DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases

Yanif Ahmad, Christoph Koch, Ki Suh Lee, Anton Morozov, Oliver Kennedy

{yanif, koch, kslee, amoroz, okennedy}@cs.cornell.edu

http://www.cs.cornell.edu/bigreddata/dbtoaster

Overview

DBToaster is a SQL compiler that generates database engines for high-performance update stream processing via incremental or (delta-) computation. DBToaster produces code that performs view maintenance of continuous aggregate queries posed on update streams, using a novel compilation approach featuring:

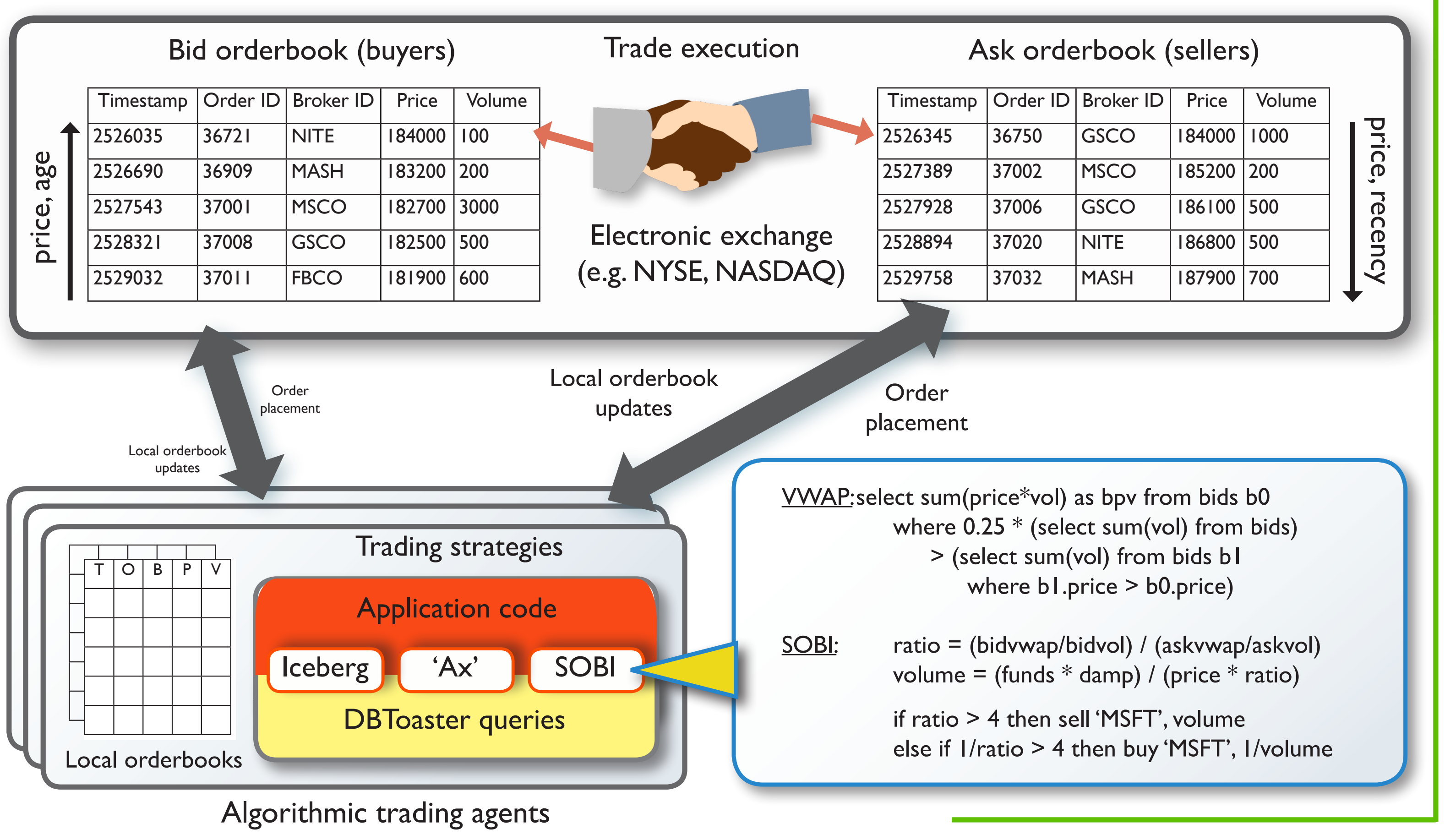
- aggressive, *recursive* query compilation
- map datastructure generation and maintenance

Our compilation technique produces a C++ function (or *handler*) for processing a single insert, update or delete that both computes a new query result and maintains supporting datastructures.

	DBToaster	Stream processors	Relational DBMS
Data model	Update streams	Append-only streams	Update streams
Query model	Continuous queries over single dynamic relation Aggregate and relational queries, with subqueries	Continuous queries over dynamic, bounded relation (windows, punctuations) Operator-specific bounds (e.g. sliding window joins)	One-time queries over database snapshots Views (no subqueries)
Query executor features	Compiled tuple handlers Query-specific map datastructures (aggregate indexes)	Compiled query plans (e.g. StreamBase, System S) Operator-specific datastructures	Interpreted query plans View maintenance with query plans Single-level "delta"

Algorithmic trading on orderbook data

Electronic exchanges such as NASDAQ and NYSE maintain orderbooks to facilitate equities trading, and have opened up these orderbooks in recent years to provide much greater visibility into the market microstructure. Orderbooks consist of two simple relations indicating buyers' and sellers' orders for a particular stock, and are heavily used in algorithmic trading.



Bid orderbook (buyers)

Timestamp	Order ID	Broker ID	Price	Volume
2526035	36721	NITE	184000	100
2526690	36909	MASH	183200	200
2527543	37001	MSCO	182700	3000
2528321	37008	GSCO	182500	500
2529032	37011	FBCO	181900	600

Ask orderbook (sellers)

Timestamp	Order ID	Broker ID	Price	Volume
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

```
VWAP:select sum(price*vol) as bpv from bids b0
        where 0.25 * (select sum(vol) from bids)
          > (select sum(vol) from bids b1
            where b1.price > b0.price)

SOBI:   ratio = (bidvwap/bidvol) / (askvwap/askvol)
        volume = (funds * damp) / (price * ratio)

        if ratio > 4 then sell 'MSFT', volume
        else if 1/ratio > 4 then buy 'MSFT', 1/volume
```

Map algebra, and recursive query compilation

DBToaster performs recursive query compilation, where each level of recursion:

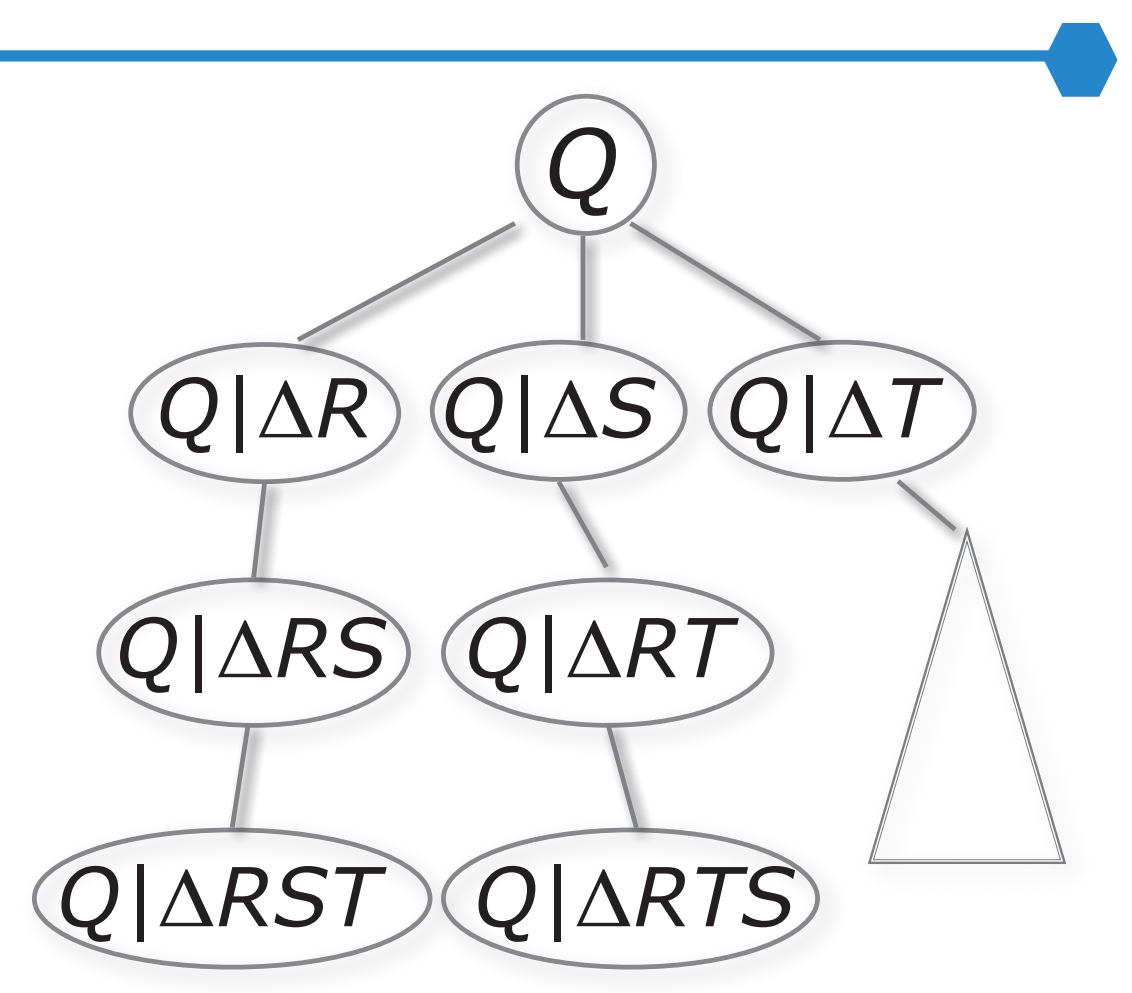
- considers a combination of base relation deltas
- simplifies the query by applying a set of map algebra transformations based around distributing and decomposing aggregates over joins
- uses the simplified query as the definition of a map datastructure
- recurs, considering further base relation deltas, from the simplified query
- terminates when all base relation combinations have been considered, and stitches together code snippets produced for each delta

DBToaster uses a map algebra to represent aggregate queries, where maps are group-by aggregate indexes computed over join subgraphs. The map algebra:

- simplifies queries by exploiting substitution of base relation with a delta tuple
- supports nested queries: map terms may appear in predicates, or aggregates
- has rules categorized as: delta rules, aggregate and relational rewrite rules, map domain maintenance, nested map maintenance

Takeaways:

- recursive compilation yields simpler and simpler queries, each step reduces the target query for aggressive simplification at the next compilation step
- DBToaster creates a query transducer for processing arbitrary update streams



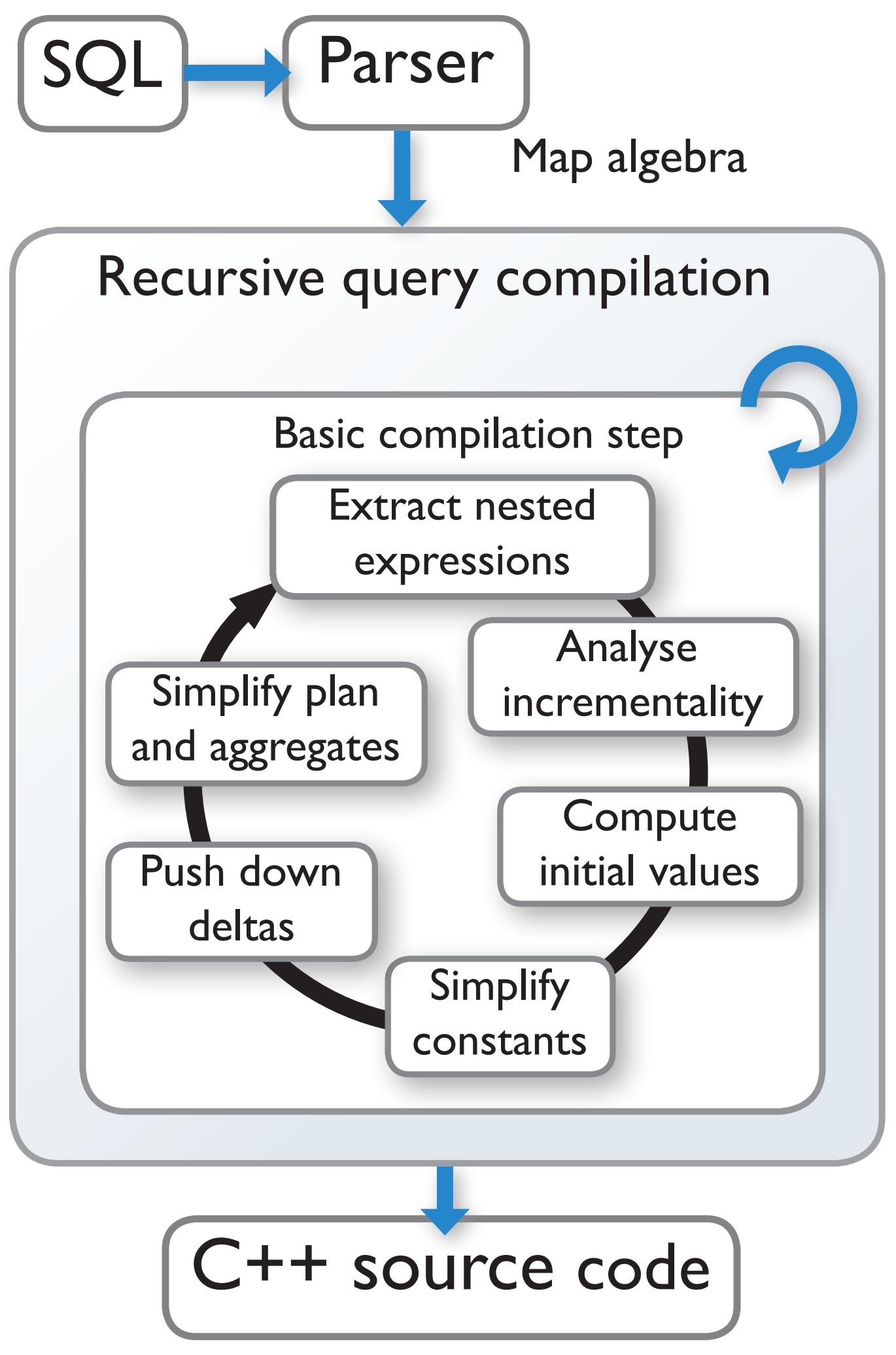
Map algebra terms

c, x	constants, variables
f + g, min(f,g)	arithmetic operators
$\mathrm{sum}_f(Q)$, $\mathrm{min}_f(Q)$	aggregates
$\mathrm{incr}_s f$, $\mathrm{incr}_s Q$	increments

Example rules

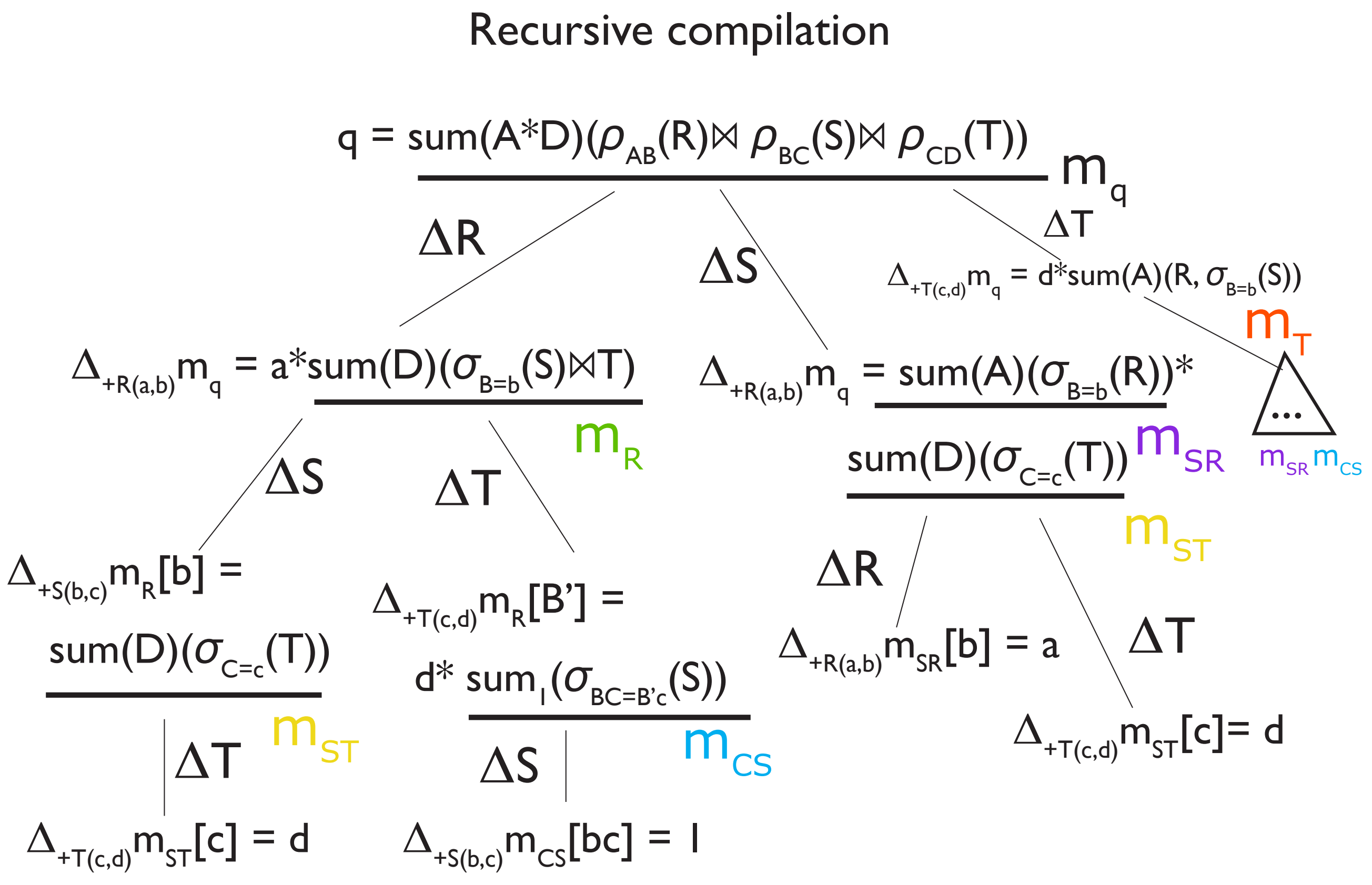
$$\begin{aligned}
\Delta_{\pm R(\vec{r})}\mathrm{sum}_f(Q) &:= \mathrm{sum}_{\Delta_{\pm R(\vec{r})} f}(Q) && (1)\\
&+ \mathrm{sum}_f(\Delta_{\pm R(\vec{r})}Q)\\
&+ \mathrm{sum}_{\Delta_{\pm R(\vec{r})} f}(\Delta_{\pm R(\vec{r})}Q)\\
\Delta_{\pm R(\vec{r})}(Q_1 \times Q_2) &:= ((\Delta_{\pm R(\vec{r})}Q_1) \times Q_2) && (2)\\
&\cup (Q_1 \times (\Delta_{\pm R(\vec{r})}Q_2))\\
&\cup ((\Delta_{\pm R(\vec{r})}Q_1) \times (\Delta_{\pm R(\vec{r})}Q_2))\\
\mathrm{sum}_{f[\vec{A};\ldots]*g[\vec{B};\ldots]}(\rho_{\vec{A}}(Q_1) \times \rho_{\vec{B}}(Q_2)) &:= \mathrm{sum}_{f[\vec{A};\ldots]}(\rho_{\vec{A}}(Q_1)) && (3)\\
&* \mathrm{sum}_{f[\vec{B};\ldots]}(\rho_{\vec{B}}(Q_2))
\end{aligned}$$

Compiler workflow



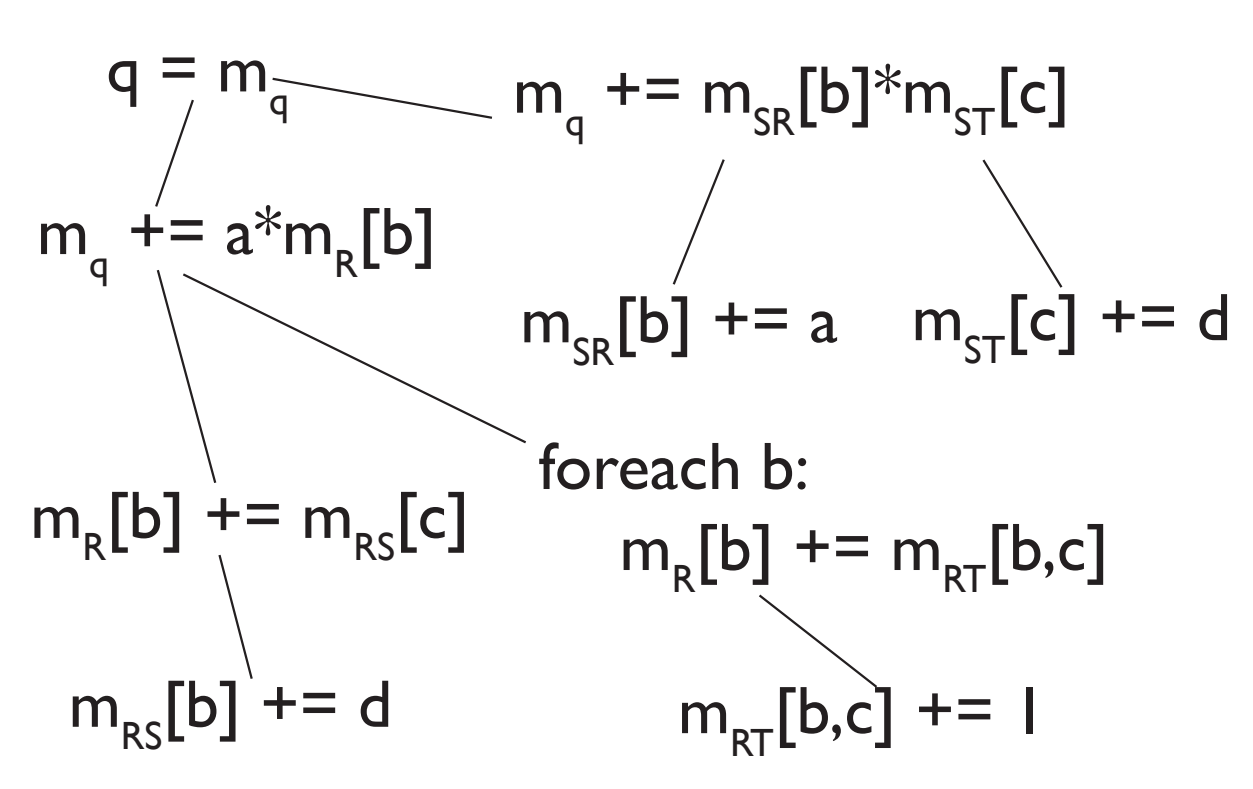
Join graph decomposition

Recursive compilation

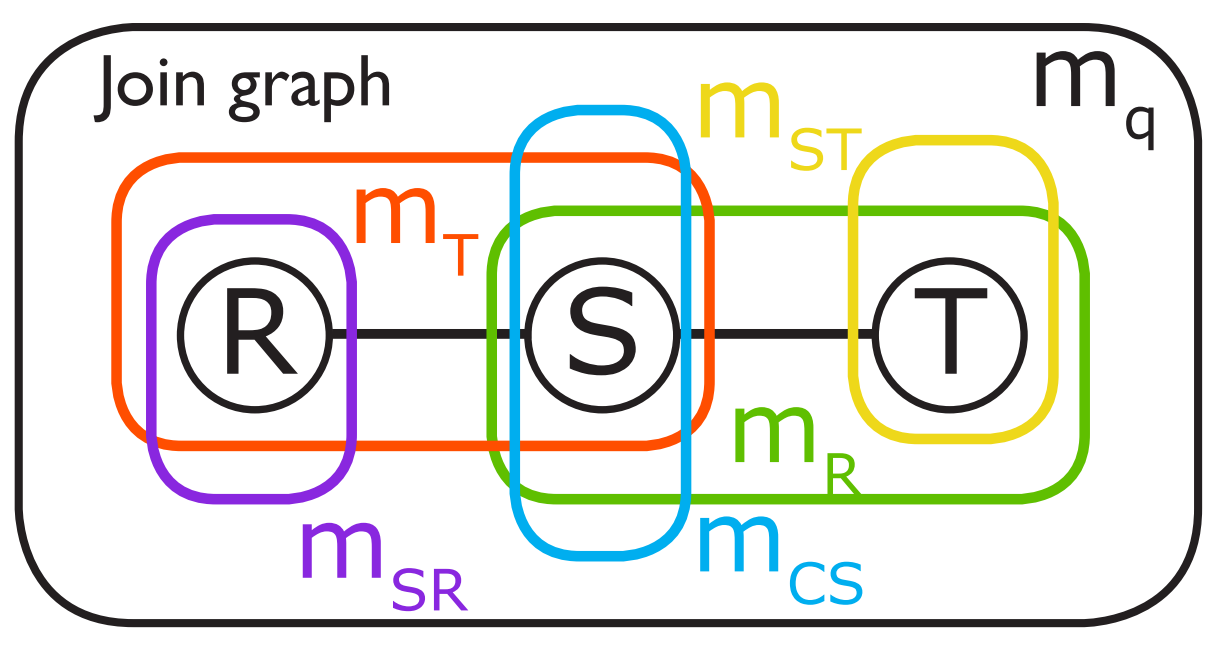


$q = \mathrm{sum}(A*D)(\rho_{AB}(R) \bowtie \rho_{BC}(S) \bowtie \rho_{CD}(T))$
m_q
ΔR
ΔS
ΔT
$\Delta_{+T(c,d)}m_q = d*\mathrm{sum}(A)(R, \sigma_{B=b}(S))$
m_T
$\Delta_{+R(a,b)}m_q = a*\mathrm{sum}(D)(\sigma_{B=b}(S) \bowtie T)$
m_R
$\Delta_{+R(a,b)}m_q = \mathrm{sum}(A)(\sigma_{B=b}(R))* \mathrm{sum}(D)(\sigma_{C=c}(T))$
m_{SR}
$m_{SR}\ m_{CS}$
m_{ST}
ΔS
ΔT
$\Delta_{+S(b,c)}m_R[b] = \mathrm{sum}(D)(\sigma_{C=c}(T))$
$\Delta_{+T(c,d)}m_R[B'] = d*\ \mathrm{sum}_1(\sigma_{BC=B'c}(S))$
ΔR
ΔT
$\Delta_{+R(a,b)}m_{SR}[b] = a$
$\Delta_{+T(c,d)}m_{ST}[c] = d$
ΔT
m_{ST}
ΔS
m_{CS}
$\Delta_{+T(c,d)}m_{ST}[c] = d$
$\Delta_{+S(b,c)}m_{CS}[bc] = 1$

Procedural code statements



$q = m_q$
$m_q \mathrel{+}= m_{SR}[b]*m_{ST}[c]$
$m_q \mathrel{+}= a*m_R[b]$
$m_{SR}[b] \mathrel{+}= a$
$m_{ST}[c] \mathrel{+}= d$
foreach b:
$m_R[b] \mathrel{+}= m_{RT}[b,c]$
$m_R[b] \mathrel{+}= m_{RS}[c]$
$m_{RS}[b] \mathrel{+}= d$
$m_{RT}[b,c] \mathrel{+}= 1$



Handler code

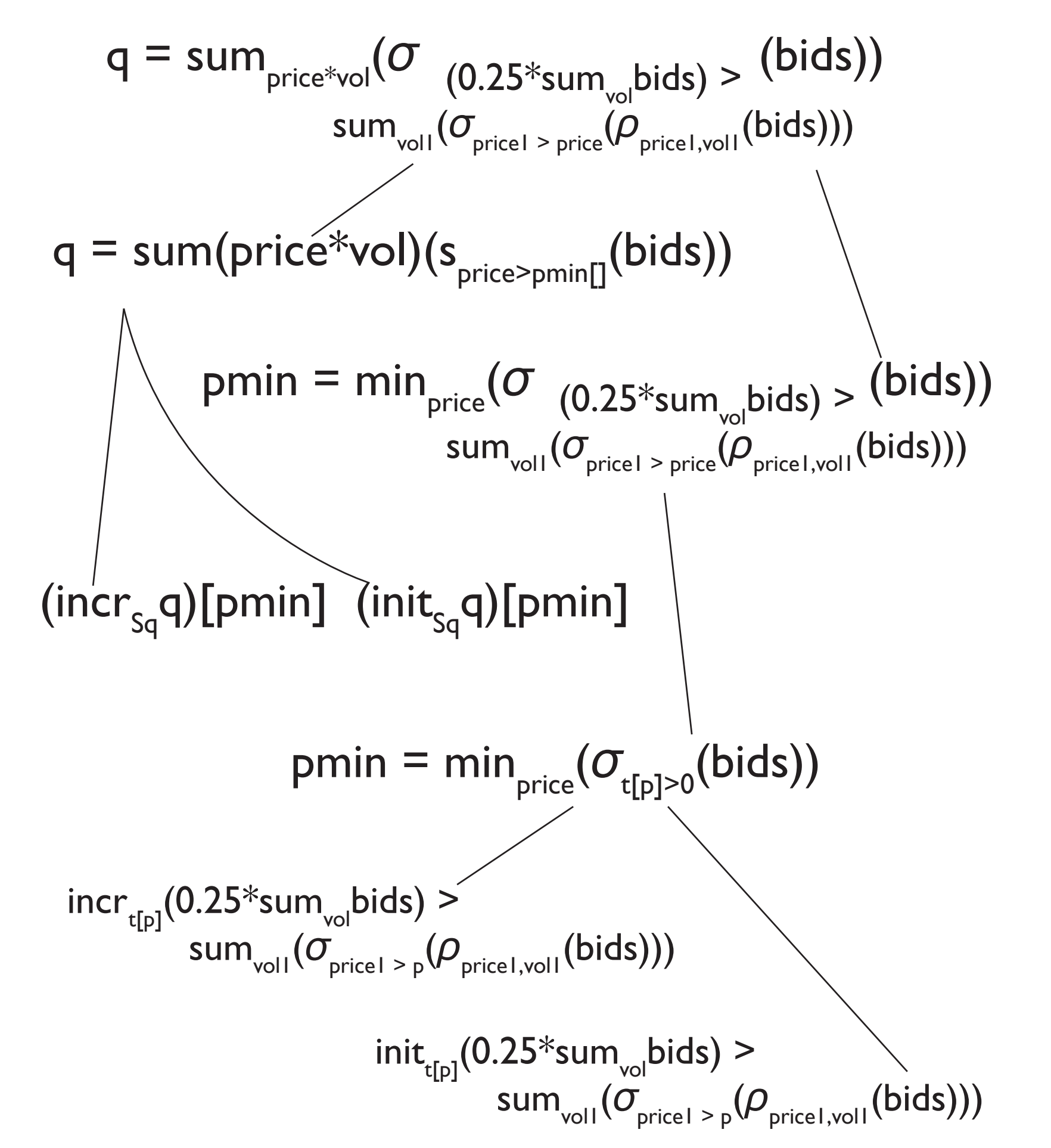
```
map<int, int> mR, mSR, mT, mST;
map<tuple<int, int>, int> mCS;

void insert_R(int a, int b)
{
   mq += a*mR[b];
   mSR[b] += a;
   iterator it = mT.begin();
   for (; it != mT.end(); ++it)
   {
      int c = it->first;
      mT[c] += a*mCS[b,c];
   }
   return mq;
}

void insert_S(int b, int c)
{
   mq += mSR[b]*mST[c];
   mR[b] += mST[c];
   mT[c] += mRS[b];
   mCS[b,c]+=1;
   return mq;
}

void insert_T(int c, int d)
{
   mq += d*mT[c];
   mST[c] += d;
   iterator it = mR.begin();
   for (; it != mR.end(); ++it)
   {
      int b = it->first;
      mR[b] += d*mCS[b,c];
   }
   return mq;
}
```

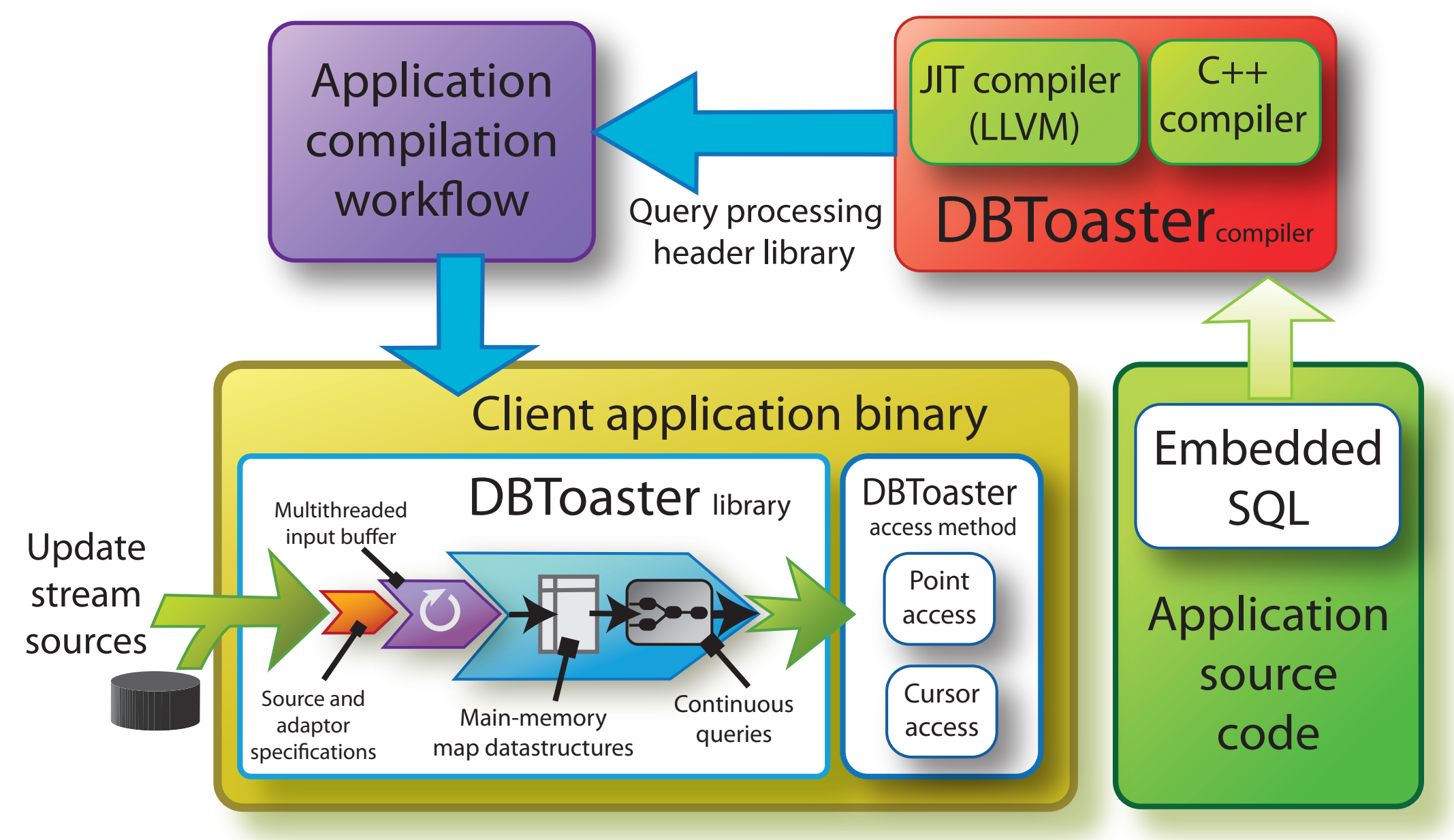
Extracting nested queries



$q = \mathrm{sum}_{price*vol}(\sigma_{(0.25*\mathrm{sum}_{vol}bids) > \mathrm{sum}_{vol1}(\sigma_{price1 > price}(\rho_{price1,vol1}(bids)))}(bids))$
$q = \mathrm{sum}(price*vol)(s_{price>pmin[]}(bids))$
$pmin = \mathrm{min}_{price}(\sigma_{(0.25*\mathrm{sum}_{vol}bids) > \mathrm{sum}_{vol1}(\sigma_{price1 > price}(\rho_{price1,vol1}(bids)))}(bids))$
$(\mathrm{incr}_{Sq}q)[pmin]$
$(\mathrm{init}_{Sq}q)[pmin]$
$pmin = \mathrm{min}_{price}(\sigma_{t[p]>0}(bids))$
$\mathrm{incr}_{t[p]}(0.25*\mathrm{sum}_{vol}bids) > \mathrm{sum}_{vol1}(\sigma_{price1 > p}(\rho_{price1,vol1}(bids)))$
$\mathrm{init}_{t[p]}(0.25*\mathrm{sum}_{vol}bids) > \mathrm{sum}_{vol1}(\sigma_{price1 > p}(\rho_{price1,vol1}(bids)))$

Nested queries require map domain maintenance (intialisation and finalisation) as attribute domains expand and contract with the update stream.

DBToaster Architecture



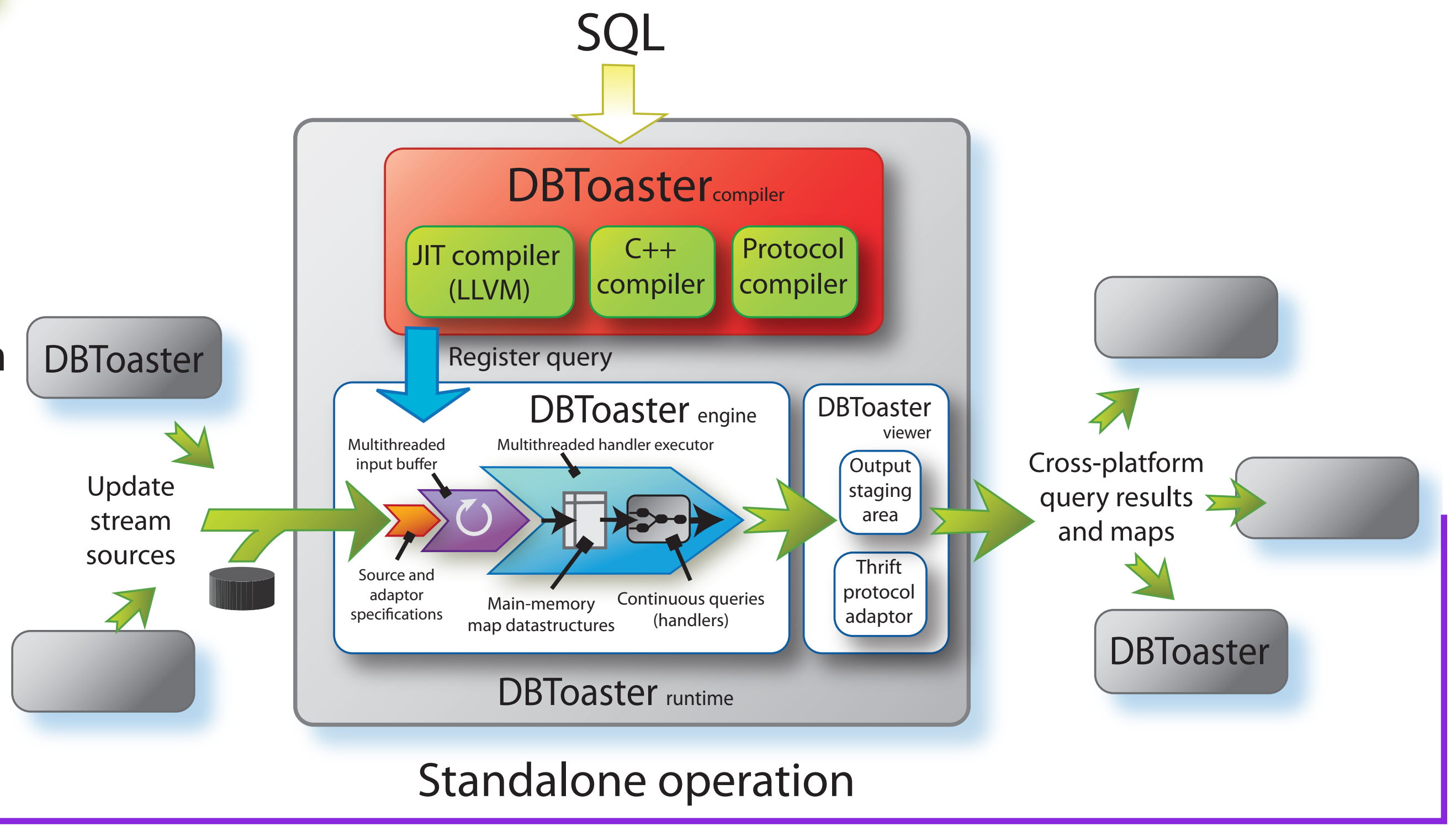
Embedded operation

With embedded operation, DBToaster provides direct in-process query processing support to client applications:

- queries run in the same process space as the app
- the DBToaster compiler behaves as a preprocessor in the app's build workflow, generating a header library
- the DBToaster engine provides two access methods for pull-based retrieval of query results beyond scalar aggregate results:
  - point access (i.e. key lookups) to maps
  - cursor-like read-only iteration over maps

With standalone operation, DBToaster runs as a server, using just-in-time compilation to dynamically build query engines. DBToaster:

- uses the Low-Level Virtual Machine (LLVM) framework as a JIT compiler for C++.
- supports both push- and pull-based result retrieval, allocating a private output staging context per client for pull-based access to maps (with the same interface as in embedded mode).
- includes a protocol compiler built on top of Apache Thrift, to provide a simple and efficient binary protocol on top of results and maps in the staging area.
- ongoing work: we are designing DBToaster from the ground-up as a shared-nothing main-memory database for scalability.



Standalone operation