CPT-S 415

Big Data

Yinghui Wu

EME B45

CPT-S 415 Big Data

Big Data: Theory and Practice

- Theory
  - Tractability revisited for querying big data
  - Parallel scalability
  - Bounded evaluability
- Techniques
  - Parallel algorithms
  - Bounded evaluability and access constraints
  - Query-preserving compression
  - Query answering using views
  - Bounded incremental query processing

Search Big Data: Theory & Practice

Fundamental question

To query big data, we have to determine whether it is feasible at all.

For a class $\mathcal{Q}$ of queries, can we find an algorithm $\mathcal{T}$ such that given any Q in $\mathcal{Q}$ and any big dataset D, $\mathcal{T}$ efficiently computes the answers Q(D) of Q in D within our available resources?

Is this feasible or not for $\mathcal{Q}$?

- ✓ Tractability revised for querying big data
- ✓ Parallel scalability
- ✓ Bounded evaluability

New theory for querying big data

BD-tractability

The good, the bad and the ugly

- Traditional computational complexity theory of almost 50 years:
  - The good: polynomial time computable (PTIME)
  - The bad: NP-hard (intractable)
  - The ugly: PSPACE-hard, EXPTIME-hard, undecidable…

What happens when it comes to big data?

Using SSD of 6G/s, a linear scan of a data set ***D*** would take

- 1.9 days when ***D*** is of 1PB (10^{15}B)
- 5.28 years when ***D*** is of 1EB (10^{18}B)

O(n) time is already beyond reach on big data in practice!

Polynomial time queries become intractable on big data!

Complexity classes within P

Polynomial time algorithms are no longer tractable on big data.
So we may consider "smaller" complexity classes

✓ NC (Nick's class): highly parallel feasible

parallel $\log^k(n)$

- parallel polylog time
- polynomially many processors

as hard as P = NP

BIG open: P = NC?

✓ L: O(log n) space

✓ NL: nondeterministic O(log n) space

✓ polylog-space: *$\log^k(n)$* space

$$L \subseteq NL \subseteq \text{polylog-space} \subset P, \quad NC \subseteq P$$

Too restrictive to include practical queries feasible on big data

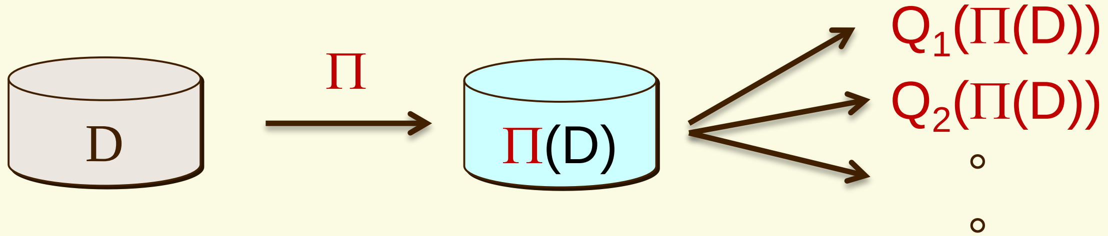
Tractability revisited for queries on big data

A class **Q** of queries is BD-tractable if there exists a PTIME preprocessing function Π such that

- ✓ for any database D on which queries of **Q** are defined,

  D' = Π(D)

- ✓ for all queries Q in **Q** defined on D, Q(D) can be computed by **evaluating Q on D'** in parallel polylog time (NC)



Does it work? If a linear scan of D could be done in log(|D|) time:

- ✓ 15 seconds when D is of 1 PB instead of 1.99 days
- ✓ 18 seconds when D is of 1 EB rather than 5.28 years

BD-tractable queries are feasible on big data

BD-tractable queries

A class **Q** of queries is BD-tractable if there exists a PTIME preprocessing function Π such that

✓ for any database D on which queries of **Q** are defined,

$D' = \Pi(D)$

✓ for all queries Q in **Q** defined on D, Q(D) can be computed by **evaluating Q on D'** in parallel polylog time (NC)

Preprocessing: a common practice of database people

✓ one-time process, offline, once for all queries in **Q**

✓ indices, compression, views, incremental computation, …

not necessarily reduce the size of D

$BDTQ_0$: the set of all BD-tractable query classes

What query classes are BD-tractable?

Boolean selection queries

- ✓ Input: A dataset D
- ✓ Query: Does there exist a tuple t in D such that t[A] = c?

Build a B^+-tree on the A-column values in D. Then all such selection queries can be answered in O(log(|D|)) time.

Graph reachability queries

- ✓ Input: A directed graph G
- ✓ Query: Does there exist a path from node s to t in G?

What else?

Relational algebra + set recursion on ordered relational databases

D. Suciu and V. Tannen: A query language for NC, PODS 1994

Some natural query classes are BD-tractable

Deal with queries that are *not* BD-tractable

Many query classes are not BD-tractable.

Breadth-Depth Search (BDS)

- ✓ Input: An unordered graph $G = (V, E)$ with a numbering on its nodes
- ✓ Qu … search of G?

Starts at a node s, and visits all its children, pushing them onto a stack in the reverse order induced by the vertex numbering. After all of s' children are visited, it continues with the node on the top of the stack, which plays the role of s

Is this problem (query class) BD-tractable?

No. Preprocessing does not help us answer such queries.

Fundamental problems for BD-tractability

BD-tractable queries help practitioners determine what query classes are tractable on big data.

Are we done yet?

No, a number of questions in connection w

Why do we need reduction?

✓ Reductions: how to trans   in the class that we know how to solve   ctable?

Analogous to our familiar NP-complete problems

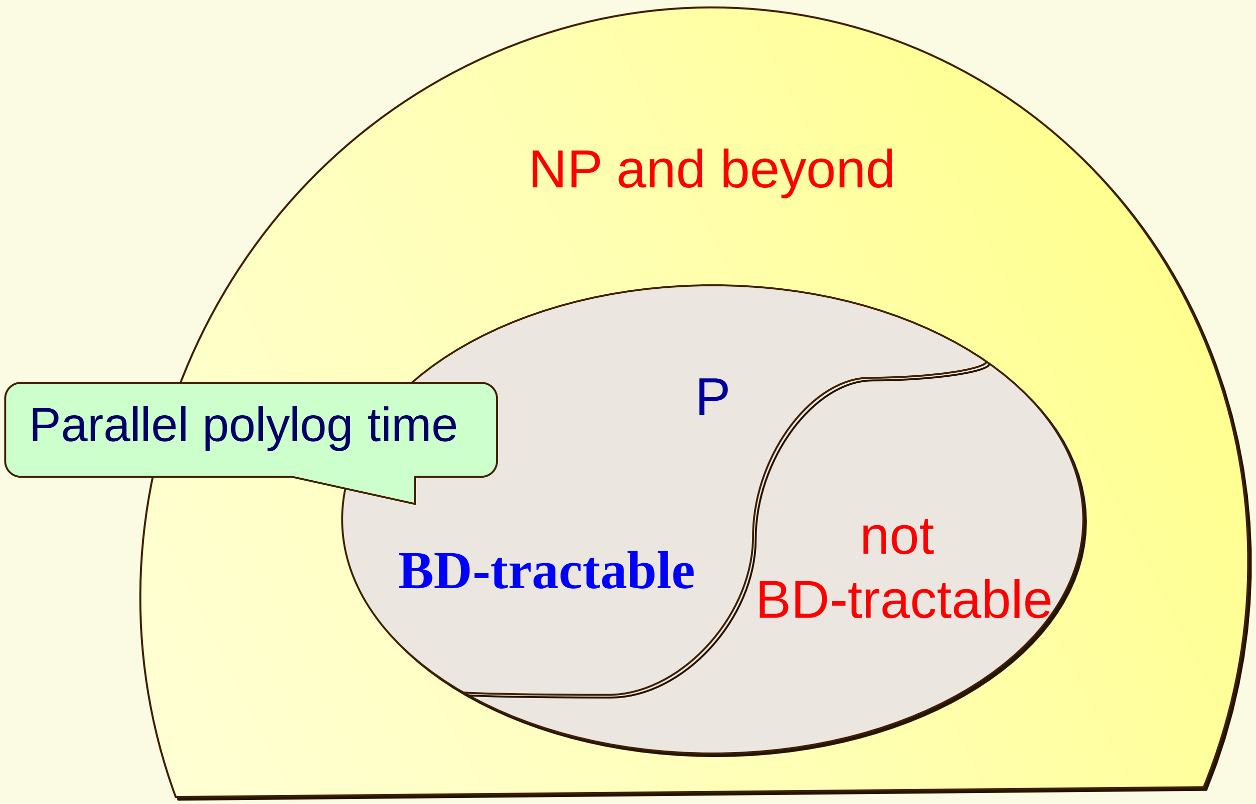
✓ Complete problems: Is there a natural problem (a class of queries) that is the hardest one in the complexity class? A problem to which all problems in the complexity class can be reduced

✓ How large is BDTQ? Compared to P?

Name one NP-complete problem that you know

Fundamental to any complexity classes: P, NP, …

Polynomial hierarchy revised



Tractability revised for querying big data

What can we get from BD-tractability?

BDTQ

Guidelines for the following.

- ✓ What query classes are feasible on big data?
- ✓ What query classes can be made feasible to answer on big data?
- ✓ How to determine whether it is feasible to answer a class Q of queries on big data?

  Reduce Q to a complete problem Q_c for BDTQ
- ✓ If so, how to answer queries in Q?
  - Compose the reduction and the algorithm for answering queries of Q_c

Why we need to study theory for querying big data

Parallel scalability

Parallel query answering

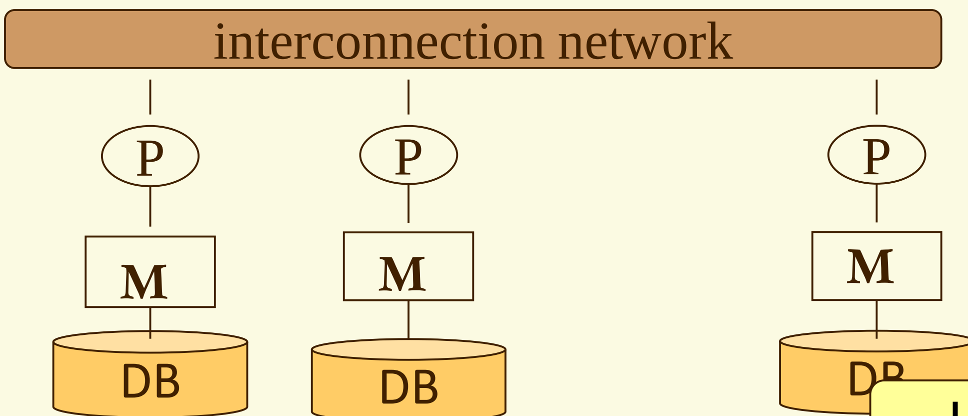
BD-tractability is hard to achieve.

Parallel processing is widely used, given more resources

Using 10000 SSD of 6G/s, a linear scan of *D* might take:

- ✓ 1.9 days/10000 = 16 seconds when *D* is of 1PB (10^{15}B)
- ✓ 5.28 years/10000 = 4.63 days when *D* is of 1EB (10^{18}B)

Only ideally!



10,000 processors

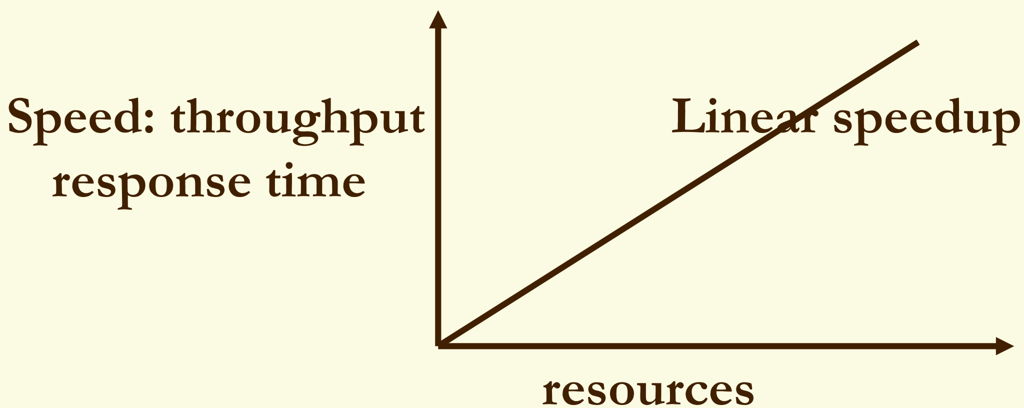
How to define “better”?

Parallel scalable: the more processors, the “better”?

Degree of parallelism -- speedup

Speedup: for a given task, TS/TL,

- ✓ TS: time taken by a traditional DBMS
- ✓ TL: time taken by a parallel system with more resources
- ✓ TS/TL: more sources mean proportionally less time for a task
- ✓ Linear speedup: the speedup is N while the parallel system has N times resources of the traditional system

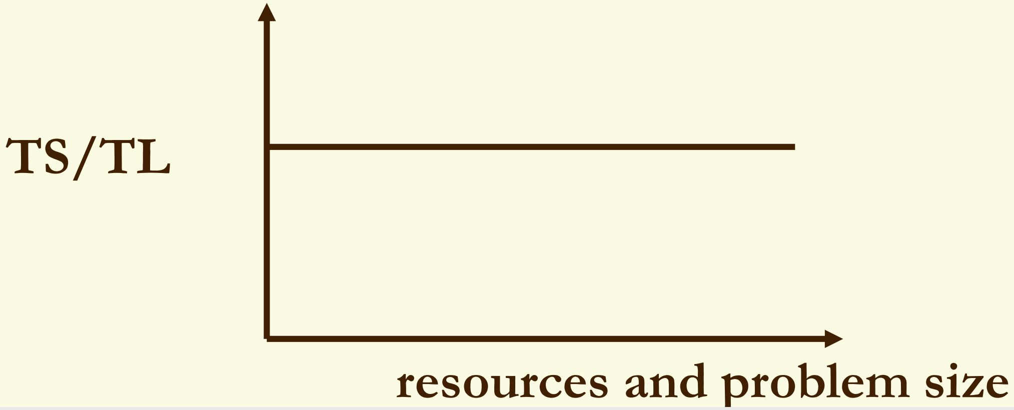


Question: can we do better than linear speedup?

Degree of parallelism -- scaleup

Scaleup: TS/TL

- A task Q, and a task Q_N, N times bigger than Q
- A DBMS M_S, and a parallel DBMS M_L,N times larger
- TS: time taken by M_S to execute Q
- TL: time taken by M_L to execute Q_N
- Linear scaleup: if TL = TS, i.e., the time is constant if the resource increases in proportion to increase in problem size



Question: can we do better than linear scaleup?

Better than linear scaleup/speedup?

NO, even hard to achieve linear speedup/scaleup!

- ✓ Startup costs: initializing each process
- ✓ Interference: competing for shared resources (network, disk, memory or even locks)

Think of blocking in MapReduce

- ✓ Skew: it is difficult to divide a task into exactly equal-sized parts; the response time is determined by the largest part

In the real world, linear scaleup is too ideal to get!
A weaker criterion: the more processors are available, the less response time it takes.

Linear speedup is the best we can hope for -- optimal!

Parallel query answering

Given a big dataset D, and n processors D1, …, Dn

- D is partitioned into fragments (D1, …, Dn)
- D is distributed to n processors: Di is stored at Si

Parallel query answering

- Input: D = (D1, …, Dn), distributed to (S1, …, Sn), and a query Q
- Output: Q(D), the answer to Q in D

Performance

- Response time (aka parallel computation cost): Interval from the time when Q is submitted to the time when Q(D) is returned
- Data shipment (aka network traffic): the total amount of data shipped between different processors, as messages

Performance guarantees: bounds on response time and data shipment

Parallel scalability

- Input: D = (D1, …, Dn), distributed to (S1, …, Sn), and a query Q
- Output: Q(D), the answer to Q in D

Complexity

- t(|D|, |Q|): the time taken by a sequential algorithm with a single processor
- T(|D|, |Q|, n): the time taken by a parallel algorithm with n processors
- Parallel scalable: if

$$T(|D|, |Q|, n) = O(t(|D|, |Q|)/n) + O((n + |Q|)^k)$$

Polynomial reduction (including the cost of data shipment, k is a constant)

When D is big, we can still query D by adding more processors if we can afford them

A distributed algorithm is useful if it is parallel scalable

linear scalability

An algorithm T for answering a class Q of queries

- ✓ Input: D = (D1, …, Dn), distributed to (S1, …, Sn), and a query Q
- ✓ Output: Q(D), the answer to Q in D

The more processors, the less response time

Algorithm T is linear scalable in

- ✓ computation if its parallel complexity is a function of |Q| and |D|/n, and in
- ✓ data shipment if the total amount of data shipped is a function of |Q| and n

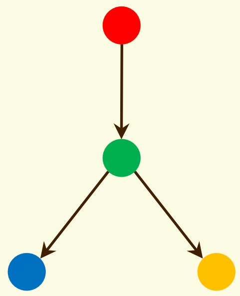
Independent of the size |D| of big D

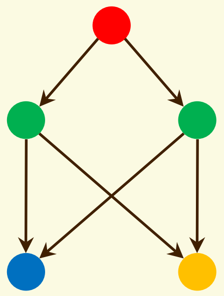
Is it always possible?

Querying big data by adding more processors

Graph pattern matching via graph simulation

- Input: a graph pattern graph Q and a graph G
- Output: Q(G) is a binary relation S on the nodes of Q and G
  - each node *u* in Q is mapped to a node *v* in G, *such that* $(u, v) \in S$
  - for each $(u,v) \in S$, each edge (u,u') in Q is mapped to an edge (v, v') in G, *such that* $(u',v') \in S$





Parallel scalable?

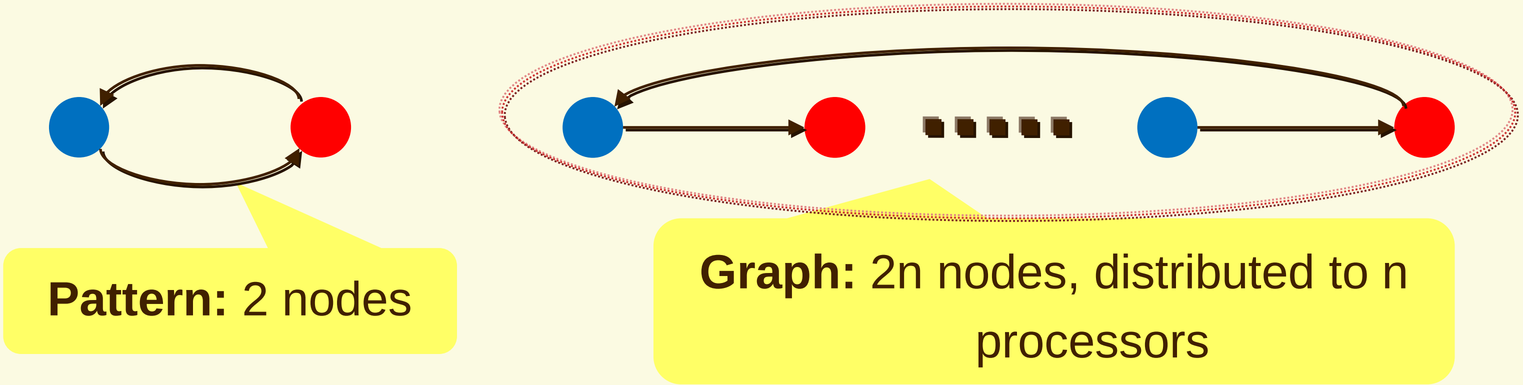
O((| V | + | VQ |) (| E | + | EQ|)) time

Impossibility

There exists NO algorithm for distributed graph simulation that is parallel scalable in either

- ✓ computation, or
- ✓ data shipment

Why?



Possibility: when G is a tree, parallel scalable in both response time and data shipment

Nontrivial to develop parallel scalable algorithms

Weak parallel scalability

Algorithm T is weakly parallel scalable in

- ✓ computation if its parallel computation cost is a function of |Q| |G|/n and $|E_f|$, and in
- ✓ data shipment if the total amount of data shipped is a function of |Q| and $|E_f|$

edges across different fragments

Rational: we can partition G as preprocessing, such that

- ✓ $|E_f|$ is minimized (an NP-complete problem, but there are effective heuristic algorithms), and
- ✓ When G grows, $|E_f|$ does not increase substantially

The cost is not a function of |G| in practice

Doable: graph simulation is weakly parallel scalable

MRC: Scalability of MapReduce algorithms

Characterize scalable MapReduce algorithms in terms of disk usage, memory usage, communication cost, CPU cost and rounds.

For a constant $\varepsilon > 0$ and a data set D, $|D|^{1-\varepsilon}$ machines, a MapReduce algorithm is in MRC if

- ✓ Disk: each machine uses $O(|D|^{1-\varepsilon})$ disk, $O(|D|^{2-2\varepsilon})$ in total.
- ✓ Memory: each machine uses $O(|D|^{1-\varepsilon})$ memory, $O(|D|^{2-2\varepsilon})$ in total.
- ✓ Data shipment: in each round, each machine sends or receives $O(|D|^{1-\varepsilon})$ amount of data, $O(|D|^{2-2\varepsilon})$ in total.
- ✓ CPU: in each round, each machine takes polynomial time in |D|.
- ✓ The number of rounds: polylog in |D|, that is, *$\log^k(|D|)$*

the larger D is, the more processors

The response time is still a polynomial in |D|

MMC: a revision of MRC

For a constant $\varepsilon > 0$ and a data set D, n machines, a MapReduce algorithm is in MMC if

- ✓ Disk: each machine uses O(|D|/n) disk, O(|D|) in total.
- ✓ Memory: each machine uses O(|D|/n) memory, O(|D|) in total.
- ✓ Data shipment: in each round, each machine sends or receives O(|D|/n) amount of data, O(|D|) in total.
- ✓ CPU: in each round, each machine takes O(Ts/n) time, where Ts is the time to solve the problem in a single machine.
- ✓ The number of rounds: *O(1), a constant number of rounds.*

Speedup: O(Ts/n) time
the more machines are used, the less time is taken

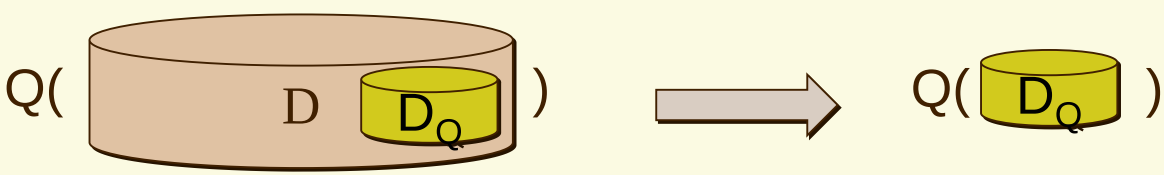
Compared with BD-tractable and parallel scalability

Bounded evaluability

Scale independence

- Input: A class $\mathcal{Q}$ of queries
- Question: Can we find, for any query $Q \in \mathcal{Q}$ and any (possibly big) dataset D, a fraction D_Q of D such that
  - $|D_Q| \leq M$, and
  - $Q(D) = Q(D_Q)$?

Independent of the size of D



Particularly useful for

- A single dataset D, eg, the social graph of Facebook
- Minimum D_Q – the necessary amount of data for answering Q

Making the cost of computing Q(D) independent of |D|!

Facebook: Graph Search

- Find me restaurants in New York my friends have been to in 2013

```
select   rid
from     friend(pid1, pid2), person(pid, name, city),
          dine(pid, rid, dd, mm, yy)
where    pid1 = p0  and  pid2 = person.pid   and
          pid2 = dine.pid and city = NYC  and yy = 2013
```

Access constraints (real-life limits)

- Facebook: 5000 friends per person
- Each year has at most 366 days
- Each person dines at most once per day
- pid is a key for relation person

How many tuples do we need to access?

Bounded query evaluation

- Find me restaurants in New York my friends have been to in 2013

```
select   rid
from     friend(pid1, pid2), person(pid, name, city),
          dine(pid, rid, dd, mm, yy)
where    pid1 = p0  and  pid2 = person.pid   and
          pid2 = dine.pid and city = NYC  and yy = 2013
```

A query plan

- Fetch 5000 pid's for friends of p0 -- 5000 friends per person
- For each pid, check whether she lives in NYC – 5000 person tuples
- For each pid living in NY[illegible] in 2013 – 5000 * 366 tuples

Contrast to Facebook : more than 1.26 billion nodes, and over 140 billion links

*Accessing 5000 + 5000 + 5000 * 366 tuples in total*

Access constraints

On a relation schema R: $X \rightarrow (Y, N)$

Combining cardinality constraints and index

- ✓ X, Y: sets of attributes of R
- ✓ for any X-value, there exist at most N distinct Y values
- ✓ Index on X for Y: given an X value, find relevant Y values

Examples

- ✓ friend(pid1, pid2): pid1 $\rightarrow$ (pid2, 5000)   5000 friends per person
- ✓ dine(pid, rid, dd, mm, yy): pid, yy $\rightarrow$ (rid, 366)   each year has at most 366 days and each person dines at most once per day
- ✓ person(pid, name, city): pid $\rightarrow$ (city, 1)   pid is a key for relation person

Access schema: A set of access constraints

Finding access schema

On a relation schema R: $X \rightarrow (Y, N)$

- ✓ Functional dependencies $X \rightarrow Y$: $X \rightarrow (Y, 1)$
- ✓ Keys X: $X \rightarrow (R, 1)$
- ✓ Domain constraints, e.g., each year has at most 366 days
- ✓ Real-life bounds: 5000 friends per person (Facebook)
- ✓ The semantics of real-life data, e.g., accidents in the UK from 1975-2005

How to find these?

  - dd, mm, yy $\rightarrow$ (aid, 610) at most 610 accidents in a day
  - aid $\rightarrow$ (vid, 192) at most 192 vehicles in an accident

- ✓ Discovery: extension of function dependency discovery

Bounded evaluability: only a small number of access constraints

Bounded queries

- Input: A class $\mathcal{Q}$ of queries, an access schema $\mathcal{A}$
- Question: Can we find by using $\mathcal{A}$, for any query $Q \in \mathcal{Q}$ and any (possibly big) dataset D, a fraction D_Q of D such that
  - $|D_Q| \leq M$, and
  - $Q(D) = Q(D_Q)$?

Examples

- The graph search query at Facebook
- All Boolean conjunctive queries are bounded
  - Boolean: Q(D) is true or false
  - Conjunctive: SPC, selection, projectio

But how to find D_Q?

Boundedness: to decide whether it is possible to compute Q(D) by accessing a bounded amount of data at all

Boundedly evaluable queries

- Input: A class $\mathcal{Q}$ of queries, an access schema $\mathcal{A}$
- Question: Can we find by using $\mathcal{A}$, for any query $Q \in \mathcal{Q}$ and any (possibly big) dataset D, a fraction D_Q of D such that
  - $|D_Q| \leq M$,
  - $Q(D) = Q(D_Q)$, and moreover,
  - *D_Q can be identified in time determined by Q and $\mathcal{A}$*?

Examples

- The graph search query at Facebook
- All Boolean conjunctive queries are bounded but are not necessarily effectively bounded!

If Q is boundedly evaluable, for any big D, we can efficiently compute Q(D) by accessing a bounded amount of data!

Deciding bounded evaluability

- ✓ Input: A query Q, an access schema $\mathcal{A}$
- ✓ Question: Is Q boundedly evaluable under $\mathcal{A}$?

Yes. doable

- ✓ Conjunctive queries (SPC) with restricted query plans:
  - Characterization: sound and complete rules
  - PTIME algorithms for checking effective boundedness and for generating query plans, in |Q| and |$\mathcal{A}$|
- ✓ Relational algebra (SQL): undecidable
  - Special cases
  - Sufficient conditions

What can we do?

Parameterized queries in recommendation systems, even SQL

Many practical queries are in fact boundedly evaluable!

Techniques for querying big data: review

An approach to querying big data

Given a query Q, an access schema $\mathcal{A}$ and a big dataset D

1. Decide whether Q is effectively bounded under $\mathcal{A}$
2. If so, generate a bounded query plan for Q
3. Otherwise, do one of the following:

① f Q,

② r?)

③

- ✓ *77% of conjunctive queries are boundedly evaluable*
- ✓ *Efficiency: 9 seconds vs. 14 hours of MySQL*
- ✓ *60% of graph pattern queries are boundedly evaluable (via subgraph isomorphism)*
- ✓ *Improvement: 4 orders of magnitudes*

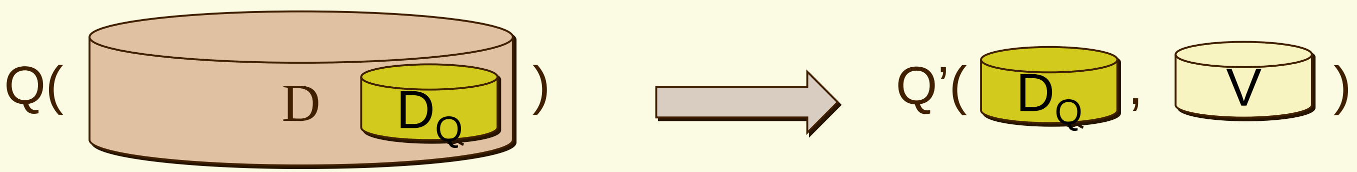
Very effective for conjunctive queries

Bounded evaluability using views

- Input: A class $\mathcal{Q}$ of queries, a set of views $\mathcal{V}$, an access schema $\mathcal{A}$
- Question: Can we find by using $\mathcal{A}$, for any query $Q \in \mathcal{Q}$ and any (possibly big) dataset D, a fraction D_Q of D such that
  - $|D_Q| \leq M$,
  - a rewriting Q' of Q using $\mathcal{V}$,
  - $Q(D) = Q'(D_Q, \mathcal{V}(D))$, and
  - D_Q can be identified in time determined by Q, $\mathcal{V}$, and $\mathcal{A}$?

access views, and additionally

a bounded amount of data

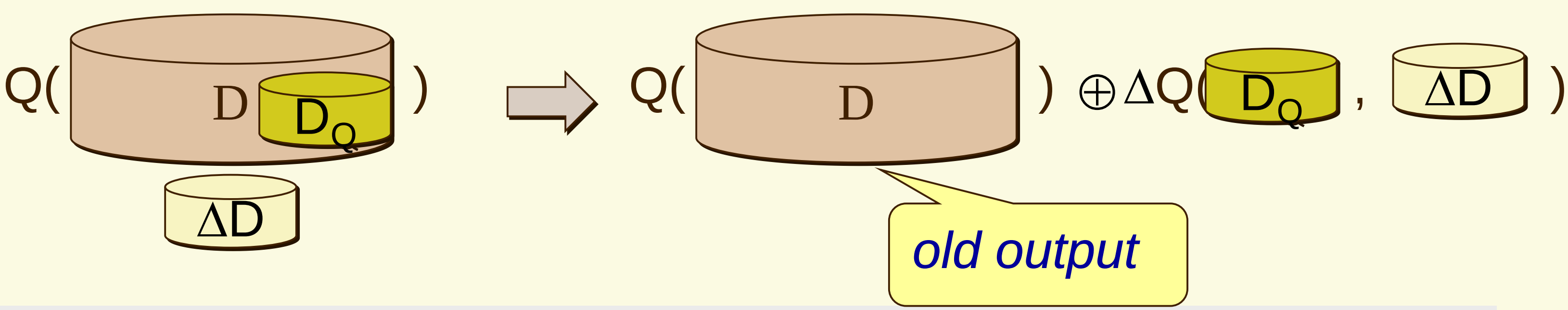


Query Q may not be boundedly evaluable, but may be boundedly evaluable with views!

Incremental bounded evaluability

- Input: A class $\mathcal{Q}$ of queries, an access schema $\mathcal{A}$
- Question: Can we find by using $\mathcal{A}$, for any query $Q \in \mathcal{Q}$, any dataset D, and any changes ΔD to D, a fraction D_Q of D such that
  - $|D_Q| \leq M$,
  - $Q(D \oplus \Delta D) = Q(D) \oplus \Delta Q(\Delta D, D_Q)$, and
  - D_Q can be identified in time determined by Q and $\mathcal{A}$?

access an additional bounded amount of data



Query Q may not be boundedly evaluable, but may be incrementally boundedly evaluable!

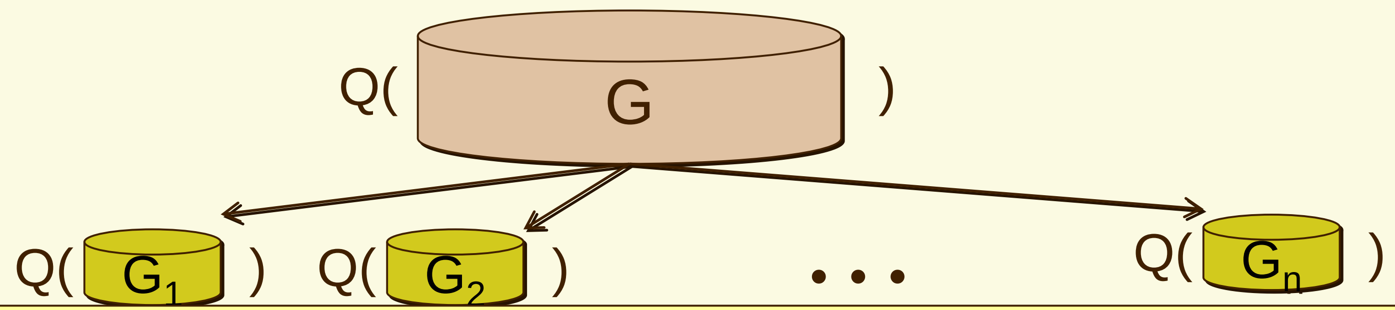
Parallel query processing

manageable sizes

Divide and conquer

- ✓ partition G into fragments $(G_1, \ldots, G_n)$, distributed to various sites
- ✓ upon receiving a query Q,
  - evaluate $Q(G_i)$ *in parallel*
  - collect partial answers at a coordinator site, and assemble them to find the answer $Q(G)$ in the entire G

evaluate Q on smaller G_i



graph pattern matching in GRAPE: 21 times faster than MapReduce

Parallel processing = Partial evaluation + message passing

Query preserving compression

The cost of query processing: **f**(|G|, |Q|)

reduce the parameter?

Query preserving compression <R, P> for a class $\mathcal{L}$ of queries

- ✓ For any data collection G, G_C = R(G)
- ✓ For any Q in $\mathcal{L}$, Q(G) = P(Q, Gc)

Compressing

Post-processing

$$
\begin{tikzcd}
\mathbf{G} \arrow[r, "R"] \arrow[d, "Q"'] & \mathbf{Gc} \arrow[d] \\
Q(\mathbf{G}) & Q(\mathbf{Gc}) \arrow[l, "P"']
\end{tikzcd}
$$

In contrast to lossless

No need to restore the original graph G or decompress the data.

Better compression ratio!

18 times faster on average for reachability queries

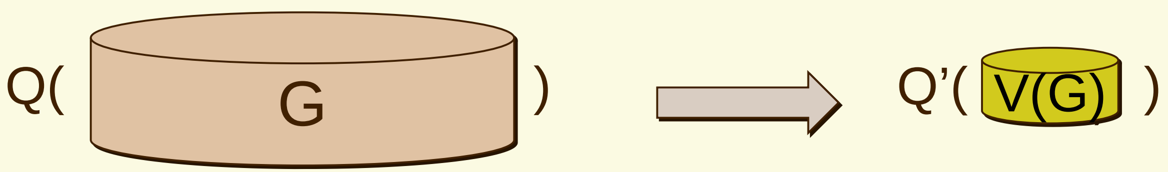
Answering queries using views

The cost of query processing: **f**(|G|, |Q|)

Query answering using views: given a query Q in a language $\mathcal{L}$ and a set $\mathcal{V}$ views, find another query Q' such that

- ✓ Q and Q' are *equivalent*
- ✓ Q' only accesses $\mathcal{V}$(G)

for any G, Q(G) = Q'(G)



$\mathcal{V}$(G) is often much smaller than G (4% -- 12% on real-life data)

Improvement: 31 times faster for graph pattern matching

The complexity is no longer a function of |G|

Incremental query answering

5%/week in Web graphs

- ✓ Real-life data is dynamic – consta
- ✓ Re-compute $Q(G \oplus \Delta G)$ starting from scratch?
- ✓ Changes ΔG are typically small

Compute $Q(G)$ once, and then incrementally maintain it

Old output

Changes to the input

Incre... proce...

- ✓ Input: Q, G, $Q(G)$, ΔG
- ✓ Output: ΔM such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta M$

New output

Changes to the output

When changes ΔG to the data G are small, typically so are the

At least twice as fast for pattern matching for changes up to 10%

Minimizing unnecessary recomputation

A principled approach: Making big data small

- ✓ Bounded evaluable queries
- ✓ Parallel query processing (MapReduce, GRAPE, etc)
- ✓ Query preserving compression: convert big data to small data
- ✓ Query answering using views: make big data small
- ✓ Bounded incremental query answering: depending on the size of the changes rather than the size of the original big data
- ✓ . . .

Including but not limited to graph queries

Yes, MapReduce is useful, but it is not the only way!

Combinations of these can do much better than MapReduce!

Summary and Review

- ✓ What is BD-tractability? Why do we care about it?
- ✓ What is parallel scalability? Name a few parallel scalable algorithms
- ✓ What is bounded evaluability? Why do we want to study it?
- ✓ How to make big data “small”?
- ✓ Is MapReduce the only way for querying big data? Can we do better than it?
- ✓ What is query preserving data compression? Query answering using views? Bounded incremental query answering?
- ✓ If a class of queries is known not to be BD-tractable, how can we process the queries in the context of big data?

Reading

1. M. Arenas, L. E. Bertossi, J. Chomicki: Consistent Query Answers in Inconsistent Databases, PODS 1999. http://web.ing.puc.cl/~marenas/publications/pods99.pdf
2. Indrajit Bhattacharya and Lise Getoor. Collective Entity Resolution in Relational Data. TKDD, 2007. http://linqs.cs.umd.edu/basilic/web/Publications/2007/bhattacharya:tkdd07/bhattacharya-tkdd.pdf
3. P. Li, X. Dong, A. Maurino, and D. Srivastava. Linking Temporal Records. VLDB 2011. http://www.vldb.org/pvldb/vol4/p956-li.pdf
4. W. Fan and F. Geerts， Relative information completeness, PODS, 2009.
5. Y. Cao. W. Fan, and W. Yu. Determining relative accuracy of attributes. SIGMOD 2013.
6. P. Buneman, S. Davidson, W. Fan, C. Hara and W. Tan. Keys for XML. WWW 2001.