c[p] <= 0)
        then a[p] else 0
      - if (c[p] + Delta c[p] <= 0) and (c[p] > 0)
        then a[p] else 0)
```

```
Delta c[p] = 0.25 * xv - (if (xp > p) then xv else 0)
Delta a[p] = if (xp = p) then xp * xv else 0
```

In-Memory Stream Engine



Embedded operation



Standalone operation

- QPs as shared libraries
- Queries run in the same process space as app

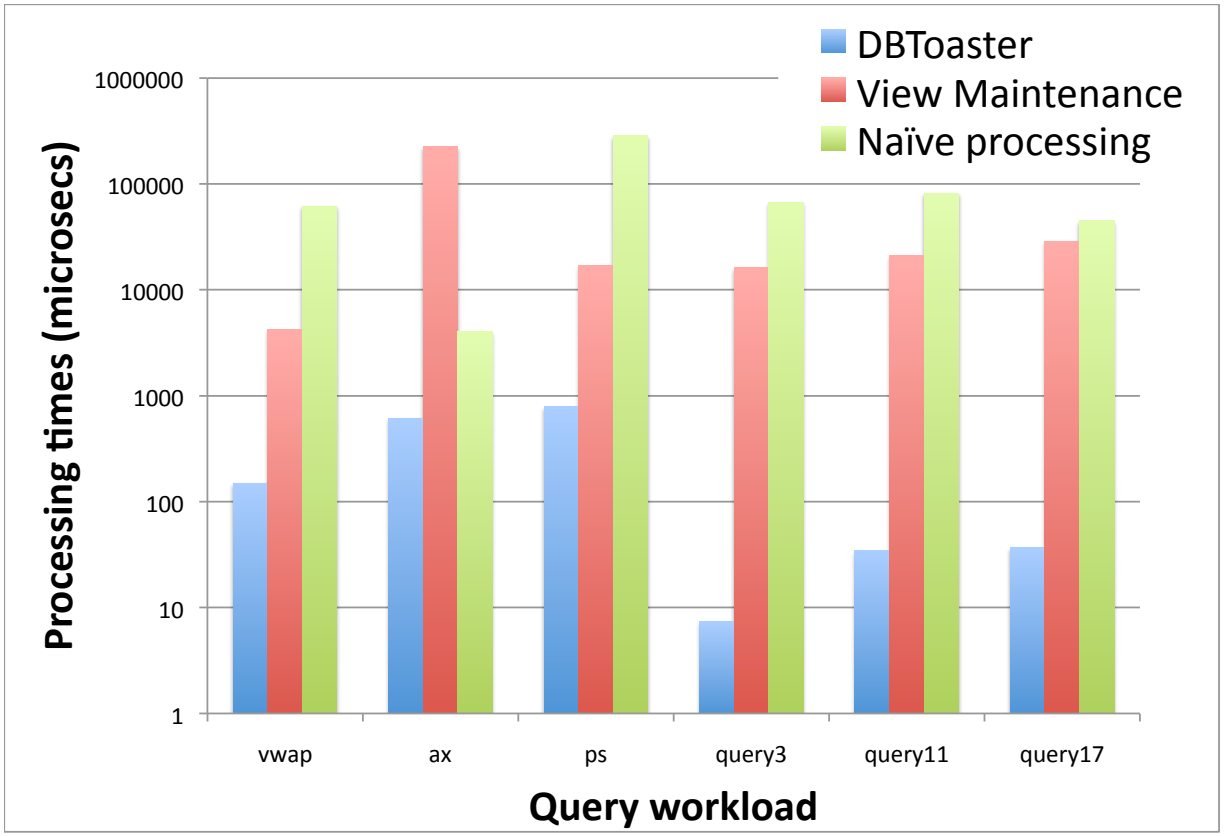
- JIT compilation of queries with LLVM
- Protocol compiler based on Thrift

Experiments: QP Performance

Workload: i) order book queries: vwap, ax, ps. ii) TPC-H: query3, query11, query17

Datasets:
- NASDAQ ITCH, Dec 08-Feb 09
- TPC-H, scale factor 1

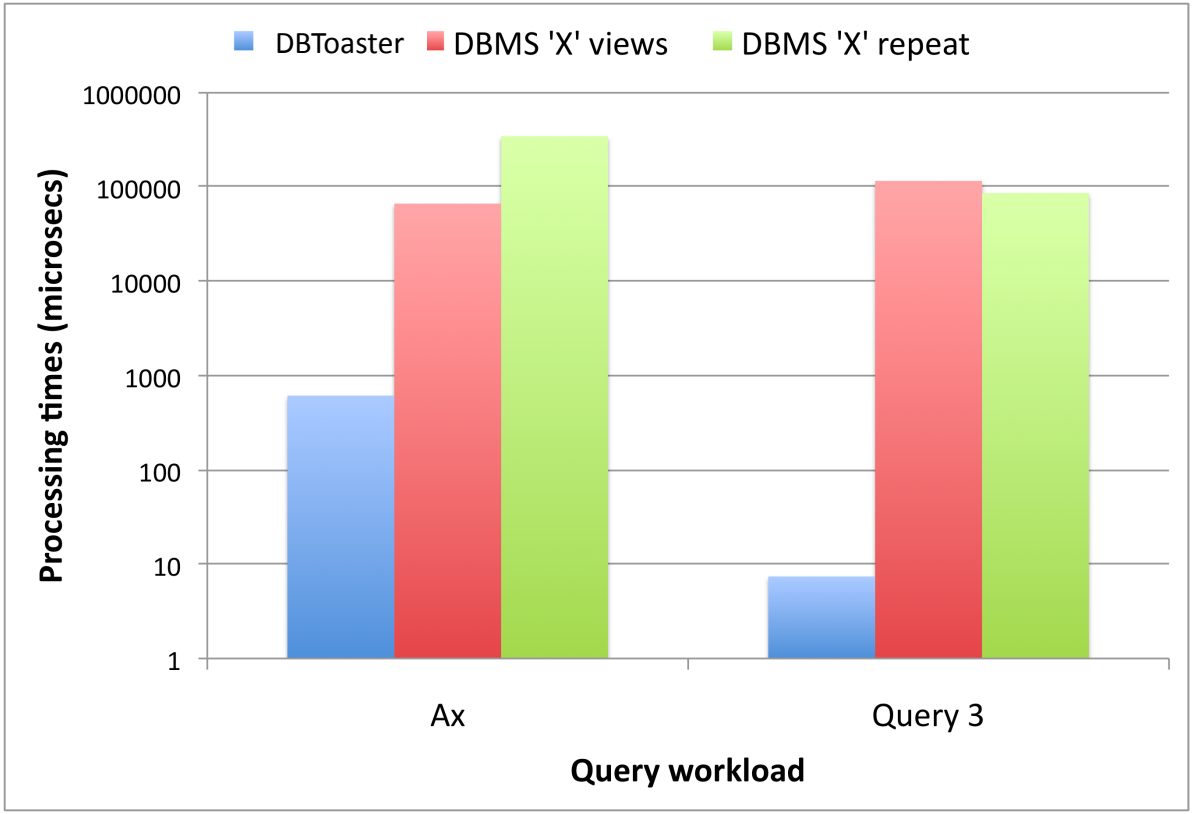
Takeaways:
- 1-2 orders of magnitude speedup on order book queries
- 2.5-4 orders of magnitude speedup on TPC-H
- Order book queries have a more complex nesting structure



Experiments: DBMS comparison

DBToaster vs. commercial DBMS ('X') that supports incremental view maintenance:

- DBMS 'X' views: incremental view maintenance
- DBMS 'X' repeat: from-scratch maintenance
- DBMS 'X' supports maintenance of only two queries incrementally
- DBToaster yields 2.5-4 orders of magnitude speedup



Query Compilation: Takeaways

- DBToaster uses ***aggressive recursive compilation***
  - Datastructure-oriented query evaluation
  - Supports incremental computation of nested aggregate queries
- DBToaster yields 2.5-4 orders of magnitude speedup over a commercial DBMS 'X'
- Now we'll see added benefits of datastructure-based evaluation…

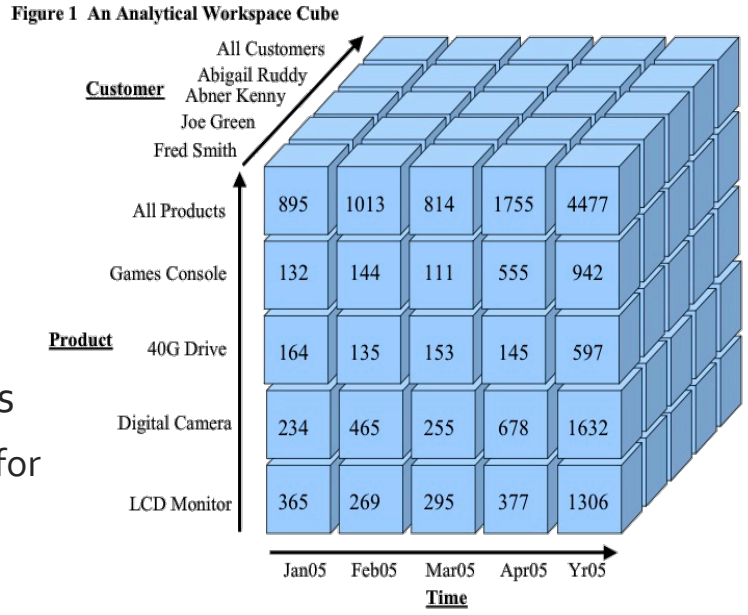
App: Massively Parallel Real-time OLAP

- OLAP cubes: multidimensional group-by aggregates
  - Heavily used for business intelligence
  - Traditionally processed on different DBMS architecture to OLTP
  - Much of the cloud DBMS work targets similar functionality, except the real-time updates aspect

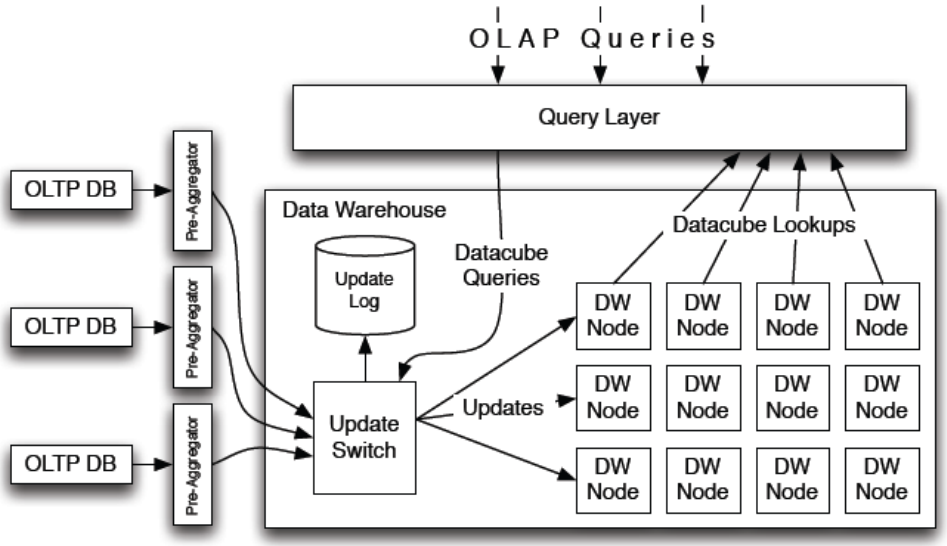

```
select      c.customer_id, month(o.date)
            sum(orders.price)
from        customers c, orders o
where       c.customer_id = o.customer_id
group by    c.customer_id, month(o.date)
```

App: Massively Parallel Real-time OLAP

- OLAP cubes: multidimensional group-by aggregates
  - Heavily used for business intelligence
  - Traditionally processed on different DBMS architecture to OLTP
  - Much of the cloud DBMS work targets similar functionality, except the real-time updates aspect
- Continuous OLAP maintains subset of cube entries
  - We'll use DBToaster to compile aggregate queries for entries, sharing maps
  - Scalability challenges:
    - Large # of dimensions & labels, large maps
    - High-throughput parallel M_3 processing
    - Responsiveness under heavy update rates



Cumulus: a Distributed M3 Engine

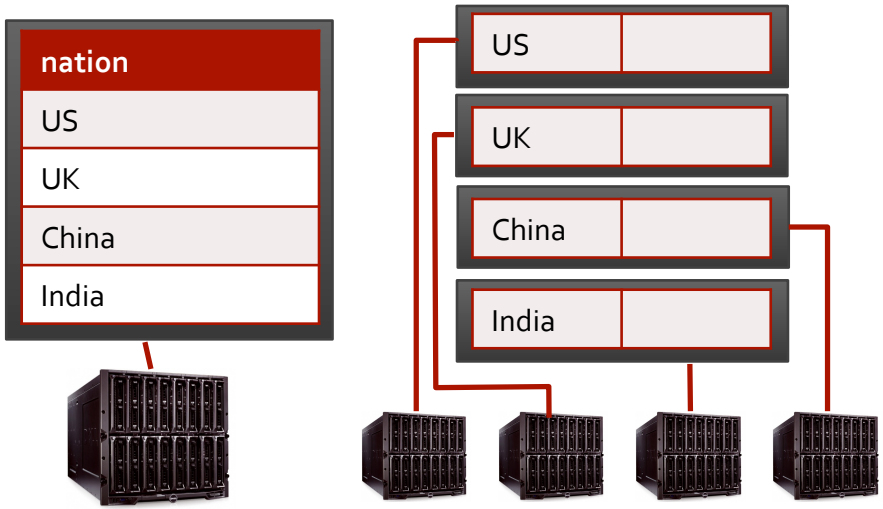


- Cumulus is a distributed shared-nothing interpreter for running DBToaster code, used for an online main-memory OLAP system.
- Exploits that M3 programs admit embarrassingly parallel evaluation
  - Each map value requires only a constant amount of work to update!
  - Insight: evaluate M3 programs in terms of multidimensional map slices

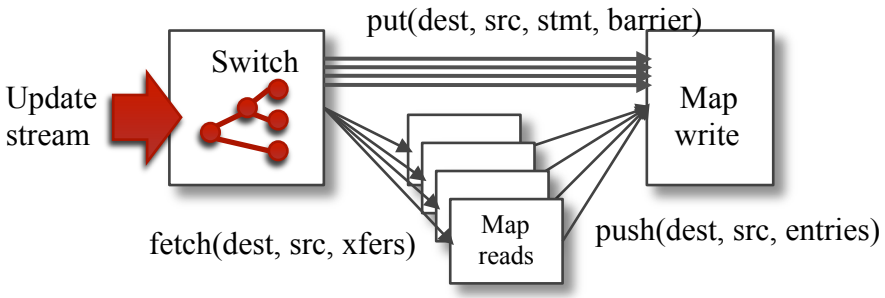
Parallelizing M3 Programs

$$stmt ::= \ldots \mid \texttt{foreach}\ \vec{z}\ \texttt{in}\ m[\vec{x}\vec{z}]\ \texttt{do}\ m[\vec{x}\vec{z}]\ \pm\texttt{=}\ expr$$

- Maintaining individual aggregates



- Message flow & types

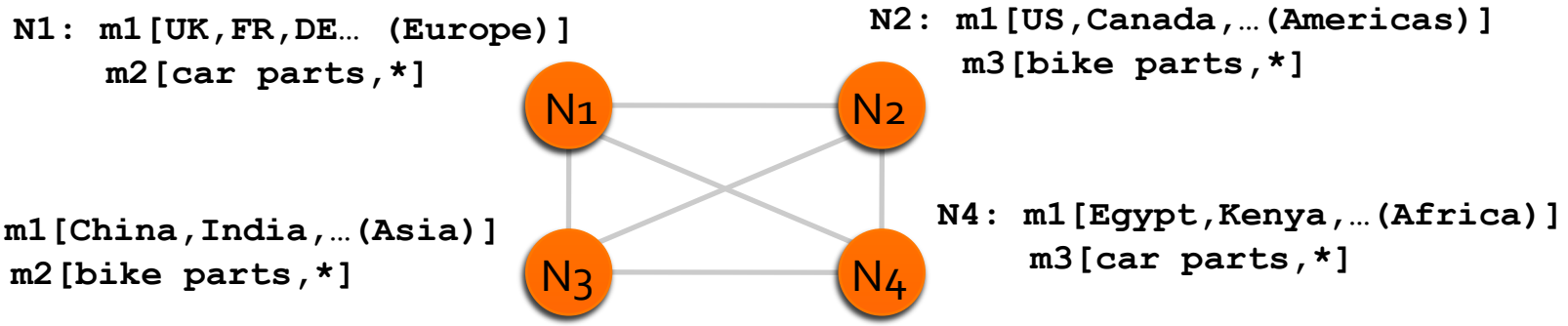


M3 Processing Protocol

DBToaster code snippet:

```
on insert PART(pk,retailprice,desc,…)
{
  …
  foreach nk in m1:
    m1[nk] += retailprice * m2[pk,nk] – m3[pk,nk];
  …
}
```

Map partitions in a cluster:

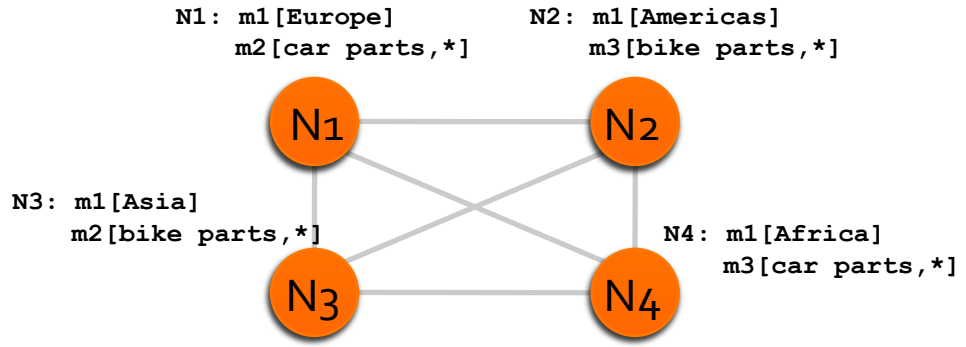


M3 Processing Protocol

DBToaster code snippet:

```
on insert
PART(pk,retailprice,desc,…)
{
  …
  foreach nk in m1:
    m1[nk] += retailprice *
      m2[pk,nk] – m3[pk,nk];
  …
}
```

Map partitions in a cluster:



Event: **+PART(21,$500, 'timing belt')**

Switch

N1

N2

N3

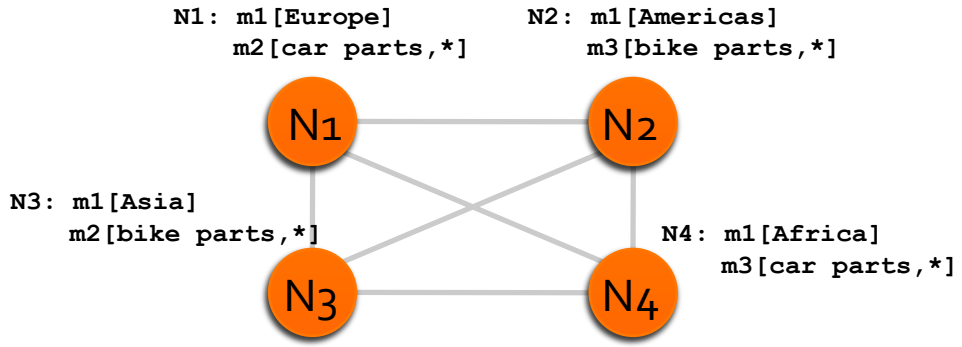
N4

M3 Processing Protocol

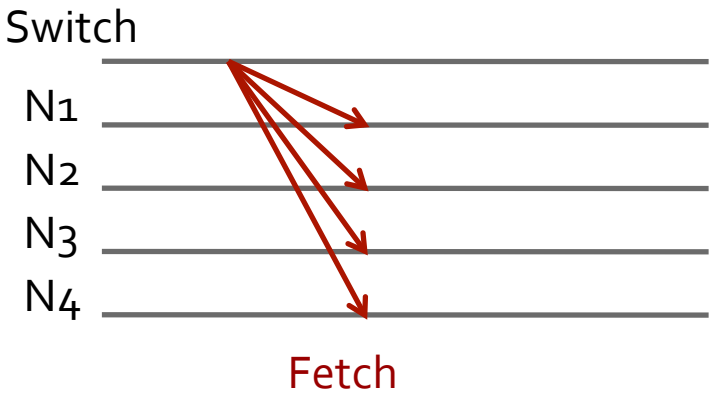
DBToaster code snippet:

```
on insert
PART(pk,retailprice,desc,…)
{
  …
  foreach nk in m1:
    m1[nk] += retailprice *
      m2[pk,nk] – m3[pk,nk];
  …
}
```

Map partitions in a cluster:



Event: **+PART(21,$500, 'timing belt')**



Switch messages:

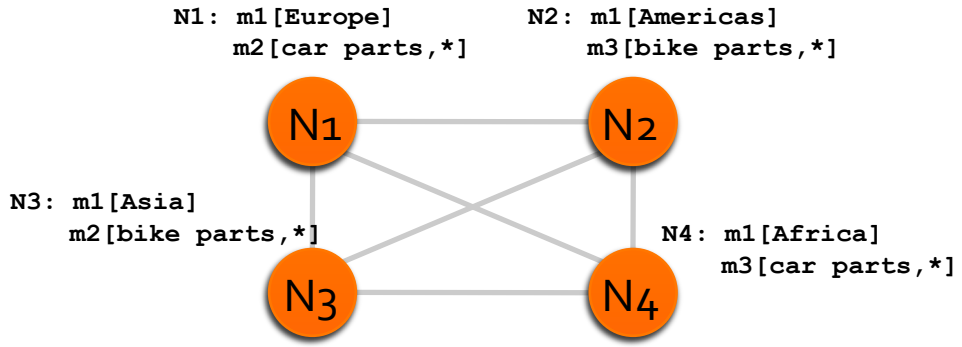
```
FETCH(N1,m2[21,*]=>
      {N1,N2,N3,N4})
FETCH(N4,m3[21,*]=>
      {N1,N2,N3,N4})

PUT({N1,N2,N3,N4},
    m1[x]+=
      m2[21,x]*m3[21,x],
    {N1,N4})
```

M3 Processing Protocol

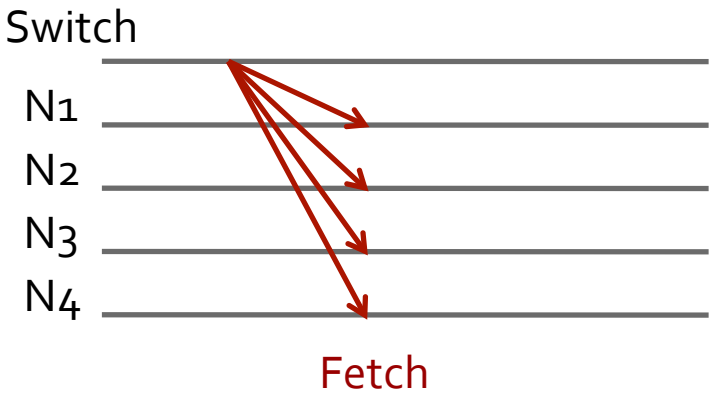
DBToaster code snippet:

```
on insert
PART(pk,retailprice,desc,…)
{
  …
  foreach nk in m1:
    m1[nk] += retailprice *
      m2[pk,nk] – m3[pk,nk];
  …
}
```

Map partitions in a cluster:



Event: **+PART(21,$500, 'timing belt')**



Switch messages:

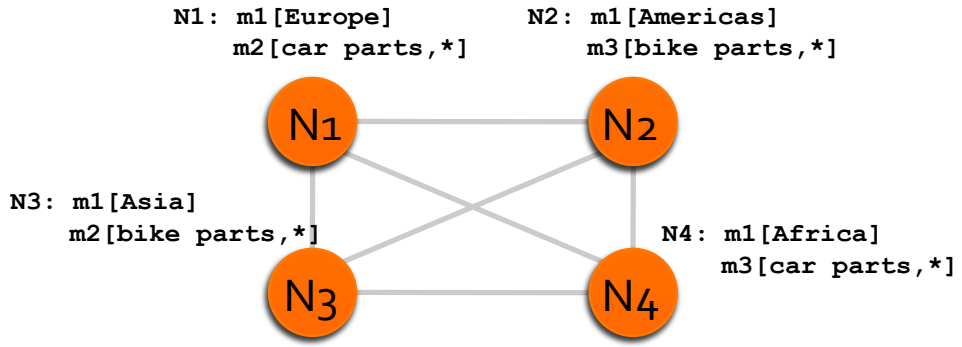
```
FETCH(N1,m2[21,*]=>
      {N1,N2,N3,N4})
FETCH(N4,m3[21,*]=>
      {N1,N2,N3,N4})

PUT({N1,N2,N3,N4},
    m1[x]+=
      m2[21,x]*m3[21,x],
    {N1,N4})
```

M3 Processing Protocol

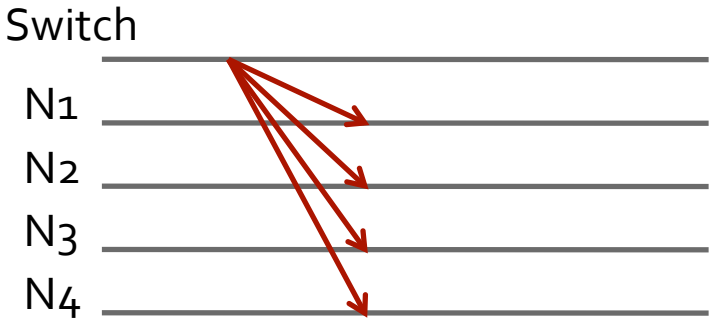
DBToaster code snippet:

```
on insert
PART(pk,retailprice,desc,…)
{
  …
  foreach nk in m1:
    m1[nk] += retailprice *
      m2[pk,nk] – m3[pk,nk];
  …
}
```

Map partitions in a cluster:



Event: **+PART(21,$500, 'timing belt')**



Fetch, put

Switch messages:

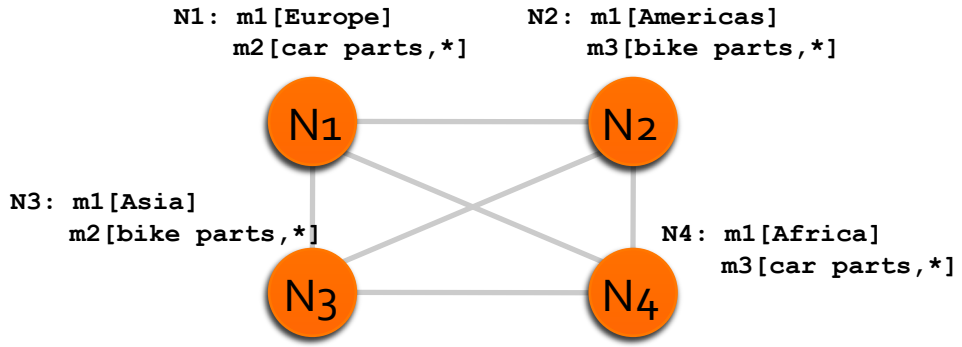
```
FETCH(N1,m2[21,*]=>
      {N1,N2,N3,N4})
FETCH(N4,m3[21,*]=>
      {N1,N2,N3,N4})

PUT([N1,N2,N3,N4],
    m1[x]+=
      m2[21,x]*m3[21,x],
    {N1,N4})
```

M3 Processing Protocol

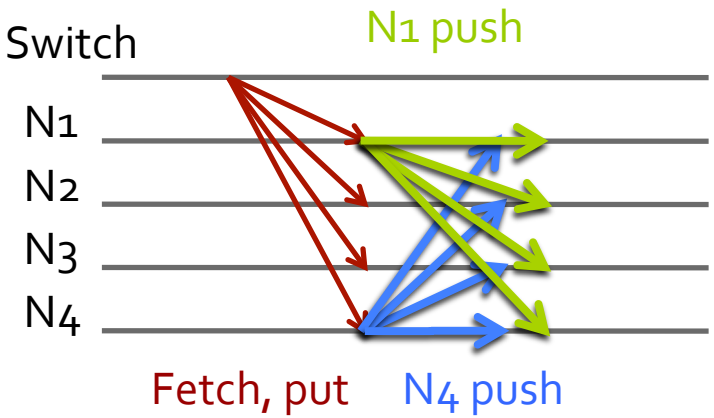
DBToaster code snippet:

```
on insert
PART(pk,retailprice,desc,…)
{
  …
  foreach nk in m1:
    m1[nk] += retailprice *
      m2[pk,nk] – m3[pk,nk];
  …
}
```

Map partitions in a cluster:



Event: **+PART(21,$500, 'timing belt')**



Switch messages:

```
FETCH(N1,m2[21,*]=>
      {N1,N2,N3,N4})
FETCH(N4,m3[21,*]=>
      {N1,N2,N3,N4})

PUT({N1,N2,N3,N4},
    m1[x]+=
      m2[21,x]*m3[21,x],
    {N1,N4})
```

Node messages:

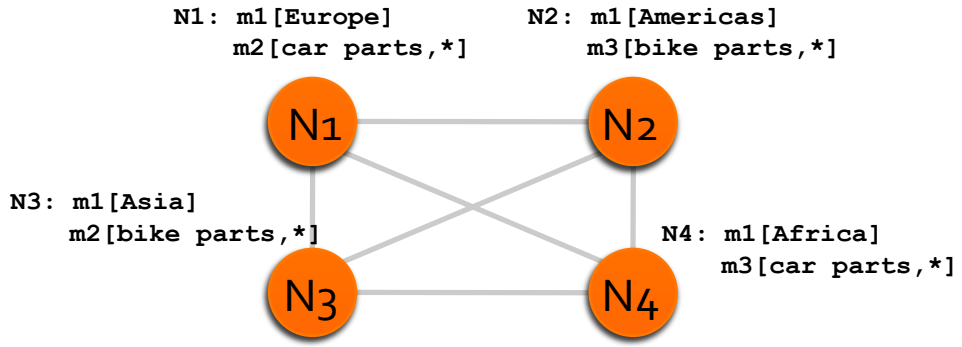
```
PUSH(N1,N1,m2[21,Europe]   =>N1)
PUSH(N1,N2,m2[21,Americas]=>N2)
PUSH(N1,N3,m2[21,Asia]     =>N3)
PUSH(N1,N4,m2[21,Africa]   =>N4)

PUSH(N4,N1,m3[21,Europe]   =>N1)
PUSH(N4,N2,m3[21,Americas]=>N2)
PUSH(N4,N3,m3[21,Asia]     =>N3)
PUSH(N4,N4,m3[21,Africa]   =>N4)
```

M3 Processing Protocol

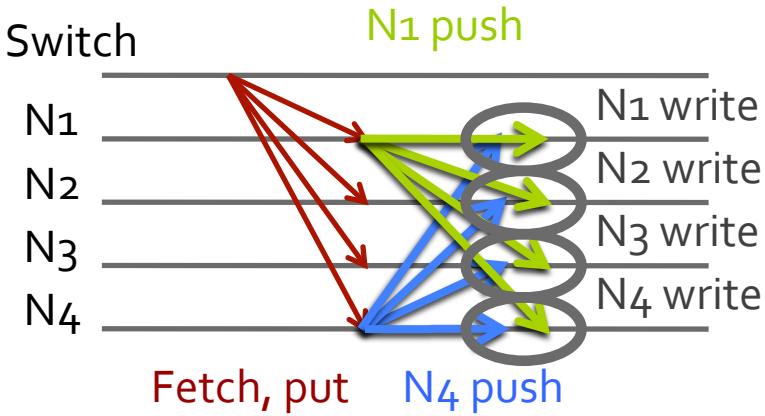
DBToaster code snippet:

```
on insert
PART(pk,retailprice,desc,…)
{
  …
  foreach nk in m1:
    m1[nk] += retailprice *
      m2[pk,nk] – m3[pk,nk];
  …
}
```

Map partitions in a cluster:



Event: **+PART(21,$500, 'timing belt')**



Switch messages:

```
FETCH(N1,m2[21,*]=>
      {N1,N2,N3,N4})
FETCH(N4,m3[21,*]=>
      {N1,N2,N3,N4})

PUT({N1,N2,N3,N4},
    m1[x]+=
      m2[21,x]*m3[21,x],
    {N1,N4})
```

Node messages:

```
PUSH(N1,N1,m2[21,Europe]   =>N1)
PUSH(N1,N2,m2[21,Americas]=>N2)
PUSH(N1,N3,m2[21,Asia]     =>N3)
PUSH(N1,N4,m2[21,Africa]   =>N4)

PUSH(N4,N1,m3[21,Europe]   =>N1)
PUSH(N4,N2,m3[21,Americas]=>N2)
PUSH(N4,N3,m3[21,Asia]     =>N3)
PUSH(N4,N4,m3[21,Africa]   =>N4)
```

Cumulus Status and Results

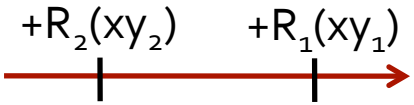
- JRuby w/ BDB backend
- TPC-H based queries:
  - Key-foreign key joins
- Running on 40-node cluster, nodes: 2.2Ghz, 16GB Ram
- Performance (sustainable rate):
  - 725 updates/sec, 40 node
  - Available memory scales linearly
- Ongoing:
  - Automatic partitioning
  - TPC-H non-key
  - EC2 deployment

```
select   s_nationkey,
         sum((p_retailprice - ps_supplycost) * ps_availqty)
from  part p, partsupp ps, supplier s
where  p_partkey = ps_partkey AND s_suppkey = ps_suppkey
group by  s_nationkey;
```

```
select l_partkey, sum(l_quantity)
from customer c, supplier s, orders o, lineitem l
where c_nationkey = s_nationkey
and   s_suppkey   = l_suppkey
and   c_custkey   = o_orderkey
and   o_orderkey  = l_orderkey
group by l_partkey;
```

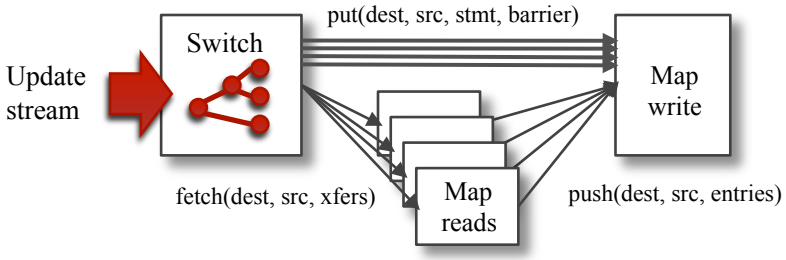
Cumulus: a push-based protocol

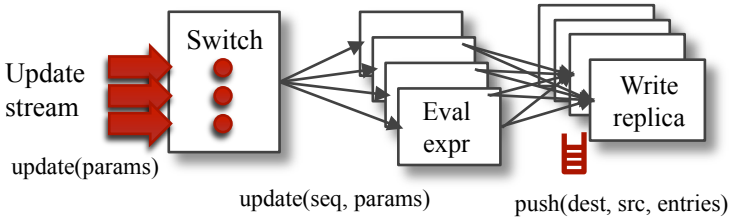
- Cumulus relies on ordered updates at nodes



- Observation: we can defer combining deltas
  - Tolerates out-of-orderness
  - Leads to fully push-based, replicated evaluation

$$+R_1(\vec{x}\vec{y_1}) : m[\vec{x}] += y_1$$
$$+R_2(\vec{x}\vec{y_2}) : m[\vec{x}] += y_2$$
$$\text{defer } +R_1\text{: } m[\vec{x}] += \underbrace{\underbrace{y_1}_{+R1.expr} + y_2}_{eval(+R1;+R2)}$$





- Key takeaway: we break the atomicity of DBToaster's triggers

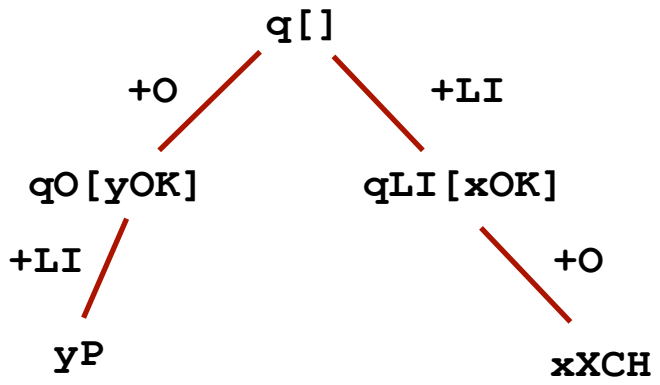
Cumulus Summary

- Massive-scale shared-nothing M3 interpreter
  - M3 processing protocol
  - Map slice based M3 evaluation
- Fun "systems-building" challenges ahead:
  - Automatic map partitioning and deployment
  - Exploiting "lazy" evaluation and eventually correct results (… think online aggregation)
  - Bootstrapping large maps… *bulk query processing*

DBToaster: Next Directions

- Bulk vs. incremental query processing

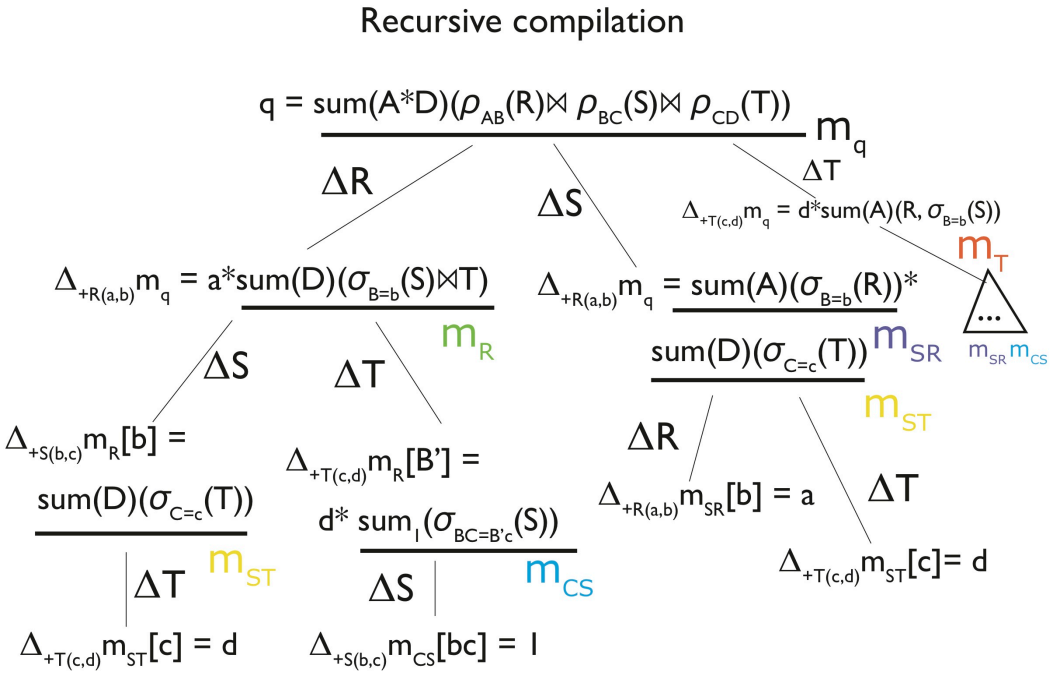
```
q[] = select sum(LI.P * O.XCH)
      from Order O, LineItem LI
      where O.OK = LI.OK;

+O(xOK, xCK, xD, xXCH) q[] += xXCH * qO[xOK];
+LI(yOK, yPK, yP)      qO[yOK] += yP;

+LI(yOK, yPK, yP)      q[] += yP * qLI[yOK];
+O(xOK, xCK, xD, xXCH) qLI[xOK] += xXCH;
```



DBToaster: Next Directions

- Bulk vs. incremental query processing

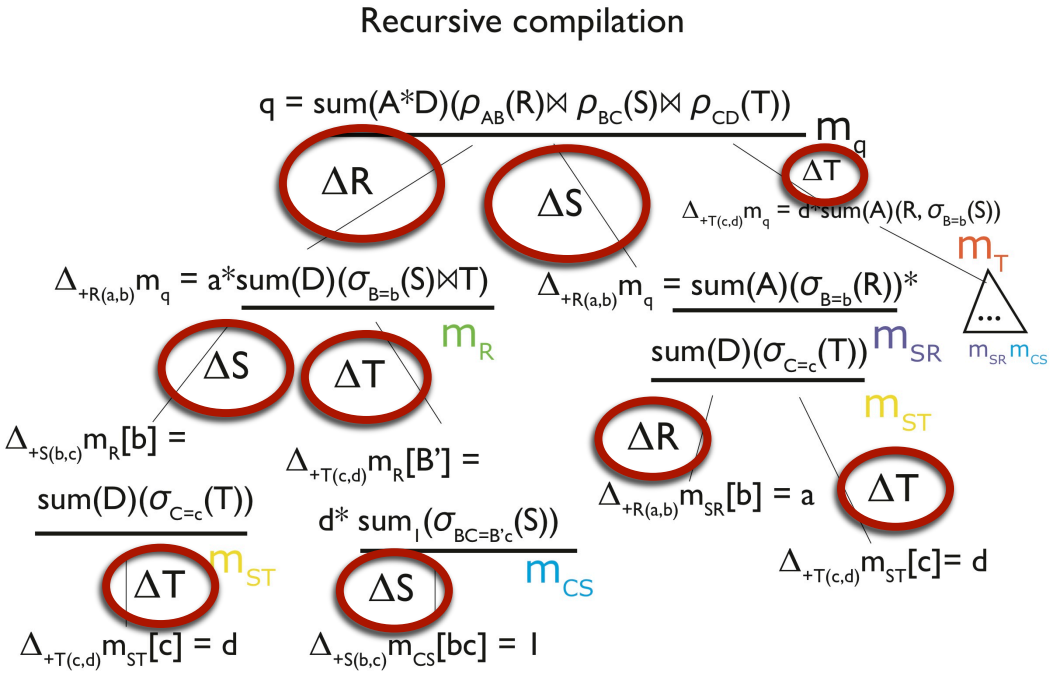
select sum(a*d) from R,S,T where R.b=S.b and S.c=T.c



DBToaster: Next Directions

- Bulk vs. incremental query processing
  - No need to compute all paths, just one

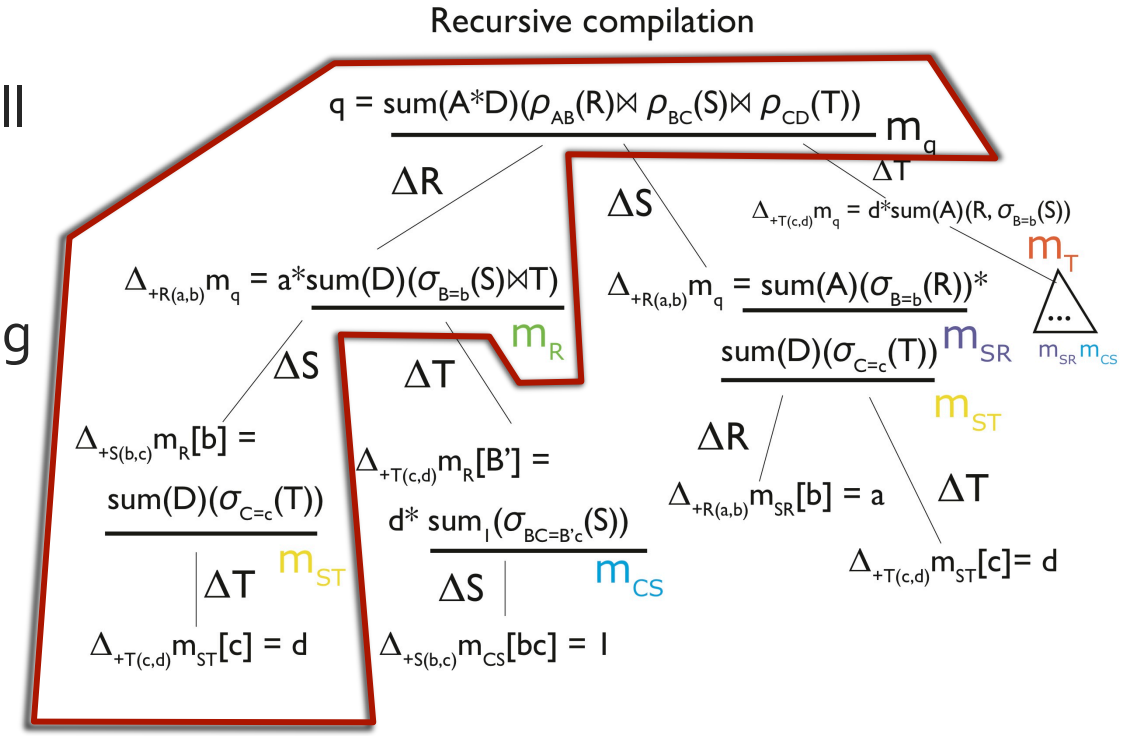
select sum(a*d) from R,S,T where R.b=S.b and S.c=T.c



DBToaster: Next Directions

- Bulk vs. incremental query processing
  - No need to compute all paths, just one
  - Picking the right one seems like join ordering
    - Different cost model because of aggregate distributivity

select sum(a*d) from R,S,T where R.b=S.b and S.c=T.c



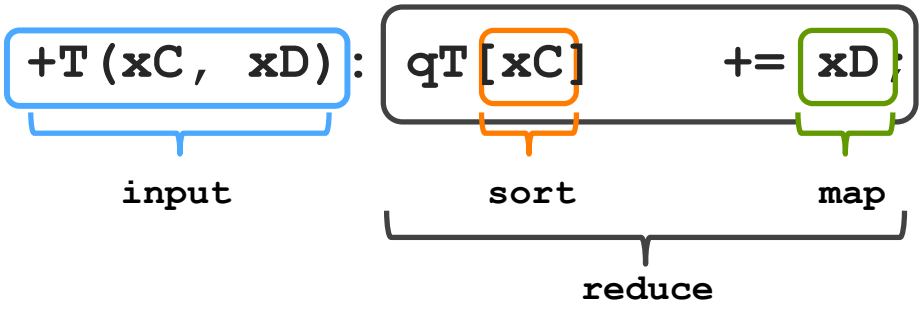
+R; +S; +T

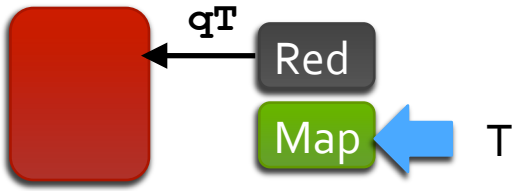
Bulk Tuple Processing

```
q[] = select sum(R.a * T.d)
      from R, S, T
      where R.b = S.b and S.c = T.c;

+R(xA, xB) q[]       += xA * qST[xB];
+S(xB, xC) qST[xB]   += qT[xC];
+T(xC, xD) qT[xC]    += xD;
```

Bulk Tuple Processing on Hadoop

```
q[] = select sum(R.a * T.d)
      from R, S, T
      where R.b = S.b and S.c = T.c;

+R(xA, xB): q[]       += xA * qST[xB];


+S(xB, xC): qST[xB]   += qT[xC];
```

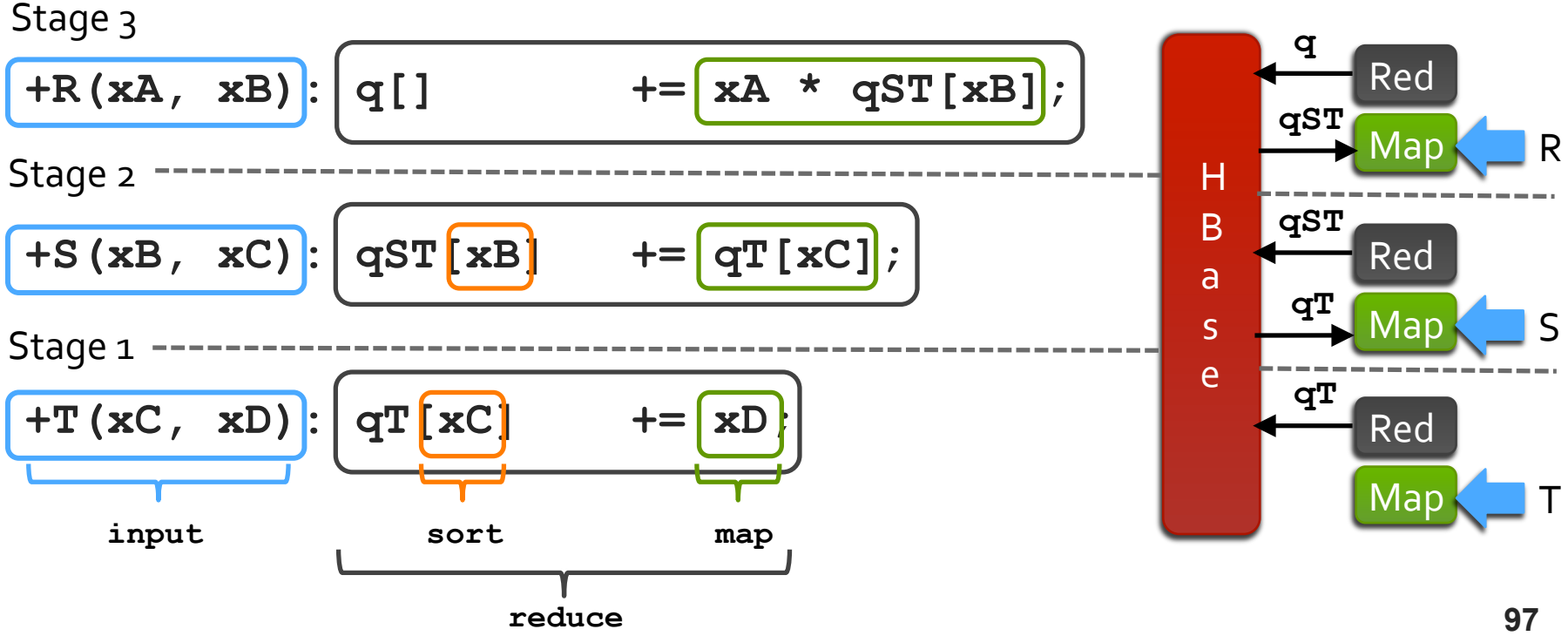




HBase
(… or HDFS)

Bulk Tuple Processing on Hadoop

```
q[] = select sum(R.a * T.d)
      from R, S, T
      where R.b = S.b and S.c = T.c;
```

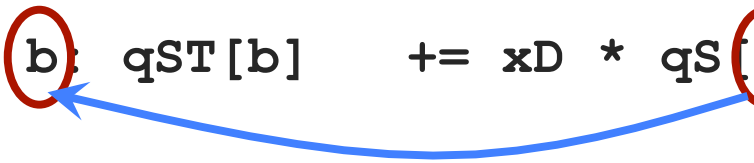


Bulk Tuple Processing: Map Slices

```
q[] = select sum(R.a * T.d)
      from R, S, T
      where R.b = S.b and S.c = T.c;

+R(xA, xB):              q[]      += xA * qST[xB];

+T(xC, xD): foreach b: qST[b]   += xD * qS[b,xC];

+S(xB, xC):              qT[xC]   += xD;
```

Bulk Tuple Processing: Map Slices

```
q[] = select sum(R.a * T.d)
      from R, S, T
      where R.b = S.b and S.c = T.c;

+R(xA, xB):                q[]      += xA * qST[xB];

+T(xC, xD): foreach b:     qST[b]   += xD * qS[b,xC];

+S(xB, xC):                qT[xC]   += xD;
```



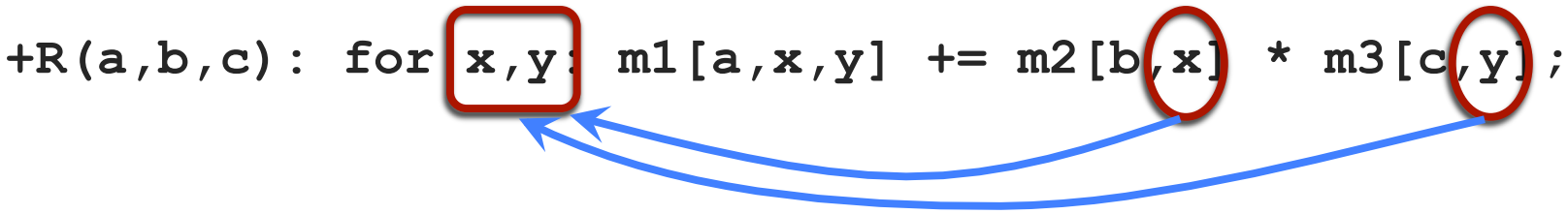
Bulk Tuple Processing: Map Slices

```
q[] = select sum(R.a * T.d)
      from R, S, T
      where R.b = S.b and S.c = T.c;

+R(xA, xB):                          q[]    += xA * qST[xB];

+T(xC, xD): for b in qS[*,xC]:       qST[b] += xD * qS[b,xC];
                    slice

+S(xB, xC):                          qT[xC] += xD;
```

Bulk Tuple Processing: Multiple Maps

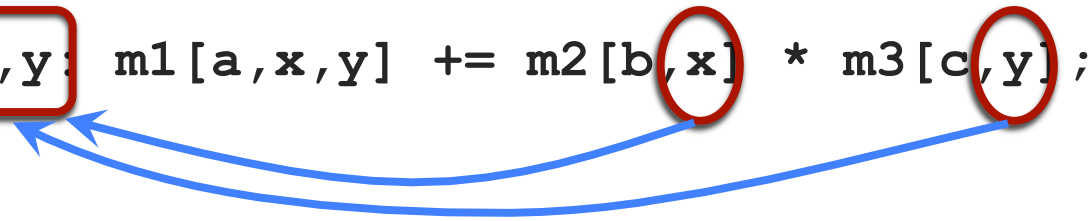
```
+R(a,b,c): for x,y: m1[a,x,y] += m2[b,x] * m3[c,y];
```

Bulk Tuple Processing: Multiple Maps

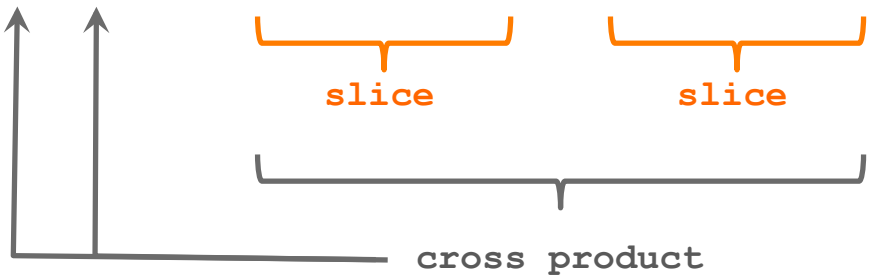


Bulk Tuple Processing: Multiple Maps

```
+R(a,b,c): for x,y: m1[a,x,y] += m2[b,x] * m3[c,y];
```



```
+R(a,b,c): m1[a,*1,*2] += m2[b,*1] * m3[c,*2];
```

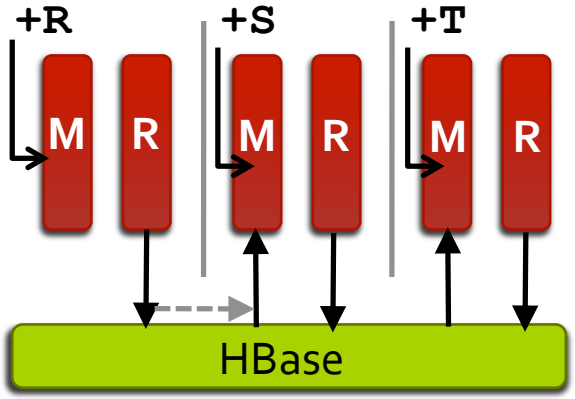


Bulk Tuple Processing: Summary

- Bulk processing with Hadoop & HBase runtime:

$+R(a,b,c)$: foreach x, y in m_1: $m_1[a,x,y]$ += $m_2[x,b] * m_3[y,c]$

input — partition (sort) — product (map) — aggregate (reduce)



  - M3: Product-Partition-Aggregate op
    - … or a key-value/aggregate "language"
- Working on:
  - Cost model for input, and intermediate result partitioning for M3 sequences
  - Whole program optimization of data placement & partitioning

DBToaster: Next Directions

- Query processing & computational foundations
  - Adaptivity spectrum: trade off incremental vs. from-scratch QP
    - e.g., "external" maps, from Hadoop jobs
  - Exploiting schema information (foreign-keys, constraints etc.)
  - Generalized incremental computation (recursion, collections etc.)
  - M3 as a direct user language
- Compiling whole DBMS, incorporating lightweight
  - Concurrency controller and scheduler
  - Optimizer and alerter
  - Storage layout

DBToaster: Conclusions

- Extremely simple evaluation language: M3
  - Query "plans" in terms of data structures, not operators
  - Extremely easy to build runtimes, allowing developers to focus on architecture & context
    - Standalone stream engine, online MPP engine, bulk MPP engine
- Query compilation: DBToaster
  - Novel recursive compilation technique
  - Revisits foundations for incremental processing
  - Natural mechanism to generate ***lightweight*** query engines

A Summary of Yanif's Projects

- DBToaster: lightweight incremental QP for update-intensive apps
- Dissertation work: Pulse, model-based stream processing
  - Ongoing work: declarative model-based databases (differential eqns., time series, frequency domain models, etc.)
- Distributed data management:
  - Borealis: distributed stream processing engine
  - SAND: query processing over global-scale networks
  - SenseWeb: network-efficient indexing in sensor web portals
  - XPORT: extensible overlays for data collection and dissemination

Thank you!

Acknowledgements:

Christoph Koch

Oliver Kennedy

Additional Slides

Formal Framework

- Formal underpinnings:
  - Ring structure over relations
    - Ring defines + as union, x as natural join
    - Captures distributivity, associativity, and deltas
  - Aggregation calculus
    - Connects two rings, one of rational numbers, and another of the relational calculus

$$R : \mathrm{dom}(R) \to \mathbb{Z}$$

R	A	B		
	1		$\mapsto$	-1
	2	3	$\mapsto$	2

S	C		
	5	$\mapsto$	2

T	B	C		
	3	5	$\mapsto$	1
	4	6	$\mapsto$	-3

$$R \cup S \;:\; \vec{x} \mapsto \big(R(\vec{x}) + S(\vec{x})\big)$$

$$R \bowtie S \;:\; \vec{x} \mapsto \sum_{\{\vec{x}\}=\{\vec{a}\}\bowtie^{\{\}}\{\vec{b}\}} R(\vec{a}) * S(\vec{b})$$

$$(-R) \;:\; \vec{x} \mapsto (-R(\vec{x}))$$

Multiset relations

$$\begin{aligned}
\Delta(\alpha+\beta) &:= ((\Delta\alpha)+\Delta\beta) \\
\Delta(\alpha*\beta) &:= ((\Delta\alpha)*\beta)+(\alpha*\Delta\beta)+((\Delta\alpha)*\Delta\beta) \\
\Delta(-\alpha) &:= -\Delta\alpha \\
\Delta x &:= \Theta_{new}(x)-\Theta(x) \quad (x \text{ is a variable}) \\
\Delta c &:= 0 \quad (c \in A)
\end{aligned}$$

Ring delta

$$\begin{aligned}
\phi &::- \phi \wedge \phi \mid \phi \vee \phi \mid (\phi) \mid \mathrm{true} \mid \mathrm{false} \mid R([x(,x)^*]) \mid t\ \theta\ t \\
t &::- t * t \mid t + t \mid (t) \mid c \mid x \mid \mathrm{Sum}(t, \phi)
\end{aligned}$$

Aggregation calculus

Normalized Aggregate Calculus

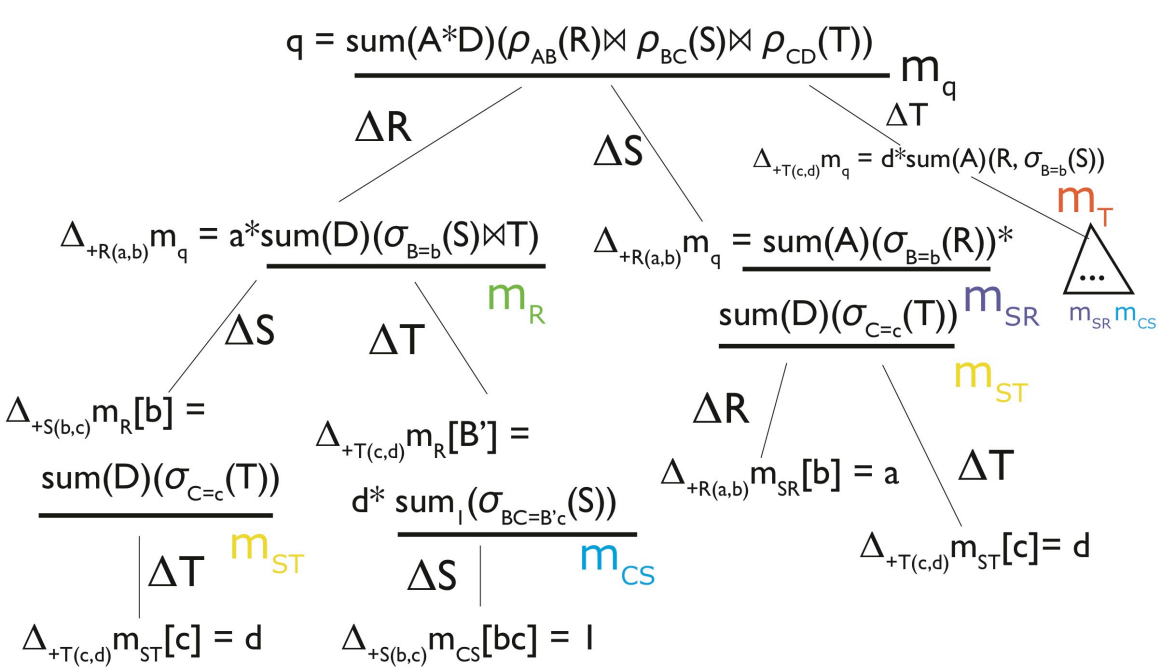
- Polynomial calculus formulae
  - Recursively monomial formulae, used as inputs to factorization
- Variable elimination via unification
  - Compute equivalence class of variables, and use for substitution
- Aggregate extraction for recursive compilation
  - Replaces maximal non-constraints only ***Sum*** subterms with map lookups
  - Replaced term used as next input to compilation
- Variable elimination for map argument simplification
  - Substitutes bound variables in map definitions (by lifting ifs)

Delta Queries

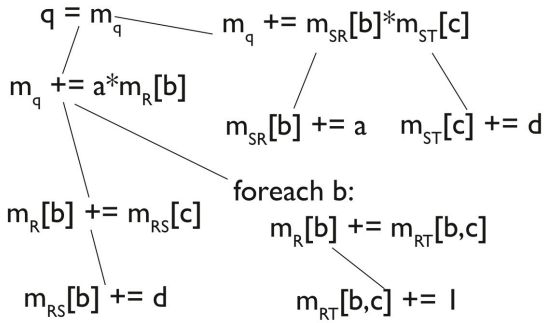
Schemas: R(a,b), S(b,c), T(c,d)

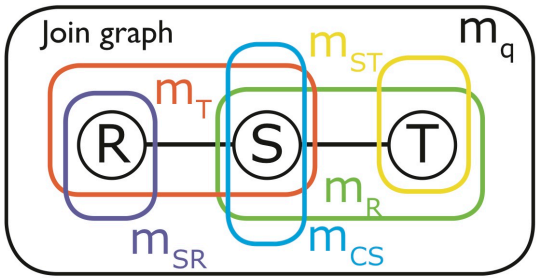
Query: `select sum(a*d) from R,S,T where R.b=S.b and S.c=T.c`

Recursive compilation



Procedural code statements





Delta Queries

Ring delta

$$
\begin{aligned}
\Delta(\alpha+\beta) &:= ((\Delta\alpha)+\Delta\beta)\\
\Delta(\alpha * \beta) &:= ((\Delta\alpha)*\beta)+(\alpha*\Delta\beta)+((\Delta\alpha)*\Delta\beta)\\
\Delta(-\alpha) &:= -\Delta\alpha\\
\Delta x &:= \Theta_{new}(x)-\Theta(x) \quad (x \text{ is a variable})\\
\Delta c &:= 0 \quad (c \in A)
\end{aligned}
$$

Atomic term and formula delta

$$
\begin{aligned}
\Delta\mathrm{Sum}(t,\phi) &:= \mathrm{Sum}((\Delta t),\phi\ )+\mathrm{Sum}(t,\Delta\phi)+\mathrm{Sum}((\Delta t),\Delta\phi)\\
\Delta(t\,\theta\,0) &:= \big(((t+\Delta t)\,\theta\,0)\wedge(t\,\bar{\theta}\,0)\big)-\big(((t+\Delta t)\,\bar{\theta}\,0)\wedge(t\,\theta\,0)\big)\\
\Delta\big(R(\vec{x})\big) &:= \bigvee_{(\vec{t}\mapsto\pm n)\in R^{\Delta\mathcal{A}}} \pm n \bigwedge_{i=1}^{|\mathrm{sch}(R)|}(x_i=t_i)
\end{aligned}
$$

```
SELECT   C1.cid, SUM(1)
FROM     Customer C1, Customer C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

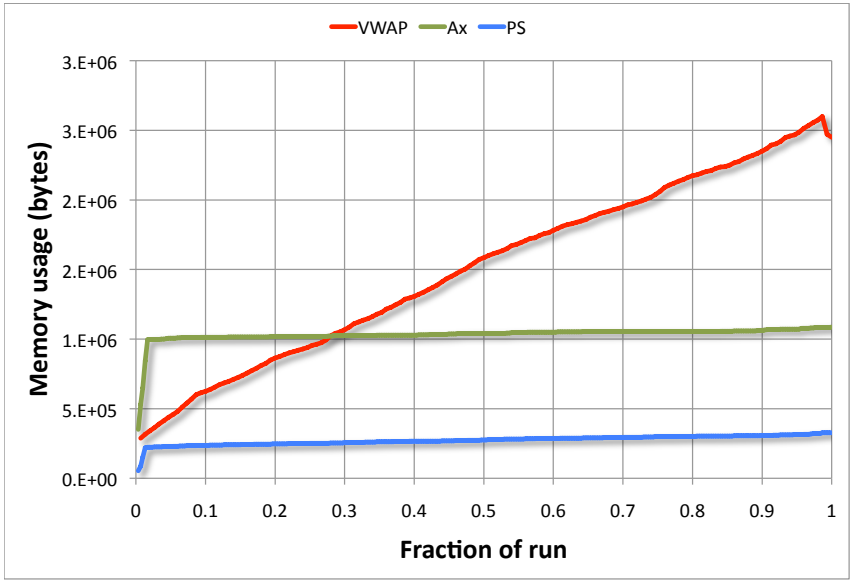
$$q[c_1] = \mathrm{Sum}(1, C(c_1,n_1)\wedge C(c_2,n_2)\wedge n_1 =$$

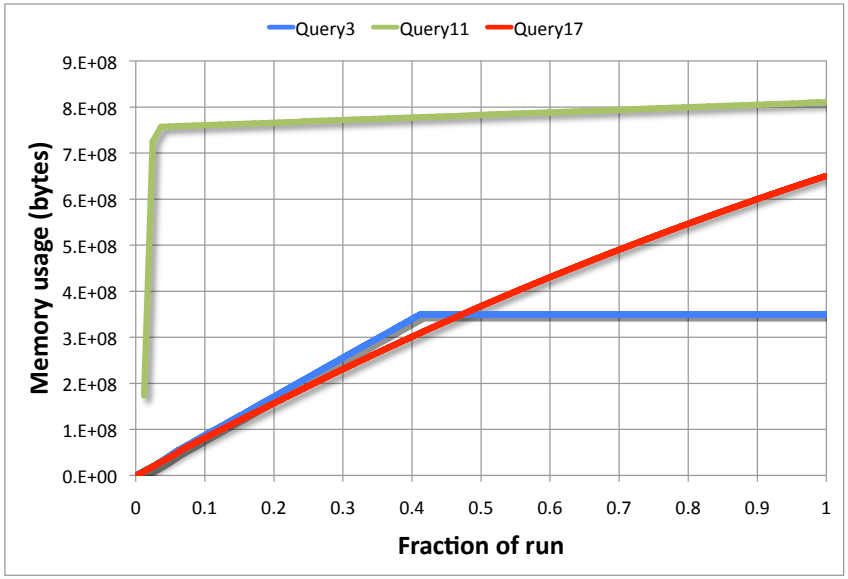
$$
\begin{aligned}
&\Delta_{\pm C(c,n)}\mathrm{Sum}(1, C(c_1,n_1)\wedge C(c_2,n_2)\wedge n_1=n_2) =\\
&\quad \mathrm{Sum}(1, \Delta_{\pm C(c,n)}(C(c_1,n_1)\wedge(C(c_2,n_2)\wedge n_1=n_2))) =\\
&\quad \mathrm{Sum}(1, ((\pm(c_1=c\wedge n_1=n))\wedge(C(c_2,n_2)\wedge n_1=n_2))\vee\\
&\qquad (C(c_1,n_1)\wedge(\pm(c_2=c\wedge n_2=n))\wedge n_1=n_2)\vee\\
&\qquad\quad ((\pm(c_1=c\wedge n_1=n))\wedge(\pm(c_2=c\wedge n_2=n))\wedge n_1=n_2) =
\end{aligned}
$$

$$
\begin{aligned}
&\pm\ \mathrm{Sum}(1, c_1=c\wedge n_1=n\wedge C(c_2,n_2)\wedge n_1=n_2)\\
&\pm\ \mathrm{Sum}(1, C(c_1,n_1)\wedge c_2=c\wedge n_2=n\wedge n_1=n_2)\\
&+\ \mathrm{Sum}(1, c_1=c\wedge n_1=n\wedge c_2=c\wedge n_2=n\wedge n_1=n_2)
\end{aligned}
$$

$$\pm\mathrm{Sum}(1, c_1=c\wedge C(c_2,n))\pm\mathrm{Sum}(1, C(c_1,n))+\mathrm{Sum}(1, c_1=c)$$

DBToaster: Memory Utilization





- Order books are small (~1Mb), TPC-H queries are viable for large servers
- VWAP and Q17 uses an old implementation of maintaining domains for initial values

Query Workload

Order book queries

VWAP: compute a vwap over the top 25% most voluminous orders

```
select   sum(b.p*b.v)
from     bids b
where 0.25*(select sum(b1.v) from bids b1) >
      (select sum(b2.v) from bids b2 where b2.p > b.p);
```

AX: find the 'ax' in an orderbook

```
select       b.broker_id, sum(a.v-b.v)
from         bids b, asks a
where        b.broker_id = a.broker_id
and          ( (a.p+-1*b.p > 1000) or (b.p+-1*a.p > 1000) )
group by     b.broker_id
```

PS: compute the price spread between order books for significant orders

```
select    sum(a.p-b.p)
from      bids b, asks a
where     ( b.v>0.0001*(select sum(b1.v) from bids b1) )
and       ( a.v>0.0001*(select sum(a1.v) from asks a1) )
```

TPC-H queries:

Query 3: shipping priority query

```
select       l.orderkey, o.shippriority, sum(l.extendedprice)
from         customer c, orders o, lineitem l
where        c.custkey = o.custkey
and          l.orderkey = o.orderkey
group by     l.orderkey, o.shippriority;
```

Query 11: important stock identification query

```
select    ps.partkey, sum(ps.supplycost * ps.availqty)
from      partsupp ps, supplier s
where     ps.suppkey = s.suppkey
group by ps.partkey having
   sum(ps.supplycost * ps.availqty) >
       (select sum(ps.supplycost * ps.availqty)
        from  partsupp ps, supplier s
        where ps.suppkey = s.suppkey);
```

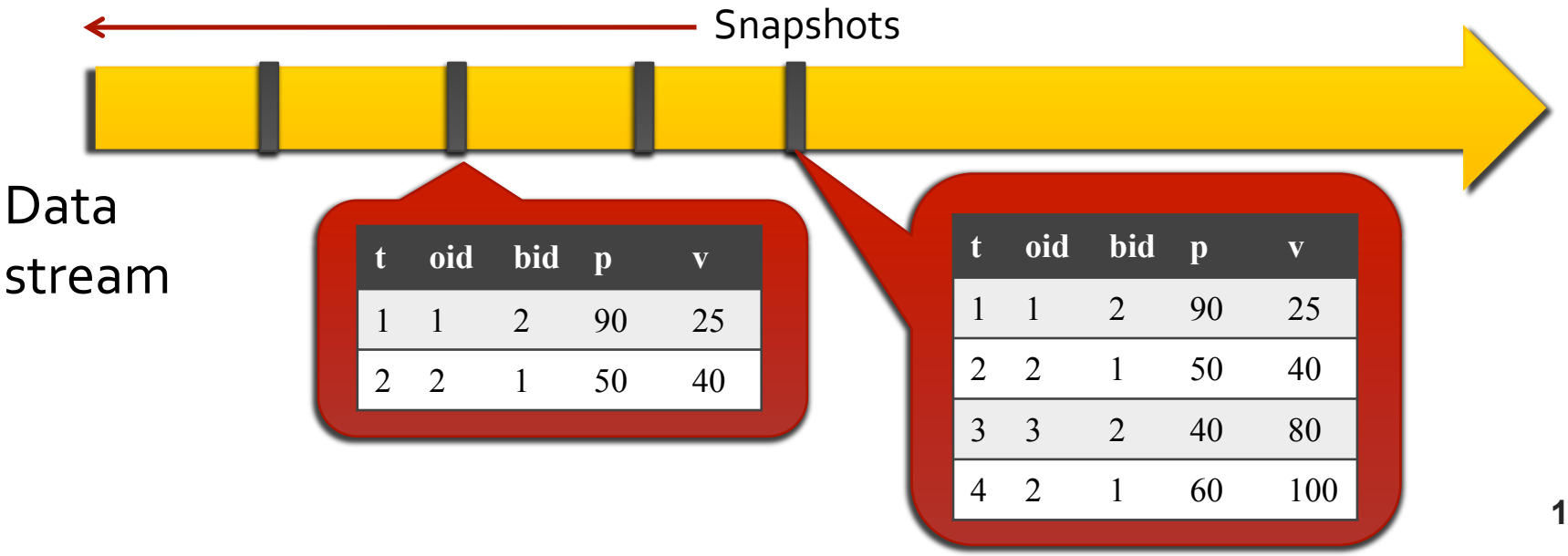
Query 17: small-quantity-order revenue query

```
select sum(l.extendedprice)
from   lineitem l, parts p
where  p.partkey = l.partkey
and    l.quantity < 0.005*
      (select sum(l2.quantity)
       from lineitem l2 where l2.partkey = p.partkey);
```

In-depth Slides

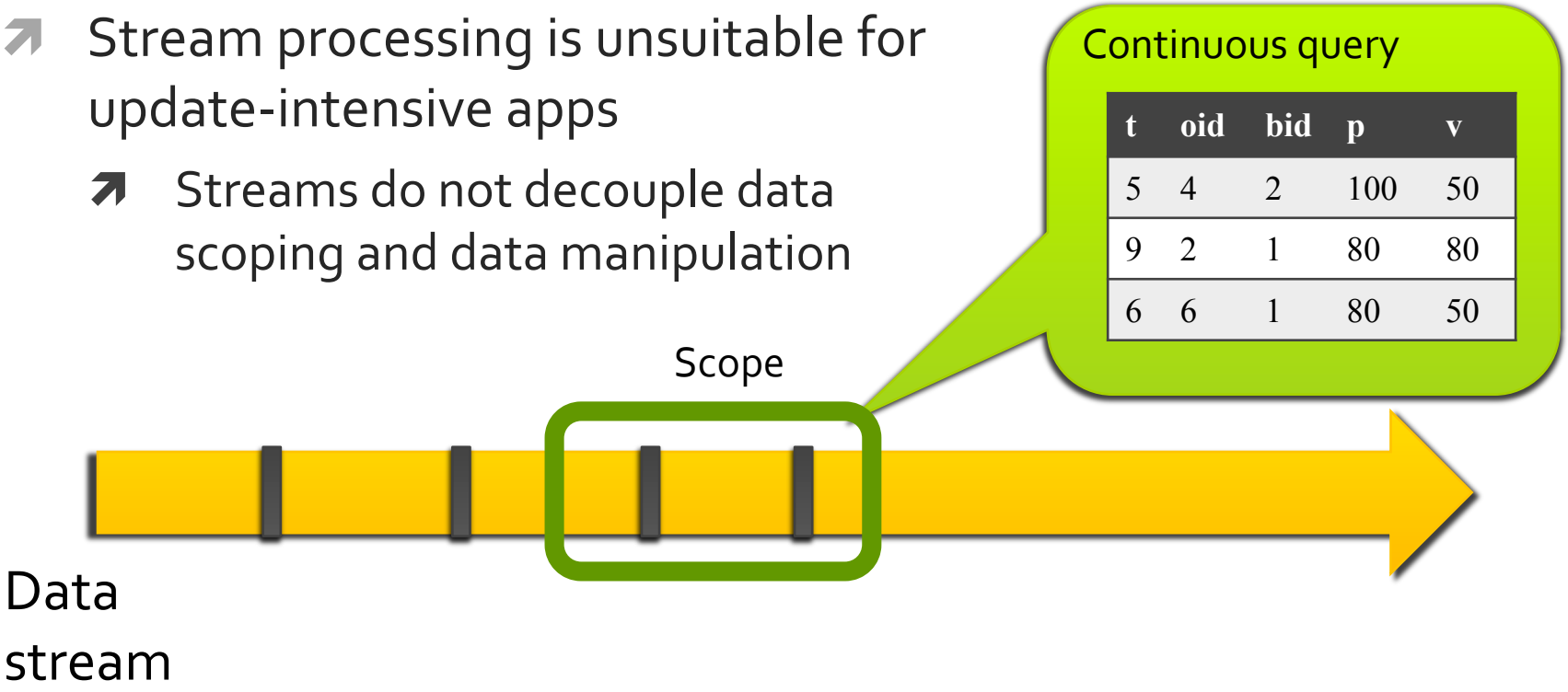
The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps
  - Streams do not decouple data scoping and data manipulation



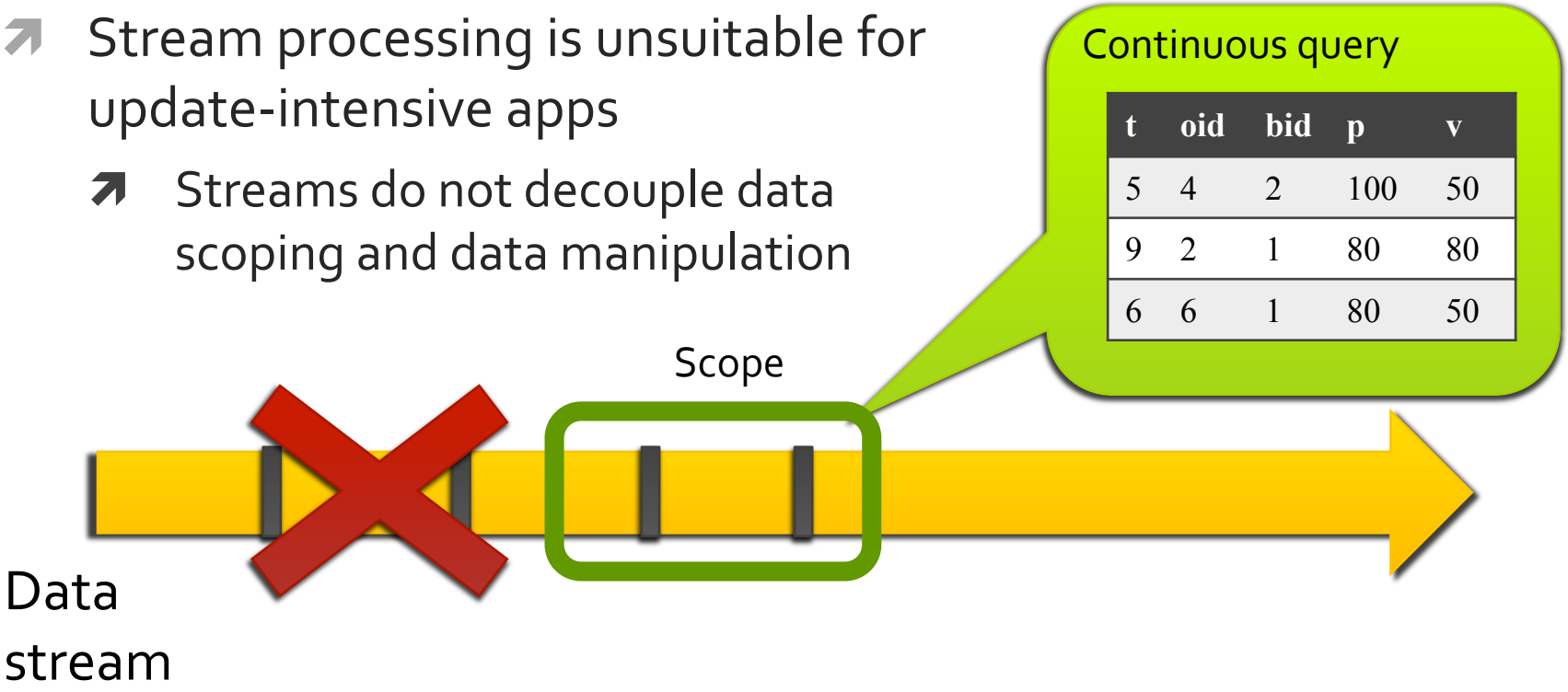
The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps
  - Streams do not decouple data scoping and data manipulation



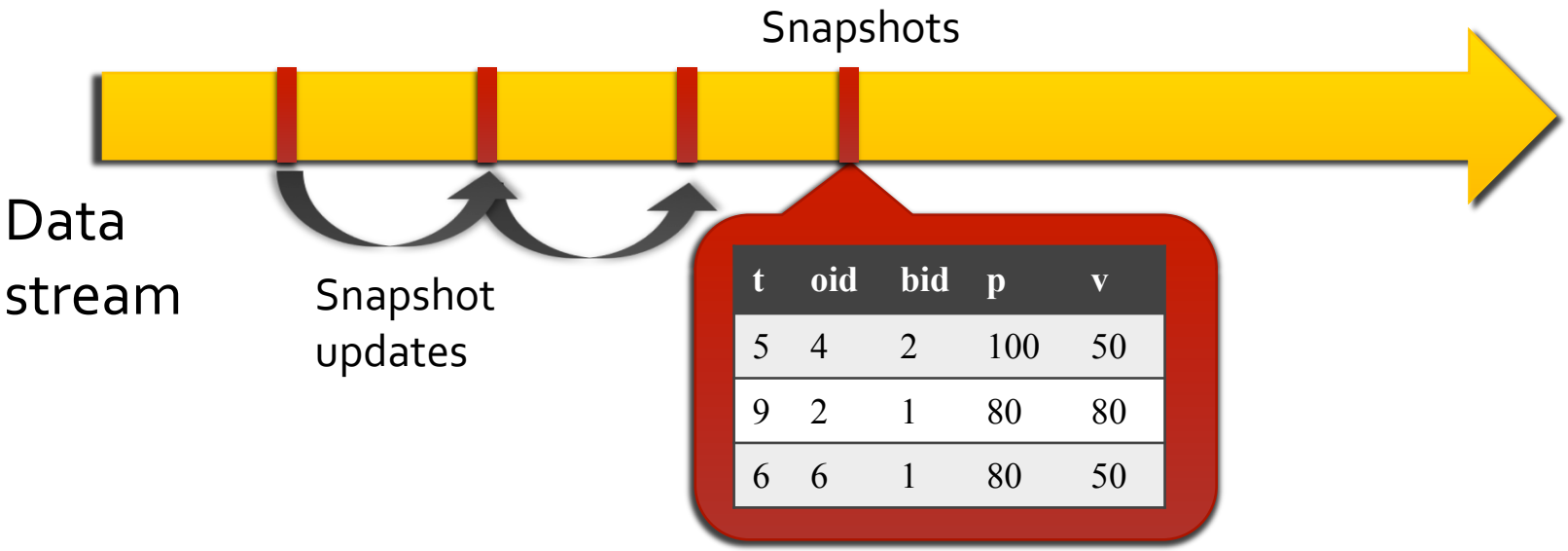
The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps
  - Streams do not decouple data scoping and data manipulation



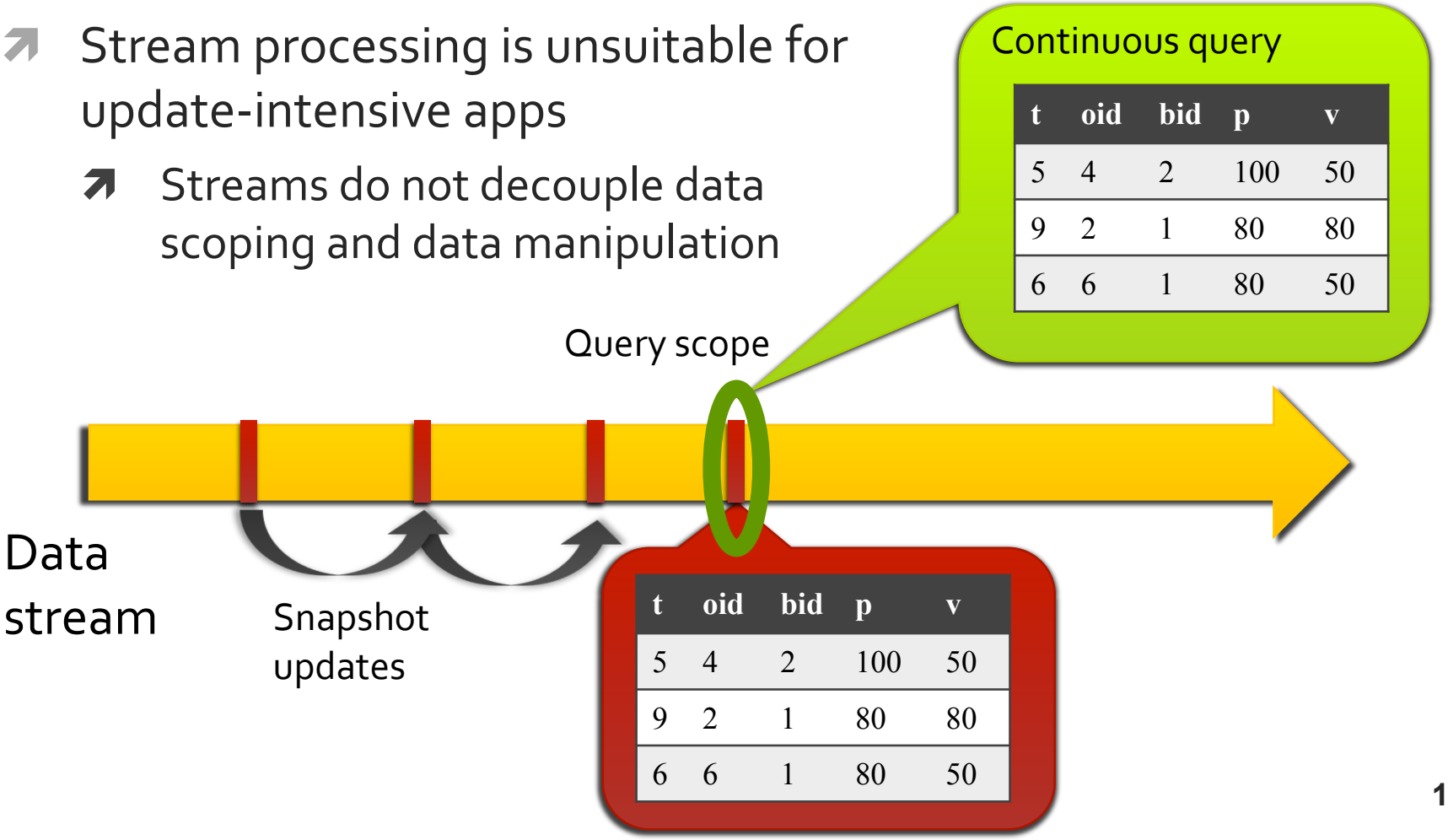
The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps
  - Streams do not decouple data scoping and data manipulation



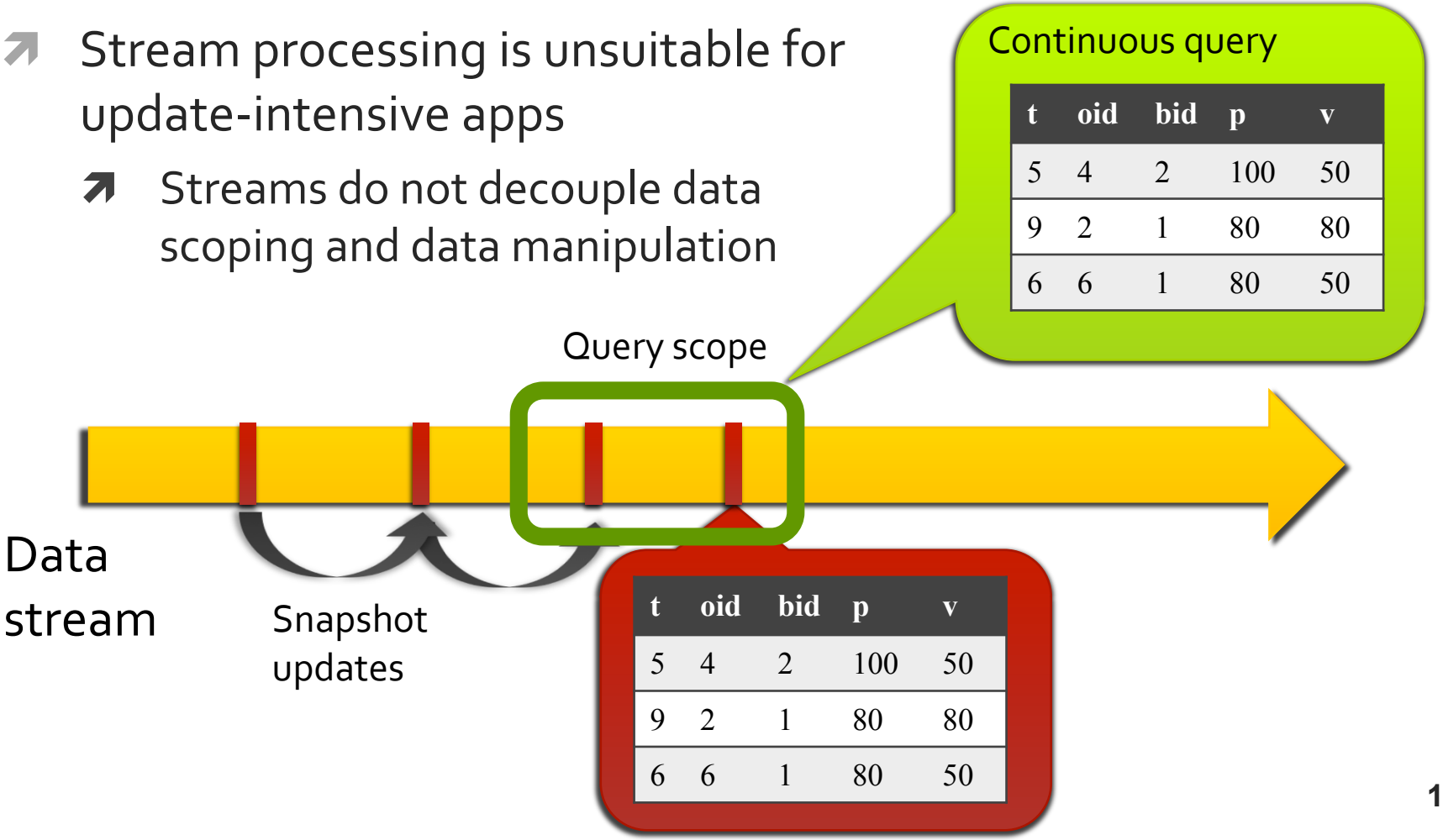
The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps
  - Streams do not decouple data scoping and data manipulation



The State of the Art in Update Processing

- Stream processing is unsuitable for update-intensive apps
  - Streams do not decouple data scoping and data manipulation



Query Factorization

```
select   sum(L.revenue), P.partcat, D.year
from   Date D, Part P
where    D.datekey = L.datekey
and    P.partkey = L.partkey
group by P.partcat, D.year
```

Query Factorization

```
        foreach pc, y: q[pc, y] =
select    sum(L.revenue)
from    Date D, Part P
where     D.datekey = L.datekey
and     P.partkey = L.partkey
and     P.partcat = pc
and     D.year = y
```

Query Factorization

```
+L(xDK, xPK, xRev) foreach pc, y: q[pc,y] +=
    select   sum(L.revenue)
    from   Date D, Part P, {<xDK, xPK, xRev>} L
    where     D.datekey = L.datekey
    and     P.partkey = P.partkey
    and     P.partcat = pc
    and     D.year = y;
```

Query Factorization

```
+L(xDK, xPK, xRev) foreach pc, y: q[pc,y] +=
    select   sum(xRev)
    from   Date D, Part P
    where     D.datekey = xDK
    and     P.partkey = xPK
    and     P.partcat = pc
    and     D.year = y;
```

Factorization:

```
select sum(t*t') from (Q x Q') :=
   (select sum(t) from Q) * (select sum(t') from Q')
```

when t is independent of Q', and t' of Q

Query Factorization

```
+L(xDK, xPK, xRev) foreach pc, y: q[pc,y] + =
    xRev*
    (select  sum(1) from Date D
    where    D.datekey = xDK
    and   D.year = y)*
    (select  sum(1) from Part P
    where    P.partkey = xPK
    and   P.partcat = pc);
```

m1[datekey,year]

m2[partkey,partcat]

Compilation Highlights

- Aggregation calculus

$$\begin{aligned} \phi &::- \phi \wedge \phi \mid \phi \vee \phi \mid (\phi) \mid \text{true} \mid \text{false} \mid R([x(,x)^*]) \mid t\ \theta\ t \\ t &::- t * t \mid t + t \mid (t) \mid c \mid x \mid \text{Sum}(t, \phi) \end{aligned}$$

- Query factorization

```
select sum(t*t') from (Q x Q') :=
    (select sum(t) from Q) *
    (select sum(t') from Q')
```

when t is independent of Q', and t' of Q

- Delta queries

$$\begin{aligned} \Delta\text{Sum}(t,\phi) &:= \text{Sum}((\Delta t),\phi\ ) + \text{Sum}(t,\Delta\phi) + \text{Sum}((\Delta t),\Delta\phi) \\ \Delta(t\,\theta\,0) &:= \big(((t+\Delta t)\,\theta\,0) \wedge (t\,\bar{\theta}\,0)\big) - \big(((t+\Delta t)\,\bar{\theta}\,0) \wedge (t\,\theta\,0)\big) \\ \Delta(R(\vec{x})) &:= \bigvee_{(\vec{t}\mapsto\pm n)\in R^{\Delta\mathcal{A}}} \pm n \bigwedge_{i=1}^{|\text{sch}(R)|} (x_i = t_i) \end{aligned}$$