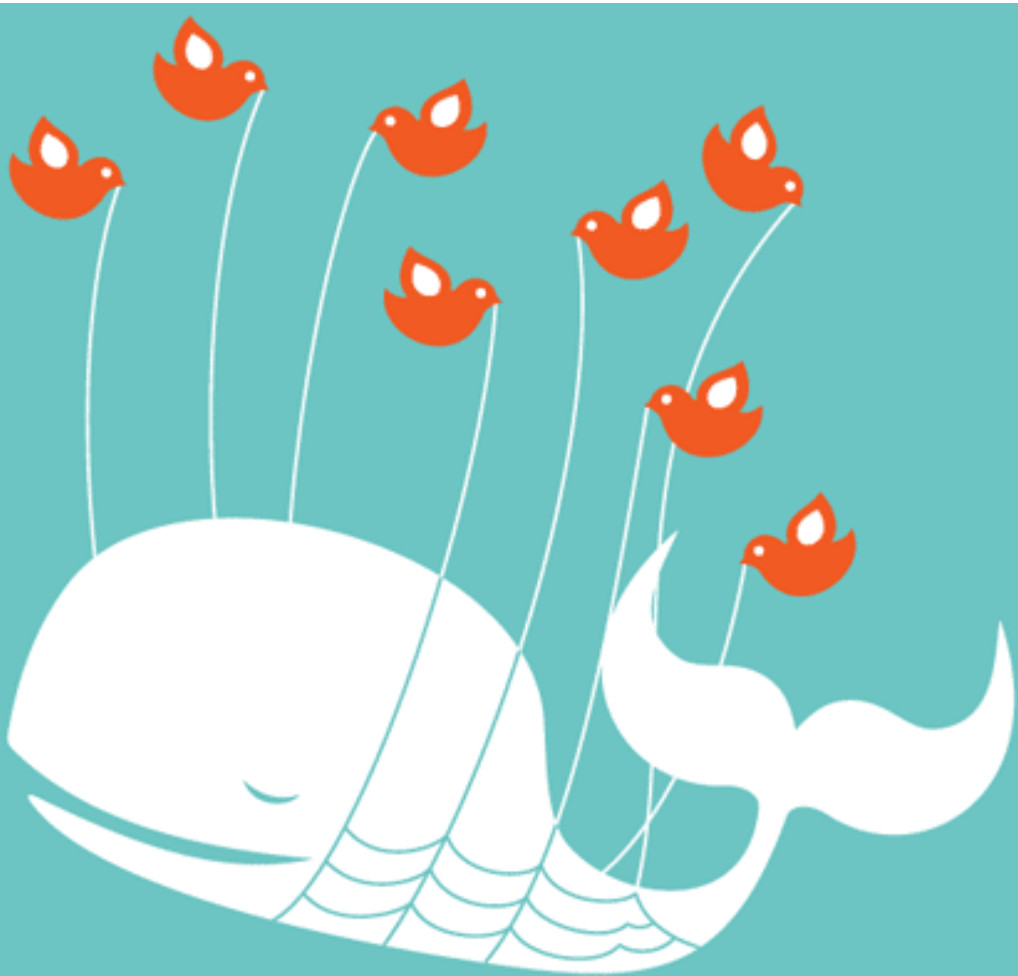
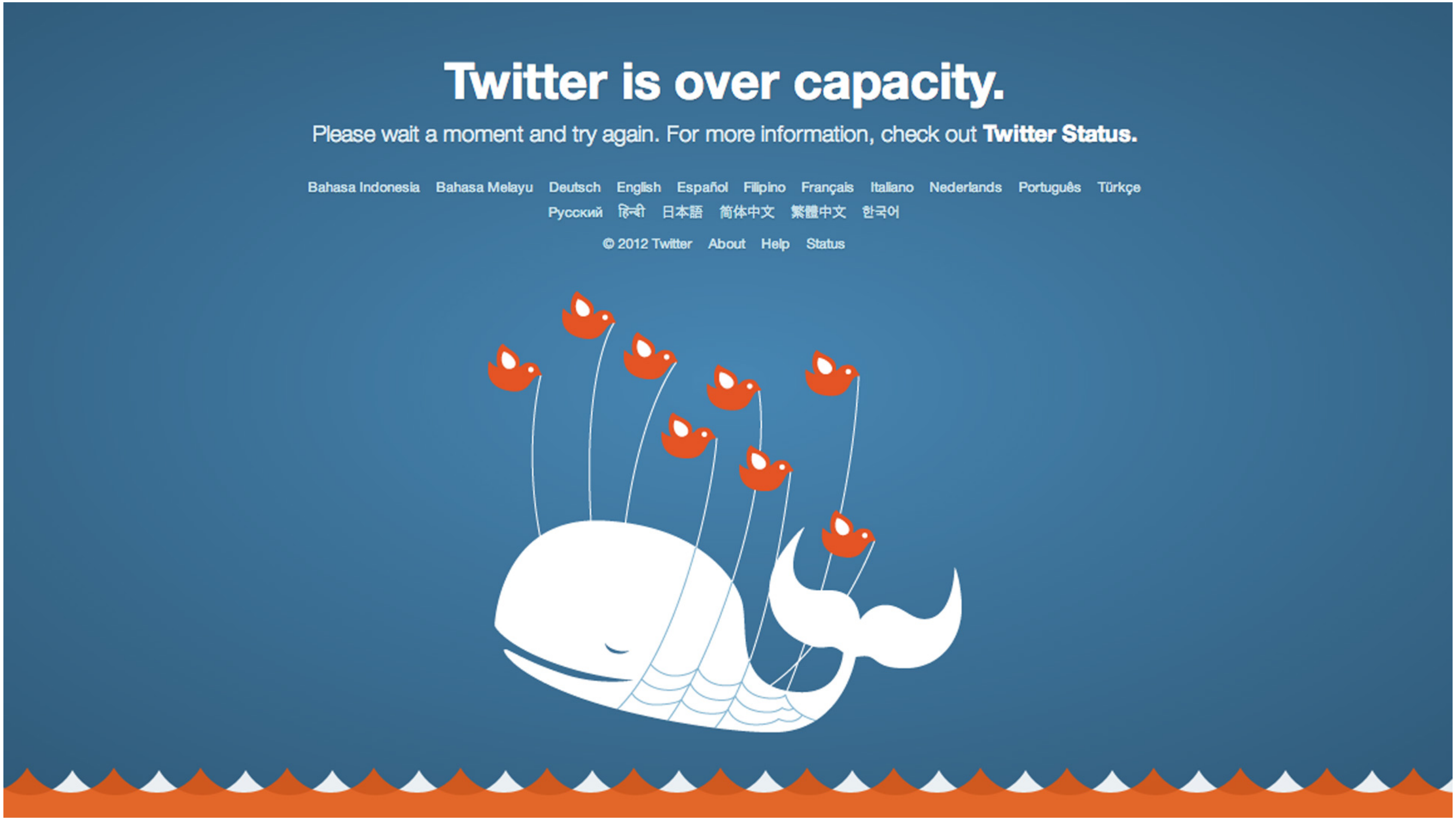


Real-time Analytics

Christoph.Koch@epfl.ch -- EPFL DATA Lab

The Fail Whale



Real-time Analytics on Big Data

- Big Data 3V: Volume, Velocity, Variety
  - This talk focuses on **velocity** and volume
- Continuous data analysis
  - Stream monitoring & mining; enforcing policies/security.
- Timely response required (low latencies!)
- Performance: high throughput and low latencies!

Comp. Arch. *not* to the rescue

- Current data growth outpaces Moore’s law.
- Sequential CPU performance does not grow anymore (already for three Intel processor generations).
  - Logical states need time to stabilize.
- Moore’s law to fail by 2020: Only a few (2?) die shrinkage iterations left.
  - Limitation on number of cores.
- Dennard scaling (the true motor of Moore’s law) has ended
  - Energy cost and cooling problems!
  - More computational power will always be more expensive!

Parallelization is no silver bullet

- Computer architecture
  - Failure of Dennard's law: Parallelization is expensive!
- Computational complexity theory
  - There are inherently sequential problems: NC<PTIME
- Fundamental impossibilities in distributed computing:
  - Distributed computation requires synchronization.
  - Distributed consensus has a minimum latency dictated by spatial distance of compute nodes (and other factors).
    - msecs in LAN, 100s of msecs in WAN. Speed of light!
  - Max # of synchronous computation steps per second, no matter how much parallel hardware available.

Paths to (real-time) performance

- **Small data (seriously!)**
- Incrementalization (online/anytime)
- Parallelization
- Specialization

Sampling: Basics

- Idea: A small random sample S of the data often well-represents all the data
  - For a fast approximate answer, apply "modified" query to S
  - Example: select agg from R where R.e is odd

Data stream:	9	3	5	2	7	1	6	5	8	4	9	1

(n=12)

Sample S:	9	5	1	8

  - If agg is avg, return average of odd elements in S

answer: 5

  - If agg is count, return average over all elements e in S of
    - n if e is odd
    - 0 if e is even

answer: 12*3/4 =9

Unbiased: For expressions involving count, sum, avg: the estimator is unbiased, i.e., the expected value of the answer is the actual answer

Probabilistic guarantees

- Example: Actual answer is 5 ± 1 with prob ≥ 0.9
- Hoeffding’s Inequality: Let X1, ..., Xm be independent random variables with 0<=Xi <= r. Let $\bar{X} = \frac{1}{m}\sum_i X_i$ and μ be the expectation of $\bar{X}$. Then, for any $\varepsilon > 0$,

$$\Pr(|\bar{X} - \mu| \geq \varepsilon) \leq 2\exp^{\frac{-2m\varepsilon^2}{r^2}}$$

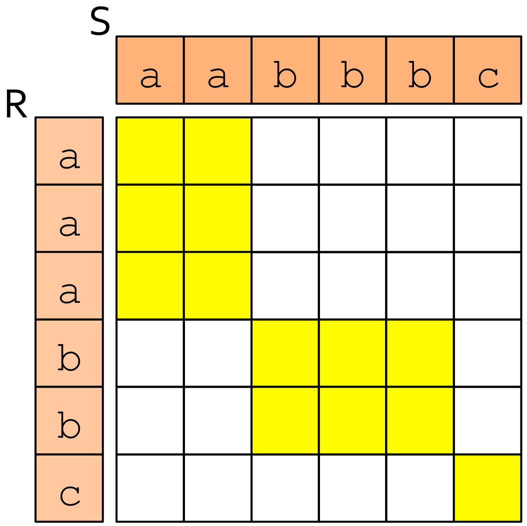
- Application to avg queries:
  - m is size of subset of sample S satisfying predicate (3 in example)
  - r is range of element values in sample (8 in example)
- Application to count queries:
  - m is size of sample S (4 in example)
  - r is number of elements n in stream (12 in example)

Queries on samples

- Generalize this to queries with joins.
- Problems:
    - How to efficiently sample from a large database?
    - Error-bounding is very hard. Requires difficult statistics for joins; open for more general SQL/analytics.
- Online aggregation: incrementally compute queries as we see a growing sample from the database.
    - Incremental operators: ripple joins!

Ripple joins: Incremental

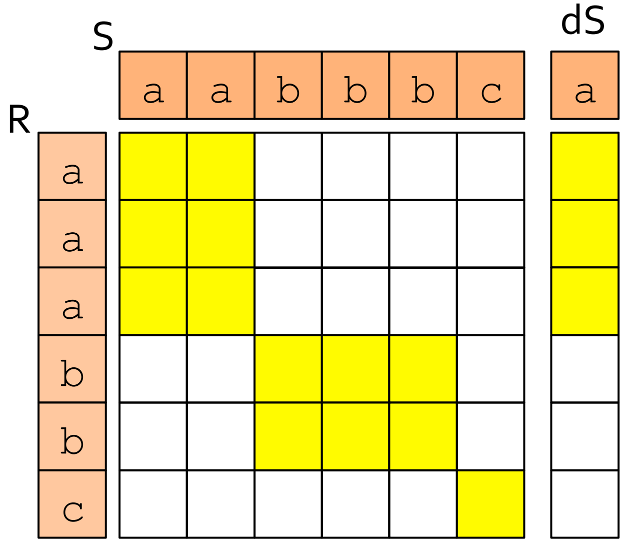
`Schema: R(A), S(A)`
`q = R natural join S;`



Count = 13

Ripple joins: Incremental

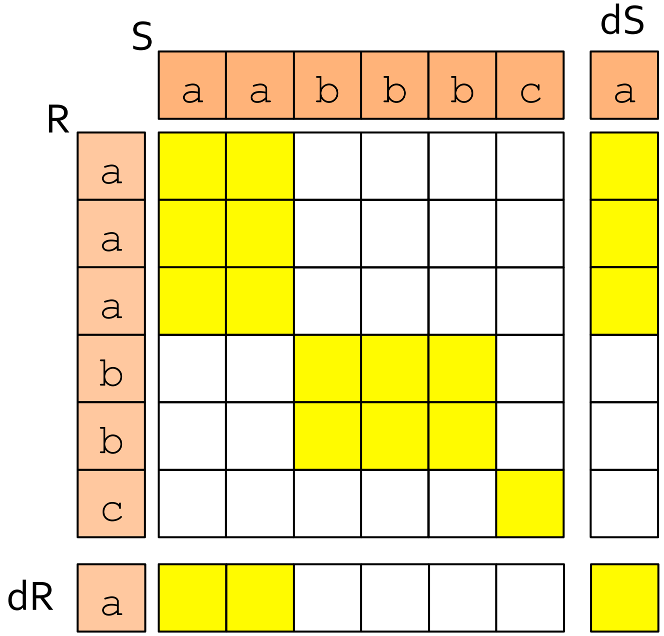
`Schema: R(A), S(A)`
`q = R natural join S;`



Count = 16

Ripple joins: Incremental

`Schema: R(A), S(A)`
`q = R natural join S;`



Count = 19

Queries on stream windows

- Windows can get large too! (particularly time-bounded windows)
- Symmetric hash-join with expiration. ~Ripple join
  - Incremental!

```
SELECT L.state, T.month, AVG(S.sales) OVER W AS movavg
FROM  Sales S, Times T, Locations L
WHERE S.timeid=T.timeid AND S.locid=L.locid
WINDOW W AS (PARTITION BY L.state
             ORDER BY T.month
             RANGE BETWEEN INTERVAL `1' MONTH PRECEDING
             AND INTERVAL `1' MONTH FOLLOWING)
```

Paths to (real-time) performance

- Small data (seriously!)
- **Incrementalization (online/anytime)**
- Parallelization
- Specialization

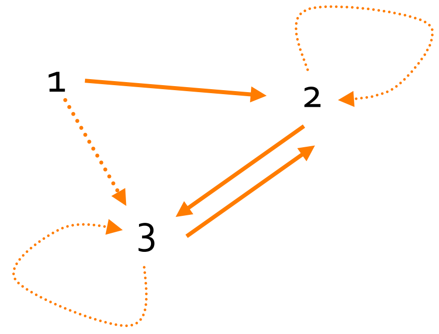
Time to understand incrementality!

- Iteration vs. updates to base data!

- Iteration: incremental once-off (seminaive datalog, gradient descent etc.)
- Updates: Incremental view maintenance (IVM).
  - Special cases (!):
    - Online aggregation
    - Window-based stream processing

Datalog example

Transitive closure of a graph:



T	
1	2
2	3
3	2
1	3
2	2
3	3

```
T(x,y) :- G(x,y)
T(x,y) :- G(x,z), T(z,y)
```

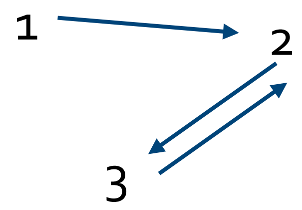
G	
1	2
2	3
3	2

Fixpoint computation:
Apply rules until fixpoint is reached.

Bottom-up: Naive Evaluation

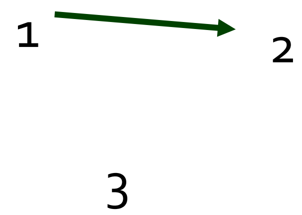
Given an EDB:

1. Start with all IDB relations empty
2. Instantiate (with constants) variables of all rules in all possible ways.
   If all subgoals become true, then infer that the head is true.
3. Repeat (2) in rounds, as long as new IDB facts can be inferred



T:

G(1,2) is TRUE!




```
T(1,2)  :- G(1,2)
T(x,y)  :- G(x,z), T(z,y)
```

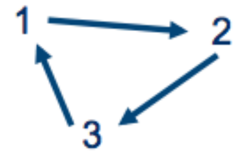
Bottom-up: Seminaive

- More efficient approach to evaluating rules
- Idea: If at round **i** a fact is inferred,
  then we must have used a rule in which one or more subgoals were instantiated to facts that were inferred on round **i-1**.
- For each IDB predicate **p**, keep both the relation **P** and a relation **ΔP**;
  the latter represents the new facts for p inferred on the most recent round.

Seminaive evaluation example

r1: `T(x,y) :- G(x,y)`
r2: `T(x,y) :- G(x,z), T(z,y)`

G:



1. Initialize IDB

T	
1	2
2	3
3	1

2. Initialize ΔIDB

ΔT	
1	2
2	3
3	1

3.a.i ΔT(x,y) := G(x,z), ΔT(z,y)

ΔT	
1	3
2	1
3	2

3.a.ii ΔT := ΔT \ T
3.a.iii T := T ∪ ΔT

T	
1	2
2	3
3	1
1	3
2	1
3	2

3.b.i ΔT(x,y) := G(x,z), ΔT(z,y)

ΔT	
1	1
2	2
3	3

3.b.ii ΔT := ΔT \ T
3.b.iii T := T ∪ ΔT

T	
1	2
2	3
3	1
1	3
2	1
3	2
1	1
2	2
3	3

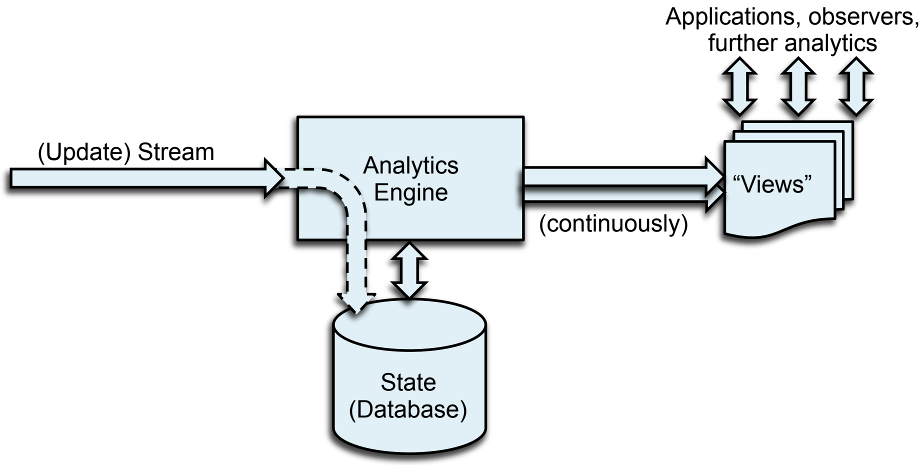
3.c
...
□

Incremental graph algorithms

- Now we talk of incrementality under updates.
- E.g. Trans. closure. Update can be done non-iteratively.
- Single-edge insert (a,b).
  TC(x,y) :- TC_old(x,a), TC_old(b,y).
  (here: reflexive TC)
- Edge delete. Possible! Represent spanning forest.
- [Patnaik and Immerman, DynFO]

Incremental analytics

- Analyzing/mining streams.
- Not by reduction to small data (samples, synopses, windows).



- Anytime algorithms.
  - Update streams.
  - Combine stream with database/historical data.
  - Can express window semantics if we want to.
- The power of algebra: orders of magnitude improvements.

Materialized Views

- SQL Views are usually intensional. Compute lazily, when needed.
- Materialized views: Do not recompute the view from scratch every time it is used.
  - Compute eagerly & store in DB.

```
CREATE MATERIALIZED VIEW empdep
REFRESH FAST ON COMMIT AS
SELECT empno, ename, dname
FROM emp e, dept d
WHERE e.deptno = d.deptno;
```

(Example in Oracle)

- Incremental view maintenance:
  - Given a DB update, perform the minimal amount og work needed to update the view.
  - (1) Determine the change to the view; (2) apply it.

Delta queries example

- Materialized view V:
```
select * from R natural join S
```

- Delta query on inserting DeltaR into R:
```
select * from DeltaR natural join S
```

- Efficient view update:
```
on insert into R tuples DeltaR do
insert into V (select * from DeltaR natural join S)
```

- Faster than replacing view by
```
select * from (R union DeltaR) natural join S
```

Deletions

- Materialized view V:
  `select * from R natural join S`

- Delta query on deleting DeltaR into R:
  `select * from DeltaR natural join S`

- Efficient view update:
  ```
  on delete from R tuples DeltaR do
  delete from V (select * from DeltaR natural join S)
  ```

- Pitfall: It's not as easy as it looks: the delta queries for inserts and deletes are generally not the same!

Self-join example

- Materialized view V (self-join / bag intersect):

```
select * from R natural join R
```

- Delta query dV on inserting DeltaR into R:

```
select * from (DeltaR natural join R)
union         (R      natural join DeltaR)
union         (DeltaR natural join DeltaR)
```

- Correct:

```
on insert into R tuples DeltaR do insert into V dV
```

- Incorrect (!!!):

```
on delete from R tuples DeltaR do delete from V dV
```

Explanation

- Incorrect (!!!):
  `on delete from R tuples DeltaR do delete from V dV`
- Assume that |R| = 2 and |dR| = 1. Then |V| = 4 and |dV| = 5
  - We try to delete more tuples than there are!

- Analogy (with corrected deletion delta):
  - `(x + dx)^2 = x^2 + (2x*dx + dx^2)`
  - `(x - dx)^2 = x^2 - (2x*dx - dx^2)`

- Intuitively, a natural join behaves like a * and a union behaves like a +.

Sets vs. multisets (bags)

- One might come to believe that the problems discussed are just due to multiset semantics, but that's not true.
- Updates and set semantics clash, we absolutely need bags.
- In set semantics, R + (R - R) does not equal (R + R) – R.
    - What do sequences of updates mean in the absence of associativity? How can we optimize?
- SQL and every relational DBMS use bag semantics.

Algebra reminder

- A semigroup is a structure (S, op) where op: S x S => S is an associative binary operation on set S.
- A group is a semigroup
  - with a neutral element e (for each a in S, a op e = e op a = a)
  - where every a in S has an inverse –a (a op –a = -a op a = e).
- Commutativity: a op b = b op a; comm. (semi)groups
- A ring is a structure (S, +, *) where (S, +) is a commutative group, (S, *) is a semigroup, and the distributive law holds:
  - a * (b + c) = a * b + a * c   (a + b) * c = a * c + b * c
  - Comm. ring: (S, *) is commutative

Cleaning up relational queries

- A generalization of relations
  - Symmetrical + operation: addition and subtraction.
  - One clean framework for insertions and deletions.
- The most important query operation: joins.
  - Still behave as expected on joins.
- A ring whose ops generalize union and join would solve all our problems.
  - A ring completely determines delta processing for its two operations.

Generalized Multiset Relations

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

- Tuple multiplicities come from a ring, such as the integers.
  - Capture databases, insertions and deletions.
- Generalize union to a group.
  - First step: make it a total operation.
  - Be able to union any two relations, regardless of schema.

Generalized Multiset Relations

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

- A (typed) tuple $\mathbb{T}$ is a *partial* function from of vocabulary of column names to data values.
- Let A be a commutative ring with 1 (such as $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$).
- A generalized multiset relation (gmr) is a function $R : \mathbb{T} \to A$ such that $R(\vec{t}) \neq 0^A$ for at most a finite number of tuples $\vec{t}$.

A ring of relations

For $R, S \in A[\mathbb{T}]$,

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

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

A ring of relations

For $R, S \in A[\mathbb{T}]$,

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

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

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

$S+T$	B	C		
		5	$\mapsto$	2
	3	5	$\mapsto$	1
	4	6	$\mapsto$	-3

$R*(S+T)$	A	B	C		
	1		5	$\mapsto$	-2
	1	3	5	$\mapsto$	-1
	1	4	6	$\mapsto$	3
	2	3	5	$\mapsto$	6

A ring of relations

- $A[\mathbb{T}]$ is a commutative ring with 1.

$$(-R) : \vec{x} \mapsto (-R(\vec{x})) \qquad 1 : \vec{x} \mapsto \begin{cases} 1 & \dots & \vec{x} = \langle\rangle \\ 0 & \dots & \vec{x} \neq \langle\rangle \end{cases} \qquad 0 : \vec{x} \mapsto 0$$

- $\mathbb{Z}[\mathbb{T}]$ is the smallest ring such that
  - all relations (set or bag-semantics) are elements and
  - $+$ and $*$ generalize union and natural join.

"$\mathbb{Z}$-relations"

Polynomials

- Polynomial ring: expressions behave exactly like we expect it from polynomials.
  - Variables: relation names.
  - Constants: constant relations (elements of the ring).
- Example: $x^2 + 2xy + 5y + 3$
  $R^2 + CRS + DS + E$.
  - R, S updatable relations (multivariate polynomial)
  - C,D,E constant relations.
  - R^2: self-join

Deltas

- Deltas follow from the ring axioms (*: distributivity!)

$$
\begin{aligned}
\Delta(\alpha + \beta) &:= ((\alpha + \Delta\alpha) + (\beta + \Delta\beta)) - (\alpha + \beta) \\
&= (\Delta\alpha) + \Delta\beta \\
\Delta(\alpha * \beta) &:= (\alpha + \Delta\alpha) * (\beta + \Delta\beta) - \alpha * \beta \\
&= (\Delta\alpha) * \beta + \alpha * (\Delta\beta) + (\Delta\alpha) * \Delta\beta \\
\Delta(-\alpha) &:= -\Delta\alpha
\end{aligned}
$$

- For polynomials of degree >0, taking the delta reduces the degree by 1! => efficiency of IVM!

Summary

- A ring of data(base relations).
- Addition generalizes union (of relational algebra).
  - Total operator: can add any two relations (typed tuples).
  - Ring (integer) multiplicities: addition is a group.
  - Databases, inserts, deletes are all the same thing: ring elements!
- Multiplication generalizes the natural join.
  - Polynomials are a useful query language.
- We get deltas for free, and they behave well!

Rings and Polynomials

- Ring: Algebraic structure with two associative operations + and *
- + is a group (there is an additive inverse).
  - One clean framework for changes (insertions and deletions)
- Powerful multiplicative operation to make queries interesting.
  - Joins, matrix multiplication
- Distributivity: query optimization and polynomials
- In a ring, we have polynomials: “queries”
  - variables = changeable data sources, e.g. relations, invertible matrices
- Notion of degree captures complexity of polynomials.
- Deltas follow from the ring axioms.
- Taking the delta reduces the degree by 1! => efficiency of IVM!

$$\begin{aligned}
\deg(\alpha * \beta) &:= \deg(\alpha) + \deg(\beta) \\
\deg(\alpha + \beta) &:= \max(\deg(\alpha), \deg(\beta)) \\
\deg(-\alpha) &:= \deg(\alpha) \\
\deg(R(\vec{x})) &:= 1.
\end{aligned}$$

$$\begin{aligned}
\Delta(\alpha + \beta) &:= ((\alpha + \Delta\alpha) + (\beta + \Delta\beta)) - (\alpha + \beta) \\
&= (\Delta\alpha) + \Delta\beta \\
\Delta(\alpha * \beta) &:= (\alpha + \Delta\alpha) * (\beta + \Delta\beta) - \alpha * \beta \\
&= (\Delta\alpha) * \beta + \alpha * (\Delta\beta) + (\Delta\alpha) * \Delta\beta \\
\Delta(-\alpha) &:= -\Delta\alpha
\end{aligned}$$

Recursive incremental processing

Given a function f, let

$$\Delta f(x) := f(x+1) - f(x) \, .$$

On increment $x \mathrel{+}= 1$: $f(x) \mathrel{+}= \Delta f(x)$.

If f is a polynomial, then $\deg(\Delta f(x)) = \max(0, \deg(f(x)) - 1)$.
So there is a k such that $\Delta^k f = 0$.

Recursive incremental processing

Given a function f, let

$$\Delta f(x) := f(x+1) - f(x).$$

On increment $x \mathrel{+}= 1$: $f(x) \mathrel{+}= \Delta f(x)$.

If f is a polynomial, then $\deg(\Delta f(x)) = \max(0, \deg(f(x)) - 1)$.
So there is a k such that $\Delta^k f = 0$.

x	$g(x) = 3x^2$	$\Delta g(x) = 6x + 3$	$\Delta^2 g(x) = 6$	$\Delta^3 g(x) = 0$
0	0	3	6	0
1	3	9	6	0
2	12	15	6	0
3	27	21	6	0
4	48	27	6	0

Compiling incremental view maintenance

Aggressive recursive incremental view maintenance: maintain $Q, \Delta Q, \Delta^2 Q, \Delta^3 Q, \ldots$.

Compile(query Q):
To incrementally maintain a materialized view of Q,

1. Compute ΔQ for tuple insertion/deletion.
2. The incremental view maintenance code is $Q \pm= \Delta Q$.
3. Recursively compile ΔQ.

Requirements on the query language L:

- L must be *closed under taking deltas*: if $Q \in L$, then $\Delta Q \in L$.
- For all $Q \in L$, there is a k such that $\Delta^k Q = 0$.

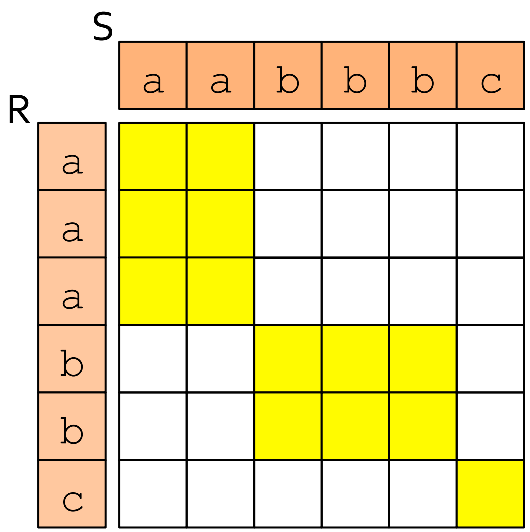
A query language for recursive IVM

CK: Incremental query evaluation in a ring of databases. PODS 2010: 87-98

- Generalized multiset relations (GMRs)
  - Integer multiplicities. Symmetrical + operation: addition and subtraction.
  - One clean framework for insertions and deletions.
  - Generalizes relational union: negative multiplicities mean delete.
  - Multiplication: generalizes the natural join operation.
  - This ring exists and is essentially unique!
- Create a useful query language that is as close as possible to the ring "language" (relations, +, *)
  - Projection/aggregation is essentially +, selection * of a GMR and a condition
  - "Aggregation Calculus"

Recursive IVM Example, SQL

```
Schema: R(A), S(A)
q = select count(*) from R natural join S;
```



Count = 13

qR

A	#
a	2
b	3
c	1

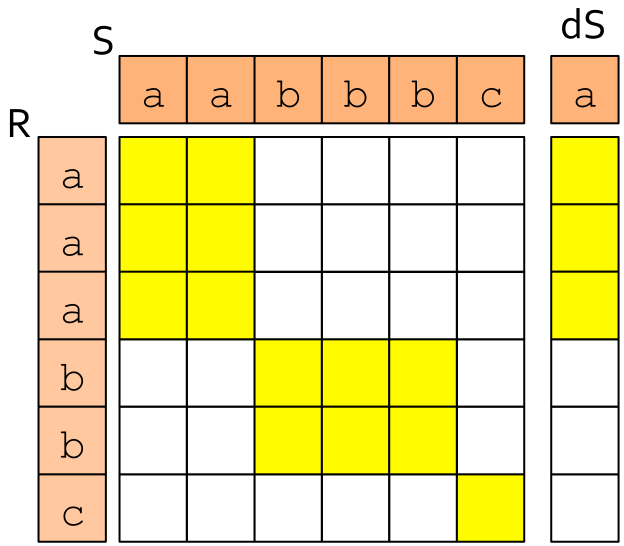
qS

A	#
a	3
b	2
c	1

```
on insert into R values (a) {
  q     += qR[a];  // select count(*)
                   // from S where A=a
  qS[a] += 1;
}
on insert into S values (a) {
  q     += qS[a];  // select count(*)
                   // from R where A=a
  qR[a] += 1;
}
```

Recursive IVM Example, SQL

```
Schema: R(A), S(A)
q = select count(*) from R natural join S;
```



Count = 16

qR

A	#
a	3
b	3
c	1

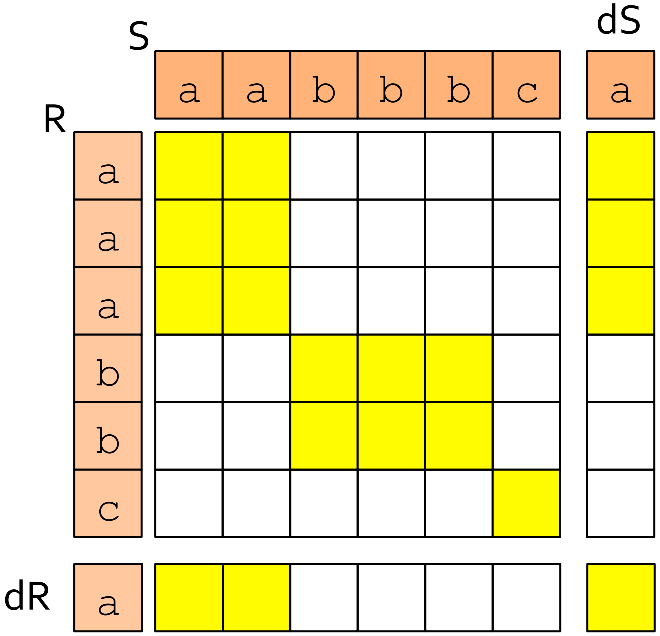
qS

A	#
a	3
b	2
c	1

```
on insert into R values (a) {
  q     += qR[a];  // select count(*)
                   // from S where A=a
  qS[a] += 1;
}
on insert into S values (a) {
  q     += qS[a];  // select count(*)
                   // from R where A=a
  qR[a] += 1;
}
```

Recursive IVM Example, SQL

```
Schema: R(A), S(A)
q = select count(*) from R natural join S;
```



Count = 19

qR

A	#
a	3
b	3
c	1

qS

A	#
a	4
b	2
c	1

```
on insert into R values (a) {
  q     += qR[a];  // select count(*)
                   // from S where A=a
  qS[a] += 1;
}
on insert into S values (a) {
  q     += qS[a];  // select count(*)
                   // from R where A=a
  qR[a] += 1;
}
```

Aggregation Calculus (AGCA)

$$\alpha \quad ::\text{-} \quad \underbrace{\alpha * \alpha \mid \alpha + \alpha \mid -\alpha \mid \mathrm{Sum}(\alpha)}_{\text{ring operations}} \mid \underbrace{R(\vec{x}) \mid f \mid x}_{\text{ring elements}} \mid (\alpha\ \theta\ 0)$$

```
q    = select sum(LI.P * O.XCH)
       from Order O, LineItem LI
       where O.OK = LI.OK;
     = Sum(O * LI * P * XCH)
deg(q) = 2
```

Deltas of AGCA queries; closure

$$\begin{aligned}
\Delta(\alpha + \beta) &:= (\Delta\alpha) + \Delta\beta \\
\Delta(\alpha * \beta) &:= (\Delta\alpha) * \beta \; + \; \alpha * (\Delta\beta) \; + \; (\Delta\alpha) * \Delta\beta \\
\Delta(-\alpha) &:= -\Delta\alpha \\
\Delta\mathrm{Sum}(\alpha) &:= \mathrm{Sum}(\Delta\alpha) \\
\Delta(\alpha\,\theta\,0) &:= \big((\alpha + \Delta\alpha)\,\theta\,0\big) * (\alpha\,\bar{\theta}\,0) - \big((\alpha + \Delta\alpha)\,\bar{\theta}\,0\big) * (\alpha\,\theta\,0)
\end{aligned}$$

$$\begin{aligned}
\Delta_{\pm R(\vec{t})} R(\vec{x}) &:= \pm(\vec{x} = \vec{t}) \\
\Delta_{\pm R(\vec{t})} S(\vec{x}) &:= 0 \qquad (R \neq S) \\
\Delta_{\pm R(\vec{t})} f &:= 0 \qquad (f \text{ built-in/constant})
\end{aligned}$$

AGCA is closed under taking deltas!

Degrees of deltas; high deltas are independent of the database

$$
\begin{aligned}
\deg(\alpha * \beta) &:= \deg(\alpha) + \deg(\beta) \\
\deg(\alpha + \beta) &:= \mathsf{max}(\deg(\alpha), \deg(\beta)) \\
\deg(-\alpha) &:= \deg(\alpha) \\
\deg(\mathrm{Sum}(\alpha)) &:= \deg(\alpha) \\
\deg(t\,\theta\,0) &:= \deg(t) \\
\deg(R(\vec{x})) &:= 1.
\end{aligned}
$$

An AGCA condition $t\,\theta\,0$ is *simple* if $\Delta t = 0$ for all update events. This is in particular true if t does not contain Sum subterms.

THEOREM 5.5. *For any AGCA term or formula α with simple conditions only,* $\deg(\Delta\alpha) = \max(0, \deg(\alpha) - 1)$.

Why compile (DBToaster) IVM code?

- Really incremental code is low-level
  - Recursive incremental view maintenance takes the idea of incrementalization to an extreme.
- Inline deltas into the query code.
- Eliminate overheads of dynamic representations of queries and interpretation. Improve cache-locality.
- Dead code elimination: Some features of the engine may not be needed. Only certain patterns of use arise.

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     qO[xOK]
      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];
                       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                    qLI[yOK]
      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 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!
- No nonincremental algorithm can do that!

DBToaster Trigger Programs

```
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
}
```

- This is (real-time) analytics, but it behaves like an OLTP workload!
- Triggers are relatively low-level: compile!

Query factorization

```
select   sum(L.revenue), P.partcat, D.year
from     Date D, Part P, LineOrder L
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, LineOrder L
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 = L.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;
```

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')
if no overlap in variables.
```

Query factorization

```
+L(xDK, xPK, xRev) foreach pc, y: q[pc, y] +=

            xRev *
            (select   sum(1)
             from     Date D
             where    D.datekey = xDK           m1[xDK,y]
             and      D.year = y) *
            (select   sum(1)
             from     Part P
             where    P.partkey = xPK           m2[xPK,pc]
             and      P.partcat = pc);
```

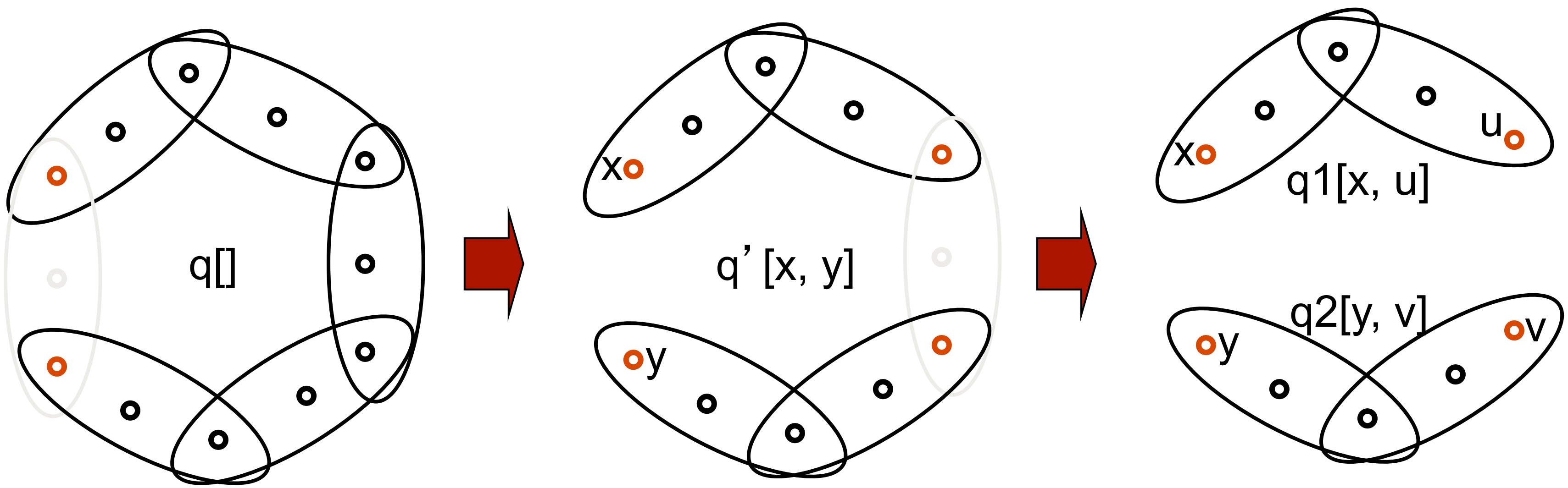
Query factorization

```
+L(xDK, xPK, xRev) foreach pc, y: q[pc, y] +=
      xRev * m1[xDK, y] * m2[xPK, pc]

m1[dk, y] = (select   sum(1)
             from     Date D
             where    D.datekey = dk
             and      D.year = y) *
m2[pk pc] = (select   sum(1)
             from     Part P
             where    P.partkey = pk
             and      P.partcat = pc);
```

Connection to query decompositions

- Recursive delta computation computes decompositions.
  - A delta computation step removes one hyperedge from the query hypergraph.



DBToaster Theory Summary

CK: Incremental query evaluation in a ring of databases. PODS 2010: 87-98

The compiled programs have surprising properties

- lower complexity than any non-incremental algorithm
- constant time for each aggregate value maintained.
- admits *embarrassing* parallelism: purely push-based parallel processing that sends *minimal* amount of data.

The DBToaster System

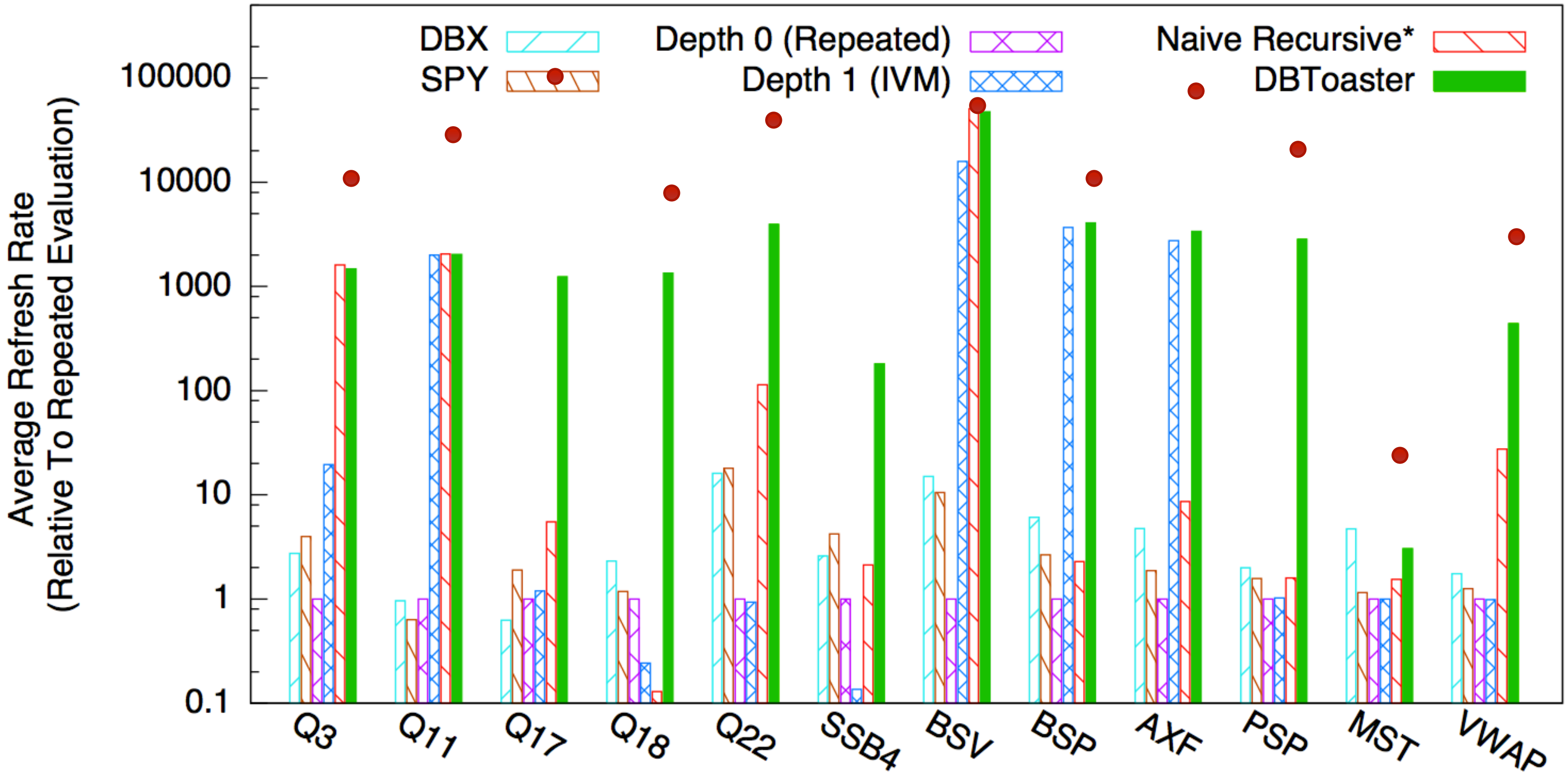
http://www.dbtoaster.org

- In this example:
  - The triggers for incrementally maintaining all the maps run in constant time!
  - No nonincremental algorithm can do that!
  - Classical IVM takes linear time.

- Triggers are really low-level: compile!
  - Aggressive inlining of deltas
  - Use algebraic laws, partial evaluation for code simplification.
  - Eliminate overheads of dynamic representations of queries and interpretation. Improve cache-locality.

CK, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, Amir Shaikhha: DBToaster: higher-order delta processing for dynamic, frequently fresh views. VLDB J. 23(2): 253-278 (2014)

DBToaster rev.2525 (2012), 1core



Yanif Ahmad, Oliver Kennedy, Christoph Koch, Milos Nikolic: DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. PVLDB 5(10): 968-979 (2012)

Paths to (real-time) performance

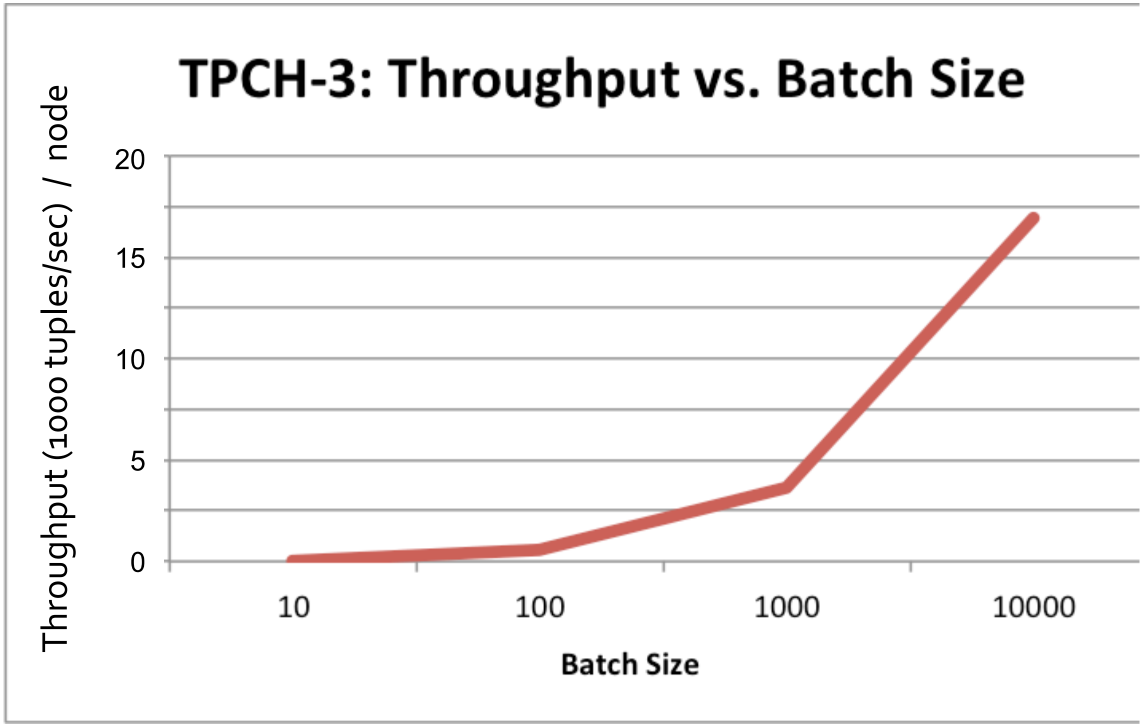
- Small data (seriously!)
- Incrementalization (online/anytime)
- **Parallelization**
- Specialization

DBToaster

- Single-core, TPC-H & algorithmic trading benchmarks
  - 4-5 orders of magnitude speedup from re-evaluation
  - 25k-70k view refreshes per second.
- Minibatching on Spark
  - latency ~0.5sec

For comparison (1-core version):

r.2525(2012): 26k/sec
r.2827(2014): 127k/sec



Low latency infrastructure

- DBToaster refresh rate, TPCH3:
  - 1-core: 127,000/s
    Spark: 2/s -- also Spark Streaming is no streaming at all
- Twitter/Apache Storm
  - ZeroMQ: >1,000,000 msgs/s/node
  - No batching, a priori no synch
- Squall -- https://github.com/epfldata/squall
  - SQL on top of Storm. Very low latencies.
  - New skew-resistant, scalable online operators (joins)

Reminder: Two-Phase Commit

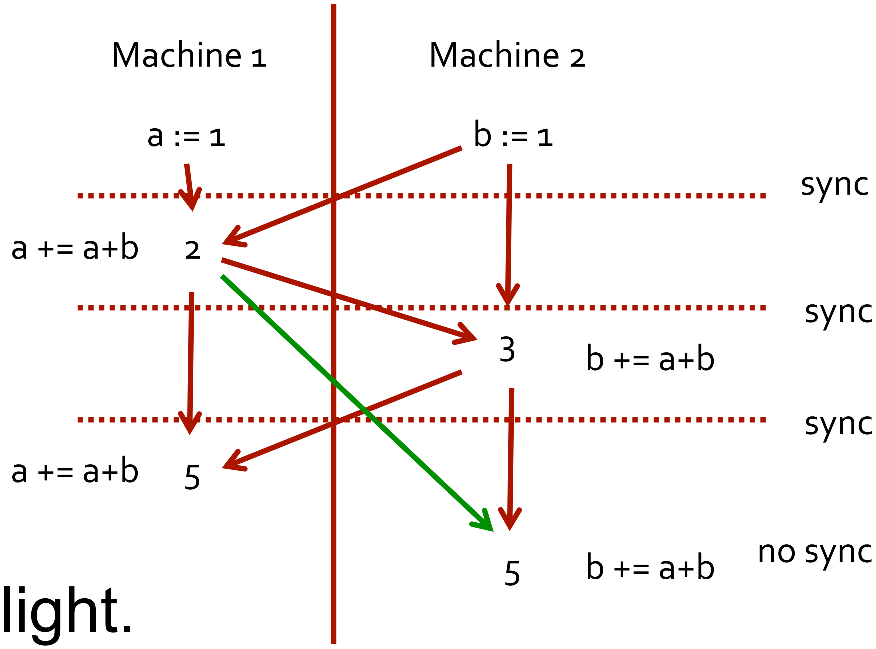
Coordinator	**Subordinate**
Send prepare	
	Force-write prepare record
	Send yes or no
Wait for all responses	
Force-write commit or abort	
Send commit or abort	
	Force-write abort or commit
	Send ACK
Wait for all ACKs	
Write end record	

The cost of synchronization

- Low-latency stream processing?
- But: no free lunch – distributed computation needs synchronization.
- Consensus
  >= 2-phase commit.

Minimum latency two network roundtrips.

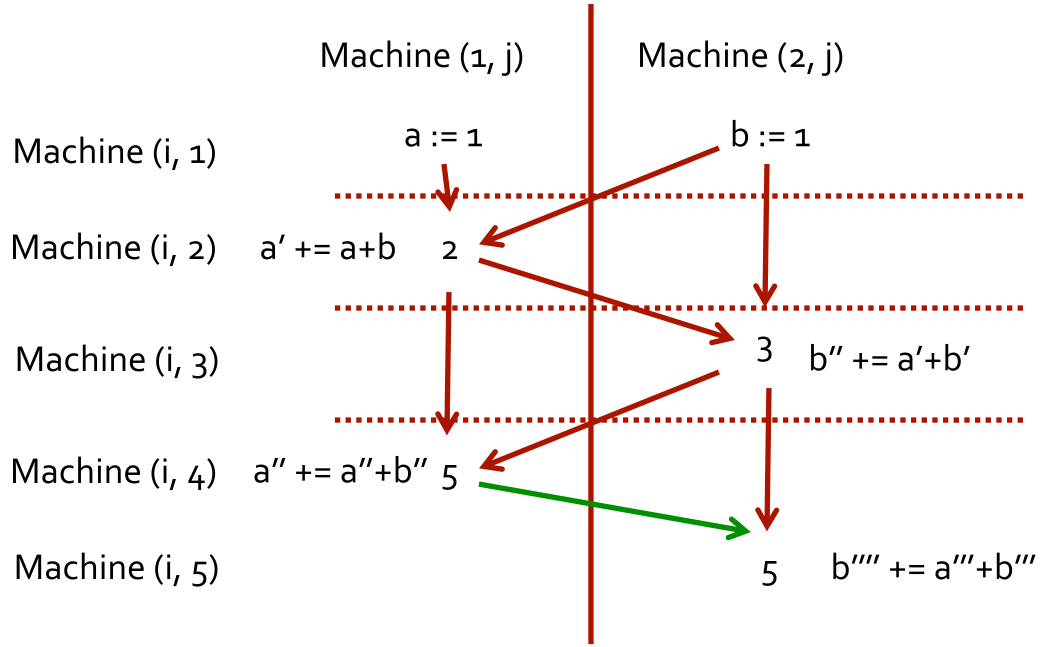
Lausanne-Shenzhen:
9491km * 4; 126ms@speed of light.



Does streaming/message passing defeat the 2PC lower bound?

- Assume we compute each statement once.
- Different machines handle statements
- Don't compute until you have received all the msgs you need.

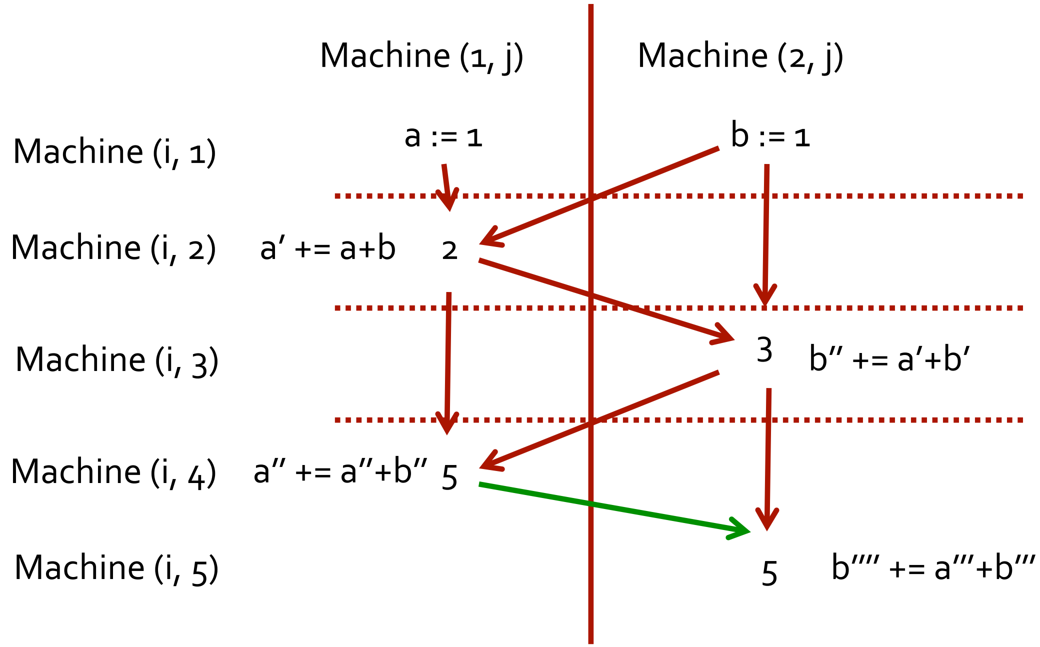
- Works!
- But requires synchronized ts on input stream.
    - One stream source or synch of stream sources!



Does streaming/message passing defeat the 2PC lower bound?

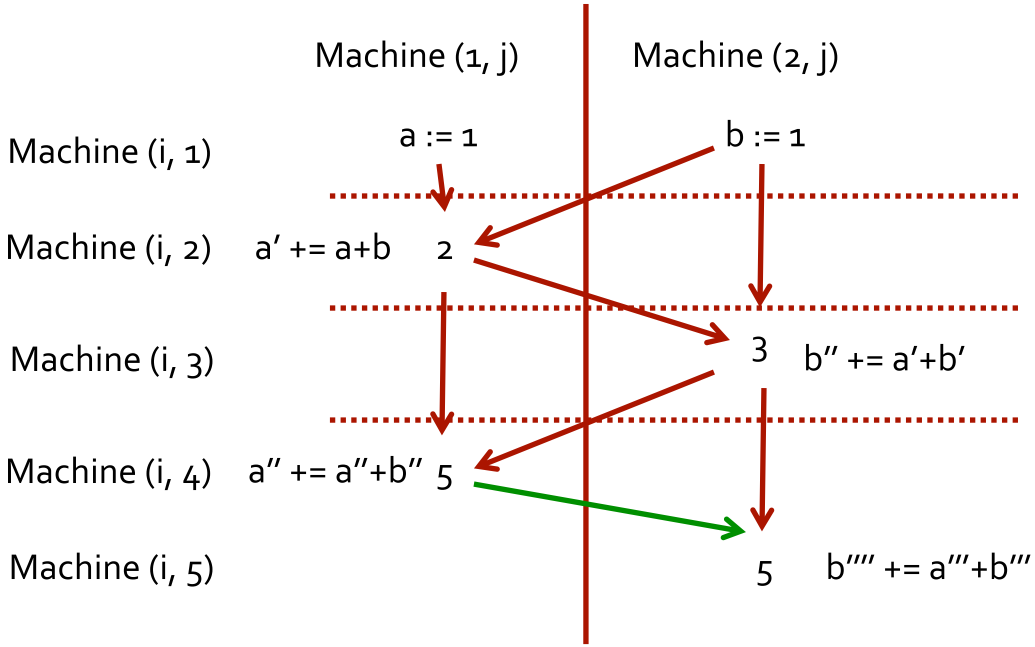
- Repeatedly compute values.
- Each msgs has a (creation) epoch timestamp
- Multiple msg can share timestamp.

- Works in this case!
- Computes only sums of two objects. We know when we have received all the msgs we need to make progress!

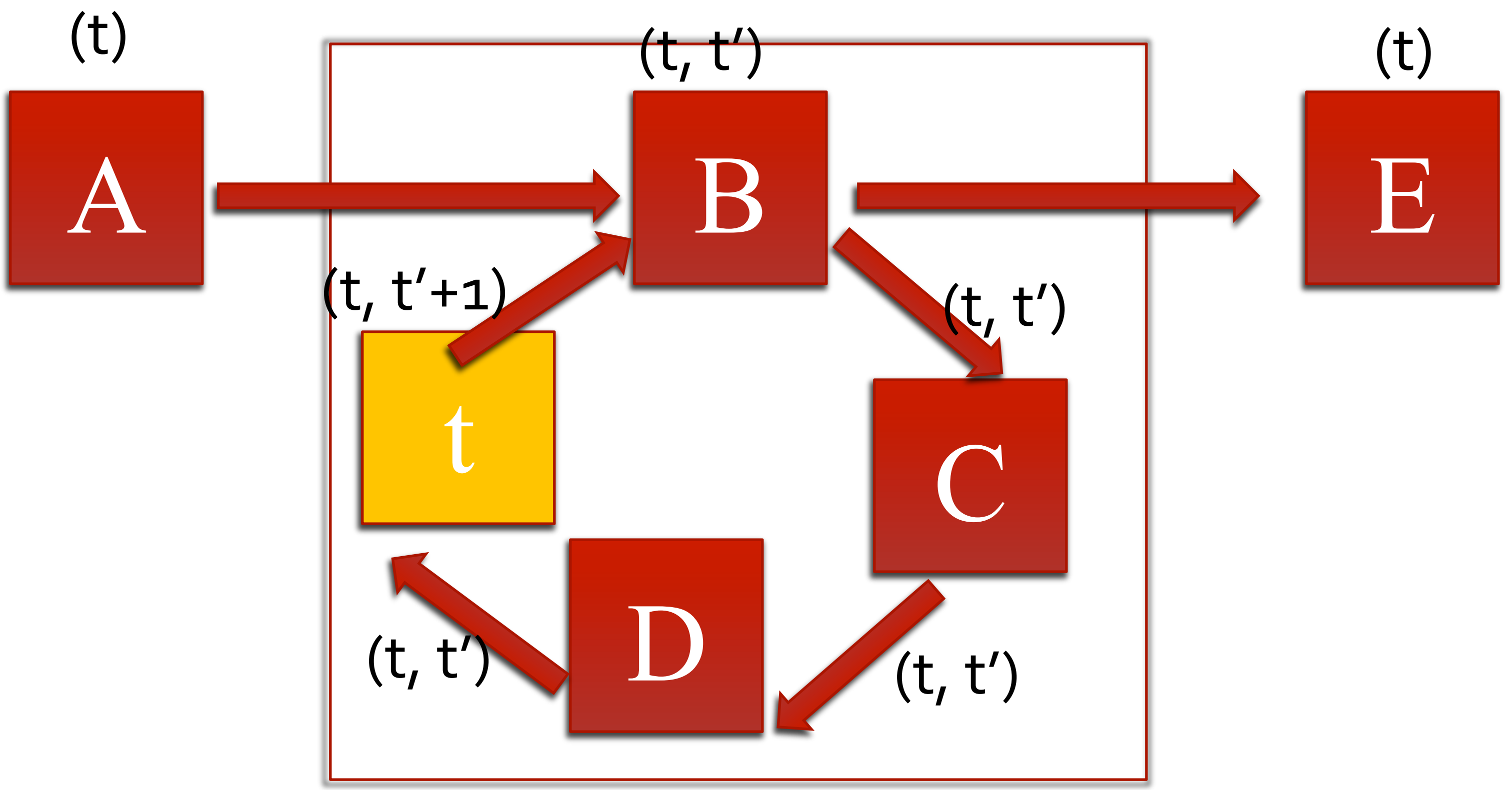


Does streaming/message passing defeat the 2PC lower bound?

- Repeatedly compute values.
- Each msgs has a (creation) epoch timestamp
- Multiple msg can share timestamp.
- Notify when no more messages of a particular ts are to come from a sender.
- Requires to wait for notify() from all sources.
- Synch again!
- If there is a cyclical dep (same vals read as written), 2PC is back!



Streaming+Iteration: Structured time [Naiad]



Paths to (real-time) performance

- Small data (seriously!)
- Incrementalization (online/anytime)
- Parallelization
- **Specialization**
  - Hardware: ASIC - conflict with economies of scale; FPGA
  - Software: compilation

Compilation

- "Lampson's law": "*Every" problem in computer science can be solved by another level of indirection.*
  - Modularization, layering; abstract data types – clean, manageable software design by abstraction
- The reverse is also ~ true: "*Every" performance problem can be solved by* ***removing*** *a level of indirection.*
  - Compilers are software for automatically eliminating levels of indirection/abstraction.

Eliminating indirection by partial evaluation

```
class Record(fields: Array[String], schema: Array[String]) {
  def apply(key: String) = fields(schema indexOf key)
}

def processCSV() = {
  val lines = FileReader("data.csv")
  val schema = lines.next().split(",")


    while (lines.hasNext) {
      val fields = lines.next().split(",")
      val record = new Record(fields, schema)
      if (record("Flag") == "yes")
          println(record("Name"))
    }

}
```

```
Name, Value, Flag
A, 7, no
B, 2, yes
```

Eliminating indirection by partial evaluation

```
class Record(fields: Rep[Array[String]], schema: Array[String]) {
  def apply(key: String) = fields(schema indexOf key)
}

def processCSV() = {
  val lines = FileReader("data.csv")
  val schema = lines.next().split(",")
  run {
    val lines1 = staticData(lines);
    while (lines1.hasNext) {
      val fields = lines1.next().split(",")
      val record = new Record(fields, schema)
      if (record("Flag") == "yes")
          println(record("Name"))
    }
  }
}
```

```
Name, Value, Flag
A, 7, no
B, 2, yes
```

Eliminating indirection by partial evaluation

```
def processCSV() = {
  val lines = FileReader("data.csv")
  val schema = lines.next().split(",")
  // 'run' block: dynamically specialized wrt schema
    val lines1 = lines;
    while (lines1.hasNext) {
      val fields = lines1.next().split(",")

      if (fields(2) == "yes")
          println(fields(0))
    }

}
```

```
Name, Value, Flag
A, 7, no
B, 2, yes
```

Eliminating indirection by partial evaluation

- Removed the schema lookup and the record abstraction.
- >10x speedup in Java/Scala!
  - Object creation/destruction (boxing/unboxing) in JVM.
  - schema.indexof is costly.

```
class Record(fields: Rep[Array[String]], schema: Array[String]) {
  def apply(key: String) = fields(schema indexOf key)
}

def processCSV() = {
  val lines = FileReader("data.csv")
  val schema = lines.next().split(",")
  run {
    val lines1 = staticData(lines);
    while (lines1.hasNext) {
      val fields = lines1.next().split(",")
      val record = new Record(fields, schema)
      if (record("Flag") == "yes")
          println(record("Name"))
    }
  }
}
```

```
def processCSV() = {
  val lines = FileReader("data.csv")
  val schema = lines.next().split(",")

    val lines1 = lines;
    while (lines1.hasNext) {
      val fields = lines1.next().split(",")

      if (fields(2) == "yes")
          println(fields(0))
    }

}
```

Domain-specific opt. examples

- Deforestation
  - myCollection.map(f).map(g) => myCollection.map(f; g)

- Data structure specialization
  - Map relation, matrix, etc. abstraction to implementation.
  - E.g. fixed-sized size array, hash table, tree, queue.
  - Good data structure implementations are imperative and low-level. Hard for automatic program analysis!
  - General-purpose compilers can't do it.

Compilation

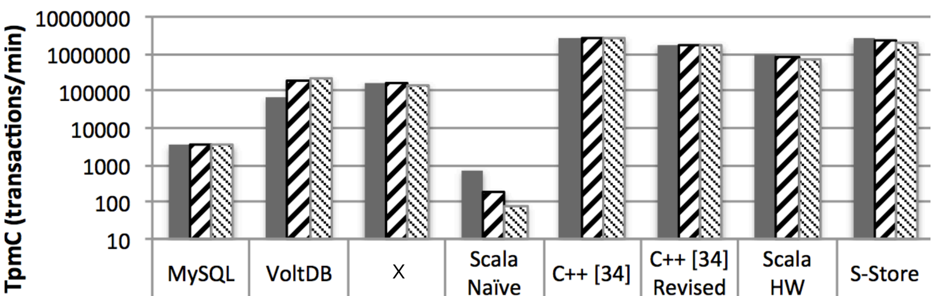
- Our goals:
  - Leverage the full power of software specialization…
  - But make it easy to do: high productivity.
- Develop analytics engines in a high level programming language.
- Have the compiler produce highly optimized code.
- This requires a compiler to be enriched with the knowledge of an expert systems programmer.
- Needs a suitable compiler framework.

The S-Store OLTP System

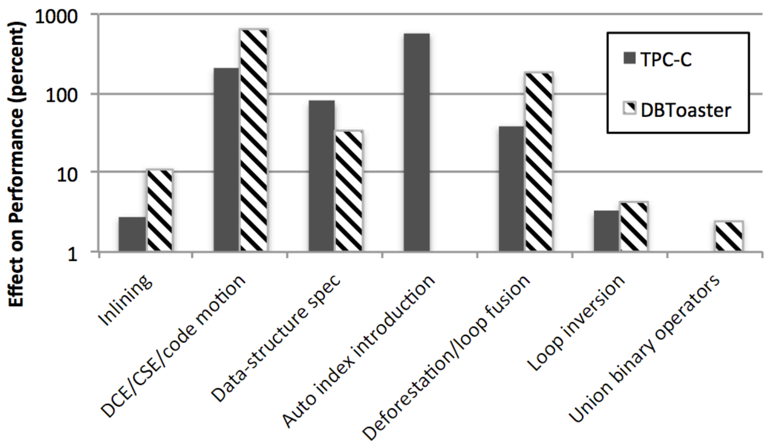
Joint work with Mohammad Dashti, Thierry Coppey, Vojin Jovanovich, EPFL DATA Lab

- A main-mem OLTP system that compiles transaction programs.
- Built in Scala.
- Compiler = LMS + DS opt.

TPC-C transaction	Version	Mutable record manipulation	Inlining	Common subexpr. elim.	Data structure specialization	Automatic index introduction	Deforestation	Loop inversion
NewOrder	S-Store	✓	✓	✓	✓	✓	✓	✓
	Hand-written	✓	✗	–	✓	✗	✓	✗
Payment	S-Store	✓	✓	✓	✓	✓	✓	✗
	Hand-written	✓	✗	–	✓	✗	✓	✗
OrderStatus	S-Store	✗	✓	✓	✓	✓	✓	✗
	Hand-written	✗	✗	–	✓	✗	✗	✗
Delivery	S-Store	✓	✓	✓	✓	✓	✓	✓
	Hand-written	✓	✗	–	✓	✗	✓	✗
StockLevel	S-Store	✗	✓	✓	✓	✓	✓	✓
	Hand-written	✗	✗	–	✓	✗	✓	✗



	MySQL	VoltDB	X	Scala Naïve	C++ [34]	C++ [34] Revised	Scala HW	S-Store
W=1	3512	70349	167007	747	2663092	1795411	1017520	2492911
W=5	3384	201142	160112	178	2673561	1795910	846982	2215348
W=10	3377	226710	149402	78	2670703	1795614	709949	2101276



The LegoBase System

Y. Klonatos, **CK**, T. Rompf, H. Chafi, “LegoBase: Building Efficient Query Engines in a High-Level Language”, SIGMOD 2014.

- An OLAP execution engine. Joint work with Oracle Labs.
- Takes Oracle TimesTen (main mem eng.) query plans (supports all plans).
- Entire engine built in Scala.
- Compiled using LMS.
  - Uses cutting-edge compiler technology: generative metaprogramming/staging; just-in-time compilation.
- Compiles to C or Scala. Optionally deployed on LLVM.

LegoBase: Effort vs. Speedup

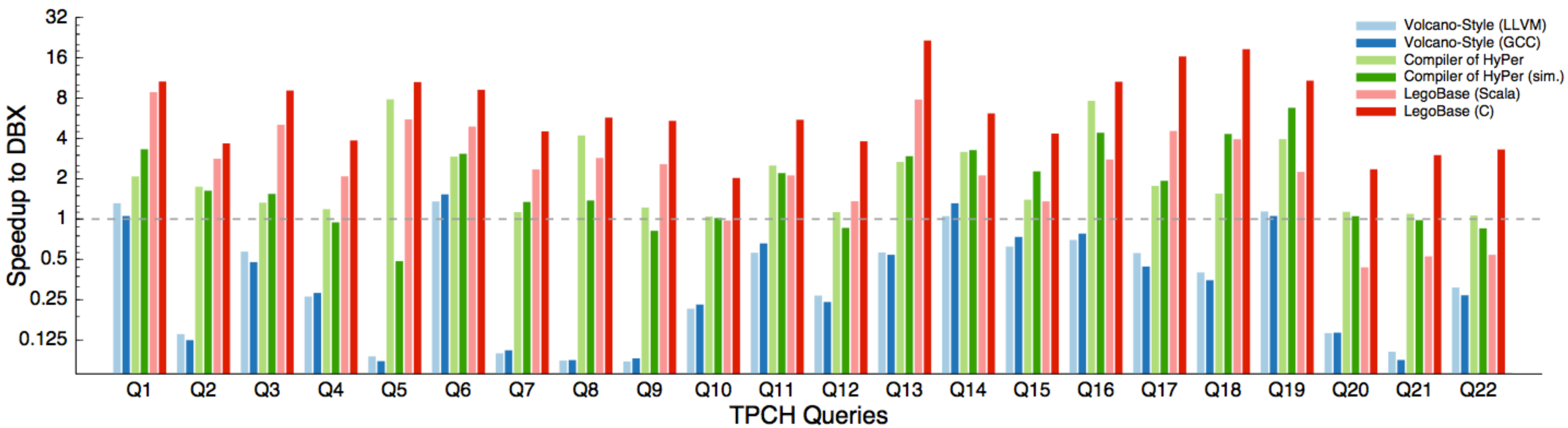
	Coding Effort	Scala LOC	Average Speedup
Operator Inlining	–	0	2.07×
Push Engine Opt.	1 Week	∼400 [6]	2.26×
Data Structure Opt.	4 Days	259	2.16×
Change Data Layout	3 Days	102	1.81×
Other Misc. Opt.	3 Days	124	–[10]
LegoBase Operators	1 Month	428	–
LMS Modifications	2 Months	3953	–
Various Utilities	1 Week	538	–
Total	∼4 Months	5831	7.7×

LegoBase on TPC-H

- Average speedup to DBX: 7.7x
  - Cache locality: 30% avg. improvement
  - Branch prediction: 154% avg. improvement

- Avg. speedup of C vs. Scala: 2.5x. Scala:
  - 30-140% more branch mispredictions
  - 10-180% more cache misses.
  - 5.5x more instructions executed.

- Same optimizations can't be obtained by compilation of low-level code – too nonlocal!



Summary

- Real-time analytics has to embrace incrementality!
- Incrementalization can give asymptotic efficiency improvements. By many orders of magnitude in practice.
- Incrementalization lowers the code => compilation.
- Distributed computation at low latencies has fundamental limits.
  - Interesting tradeoffs; Special cases (embarassing parallelism)
  - Spark is by far not the final word in Big Data infrastructure:
  - Very hard systems problems, huge design space to explore.