LogicBlox Smart database for next-generation applications

Benchmarking @LogicBlox

George Kollias (LogicBlox)

LDBC TUC Meeting, November 14, 2014 - Athens, Greece

Two words about LogicBlox, Inc

Product

- planning
- prediction
- optimization

Customers

- Big retail companies
  - mostly

Other Projects

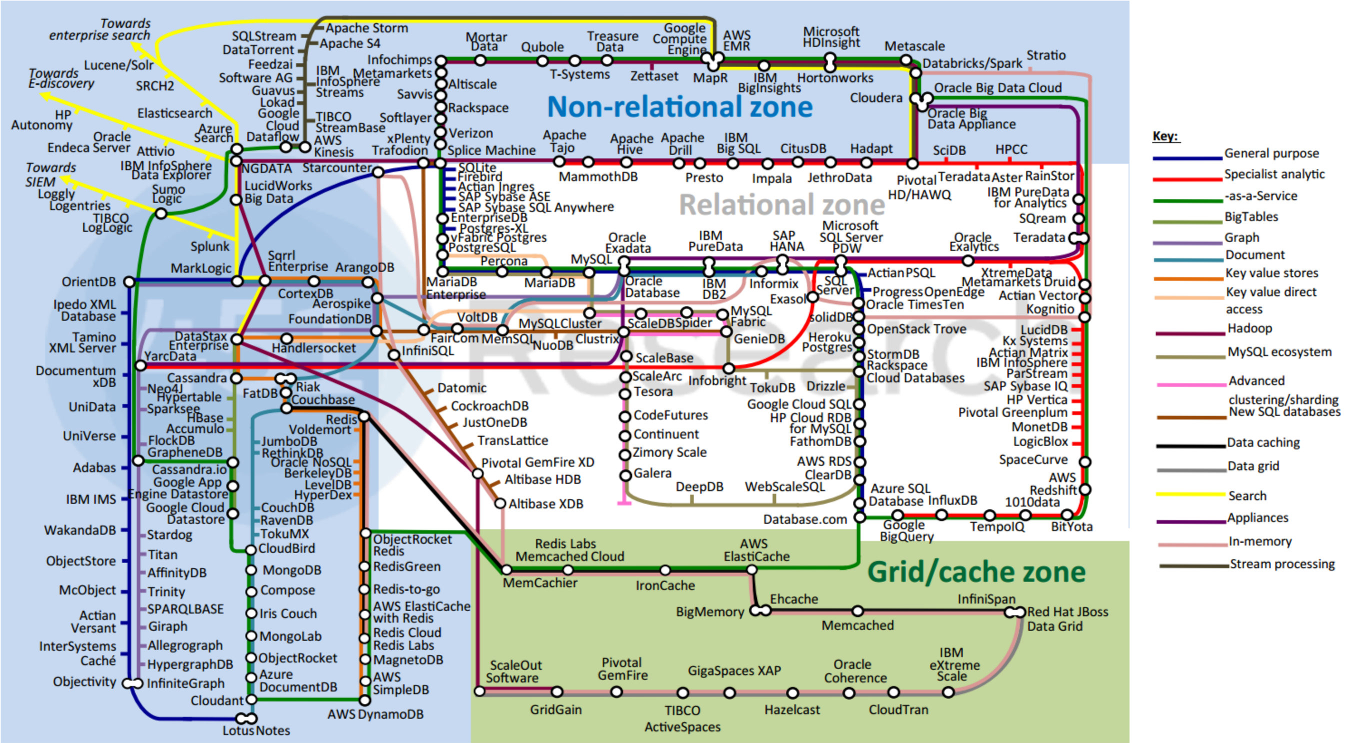
- Darpa, MUSE

Choose many

451 Research

Data Platforms Landscape Map

OCTOBER 2014



Choose many?

- Specialization = result of innovation in DB community during mid-90s
- Example: column stores / MonetDB / analytics
- Stonebraker: “purpose-build, 10x to 100x faster than general purpose”

But

- Plethora of specialized systems = increased costs
- Specialized systems are only worth it if 10x-100x better

Choose less

“While the success of specialized columnar systems seemed to underline the end of the "one system fits all" paradigm as proclaimed by Michael Stonebraker, this issue clearly shows that this is still a debatable proposition. Both the Microsoft SQL Server as well as the Openlink Virtuoso systems show that tight integration of columnar technology in row-based systems is both possible and desirable.”

Peter Boncz
IEEE Computer Society Data Engineering Bulletin
Special Issue on "Column Store Systems"
March 2012

Choose one?

- many specialized technologies put together = “One Size Fits All” system?
  - they still require expertise to tune each of them
- LogicBlox engine designed to be “One Size Fits All” system…
  - <10x worse than any specialized system
- … without many tuning knobs
  - transparent to the user

Underlying technologies

Language

LogiQL

- Datalog variant
  - Declarative

- Recursion
  - Essential for handling complex graph queries
  - Aggregation in Recursion
  - Negation in Recursion

- Integrity constraints

- Event handling (~triggers)

- Incrementally maintained rules (~materialized views)

Join Algorithm(s)

- Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm
  - Todd L. Veldhuizen
  - ICDT '14
  - http://arxiv.org/abs/1210.0481
- Multi-way join
  - Variant of Sort-Merge Join

Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm

Todd L. Veldhuizen
LogicBlox Inc.
Two Midtown Plaza
1349 West Peachtree Street NW
Suite 1880, Atlanta GA 30309
tveldhui@{logicblox.com,acm.org}

ABSTRACT

Recent years have seen exciting developments in join algorithms. In 2008, Atserias, Grohe and Marx (henceforth AGM) proved a tight bound on the maximum result size of a full conjunctive query, given constraints on the input relation sizes. In 2012, Ngo, Porat, Ré and Rudra (henceforth NPRR) devised a join algorithm with worst-case running time proportional to the AGM bound [8]. Our commercial Datalog system LogicBlox employs a novel join algorithm, *leapfrog triejoin*, which compared conspicuously well to the NPRR algorithm in preliminary benchmarks. This spurred us to analyze the complexity of leapfrog triejoin. In this paper we establish that leapfrog triejoin is also worst-case optimal, up to a log factor, in the sense of NPRR. We improve on the results of NPRR by proving that leapfrog triejoin achieves worst-case optimality for finer-grained classes of database instances, such as those defined by constraints on projection cardinalities. We show that NPRR is *not* worst-case optimal for such classes, giving a counterexample where leapfrog triejoin runs in $O(n \log n)$ time, compared to $\Theta(n^{1.375})$ time for NPRR. On a practical note, leapfrog triejoin can be implemented using conventional data structures such as B-trees, and extends naturally to $\exists_1$ queries. We believe our algorithm offers a useful addition to the existing toolbox of join algorithms, being easy to absorb, simple to implement, and having a concise optimality proof.

General Terms

Algorithms,Theory

1. INTRODUCTION

Join processing is a fundamental and comprehensively-studied problem in database systems. Many useful queries can be formulated as one or more *full conjunctive queries*. A full conjunctive query is a conjunctive query with no projections, i.e., every variable in the body appears in the head [3, 1]. As a running example we use the query defined by this Datalog rule:

$$Q(a,b,c) \leftarrow R(a,b), S(b,c), T(a,c). \tag{1}$$

where a, b, c are query variables (for intuition: if $R = S = T$, then Q finds triangles.)

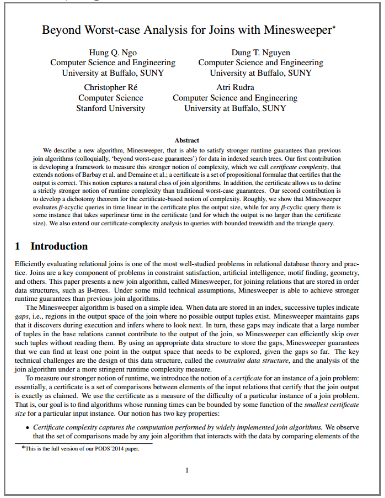
Given constraints on the sizes of the input relations such as $|R| \leq n$, $|S| \leq n$, $|T| \leq n$, what is the maximum possible query result size $|Q|$? This question has practical import, since a tight bound $|Q| \leq Q^*$ implies an $\Omega(Q^*)$ worst-case running time for algorithms answering such queries.

Atserias, Grohe and Marx (henceforth AGM [2]) established a tight bound on the size of Q: the *fractional edge cover* bound (Section 2.2). For the case where $|R| = |S| = |T| = n$, the fractional cover bound yields $|Q| \leq n^{3/2}$. In earlier work, Grohe and Marx [6] gave an algorithm with running time $O(|Q^*|^2 f(n))$, where $f(n)$ is a polynomial determined by the fractional cover bound. In 2012, Ngo, Porat, Ré and Rudra (henceforth NPRR [8]) devised a groundbreaking algorithm with worst-case running time $O(Q^*)$, matching the AGM bound. The algorithm is non-trivial, and its implementation and analysis depend on rather deep machinery developed in the paper.

The NPRR algorithm was brought to our attention by Dung Nguyen, who implemented it experimentally using our framework. LogicBlox uses a novel and hitherto proprietary join algorithm we call *leapfrog triejoin*. Preliminary benchmarks suggested that leapfrog triejoin performed dramatically better than NPRR on some test problems [9]. These benchmark results motivated us to analyze our algorithm, in light of the breakthroughs of NPRR.

Conventional join implementations employ a stable of join operators (see e.g. [5]) which are composed in a tree to produce the query result; this tree is prescribed by a query plan produced by the optimizer. The query plan often relies on producing intermediate results. In contrast, leapfrog triejoin joins all input relations simultaneously without producing any intermediate re-

- Beyond Worst-Case Analysis for Joins with Minesweeper
  - Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, Atri Rudra
  - PODS '14
  - http://arxiv.org/abs/1302.0914
- Multi-way join



Beyond Worst-case Analysis for Joins with Minesweeper*

Hung Q. Ngo
Computer Science and Engineering
University at Buffalo, SUNY

Dung T. Nguyen
Computer Science and Engineering
University at Buffalo, SUNY

Christopher Ré
Computer Science
Stanford University

Atri Rudra
Computer Science and Engineering
University at Buffalo, SUNY

Abstract

We describe a new algorithm, Minesweeper, that is able to satisfy stronger runtime guarantees than previous join algorithms (colloquially, 'beyond worst-case guarantees') for data in indexed search trees. Our first contribution is developing a framework to measure this stronger notion of complexity, which we call *certificate complexity*, that extends notions of Barbay et al. and Demaine et al.; a certificate is a set of propositional formulae that certifies that the output is correct. This notion captures a natural class of join algorithms. In addition, the certificate allows us to define a strictly stronger notion of runtime complexity than traditional worst-case guarantees. Our second contribution is to develop a dichotomy theorem for the certificate-based notion of complexity. Roughly, we show that Minesweeper evaluates β-acyclic queries in time linear in the certificate plus the output size, while for any β-cyclic query there is some instance that takes superlinear time in the certificate (and for which the output is no larger than the certificate size). We also extend our certificate-complexity analysis to queries with bounded treewidth and the triangle query.

1 Introduction

Efficiently evaluating relational joins is one of the most well-studied problems in relational database theory and practice. Joins are a key component of problems in constraint satisfaction, artificial intelligence, motif finding, geometry, and others. This paper presents a new join algorithm, called Minesweeper, for joining relations that are stored in order data structures, such as B-trees. Under some mild technical assumptions, Minesweeper is able to achieve stronger runtime guarantees than previous join algorithms.

The Minesweeper algorithm is based on a simple idea. When data are stored in an index, successive tuples indicate *gaps*, i.e., regions in the output space of the join where no possible output tuples exist. Minesweeper maintains gaps that it discovers during execution and infers where to look next. In turn, these gaps may indicate that a large number of tuples in the base relations cannot contribute to the output of the join, so Minesweeper can efficiently skip over such tuples without reading them. By using an appropriate data structure to store the gaps, Minesweeper guarantees that we can find at least one point in the output space that needs to be explored, given the gaps so far. The key technical challenges are the design of this data structure, called the *constraint data structure*, and the analysis of the join algorithm under a more stringent runtime complexity measure.

To measure our stronger notion of runtime, we introduce the notion of a *certificate* for an instance of a join problem: essentially, a certificate is a set of comparisons between elements of the input relations that certify that the join output is exactly as claimed. We use the certificate as a measure of the difficulty of a particular instance of a join problem. That is, our goal is to find algorithms whose running times can be bounded by some function of the *smallest certificate size* for a particular input instance. Our notion has two key properties:

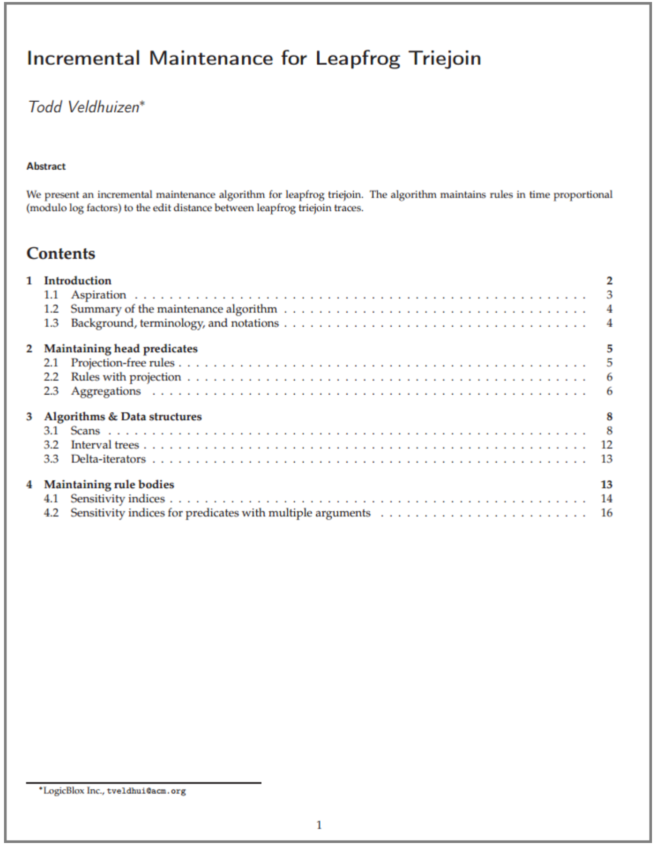
- *Certificate complexity captures the computation performed by widely implemented join algorithms.* We observe that the set of comparisons made by any join algorithm that interacts with the data by comparing elements of the

*This is the full version of our PODS'2014 paper.

1

Incremental Maintenance

- Incremental Maintenance for Leapfrog Triejoin
  - Todd L. Veldhuizen
  - March '13
  - http://arxiv.org/abs/1303.5313
- Each rule is incrementally maintained
- The work done to maintain the rule is proportional to the number of updates



Incremental Maintenance for Leapfrog Triejoin

*Todd Veldhuizen**

Abstract

We present an incremental maintenance algorithm for leapfrog triejoin. The algorithm maintains rules in time proportional (modulo log factors) to the edit distance between leapfrog triejoin traces.

Contents

*LogicBlox Inc., tveldhui@acm.org

Transaction Processing

- Transaction Repair: Full Serializability Without Locks
  - Todd L. Veldhuizen
  - March ‘14
  - http://arxiv.org/abs/1403.5645

- Lock-free, scalable transaction processing that achieves full serializability

Transaction Repair: Full Serializability Without Locks

Todd L. Veldhuizen
LogicBlox Inc.
Two Midtown Plaza
1349 West Peachtree Street NW
Suite 1880, Atlanta GA 30309

ABSTRACT

Transaction Repair is a method for lock-free, scalable transaction processing that achieves full serializability. It demonstrates parallel speedup even in inimical scenarios where all pairs of transactions have significant read-write conflicts. In the transaction repair approach, each transaction runs in complete isolation in a branch of the database; when conflicts occur, we detect and repair them. These repairs are performed efficiently in parallel, and the net effect is that of serial processing. Within transactions, we use no locks. This frees users from the complications and performance hazards of locks, and from the anomalies of sub-serializable isolation levels. Our approach builds on an incrementalized variant of leapfrog triejoin, an algorithm for existential queries that is worst-case optimal for full conjunctive queries, and on well-established techniques from programming languages: declarative languages, purely functional data structures, incremental computation, and fixpoint equations.

1. INTRODUCTION

1.1 Scenario

Consider the following artificial scenario chosen to highlight essential issues. A database tracks available quantities of warehouse items identified by sku number (*stock-keeping unit*). Each transaction adjusts quantities for a subset of skus, updating a database predicate inventory[sku] = qty. Suppose there are n skus, and each transaction adjusts skus chosen independently with probability $\alpha n^{-1/2}$. Most pairs of transactions will conflict when $\alpha \gg 1$: the expected number of skus common to two transactions is $E[\cdot] = n \cdot \left(\alpha n^{-1/2}\right)^2 = \alpha^2$, an instance of the Birthday Paradox.

Row-level locking is a bottleneck when $\alpha \gg 1$: since most transactions have skus in common, they quickly encounter lock conflicts and are put to sleep. Figure 1 (left) shows parallel speedup of transaction throughput for $\alpha = 0.1$, $\alpha = 1.0$, and $\alpha = 10$, using an efficient implementation of row-level locking on a multicore machine. Note that for $\alpha = 10$ there is no parallel speedup: there are so many conflicts that throughput is reduced to that of a single cpu.

Our approach, which we call *transaction repair*, is rather different. The LogicBlox database has been engineered from the ground-up to use purely functional and versioned data structures. Transactions run simultaneously, with no locking, each in complete isolation in its own branch of the database. We then detect conflicts and repair them. These repairs are performed efficiently in parallel, and the net result is a database state indistinguishable from sequential processing of transactions. With this approach, we are able to achieve parallel speedup even when there are large amounts of conflicts between transactions (Figure 1, right).

It does not strain credulity to report that transaction repair can achieve parallel speedup for the trivial scenario just described. Remarkably, our technique applies to arbitrary mixtures of complex transactions.

1.2 Transaction repair

Transaction repair combines three major ingredients:

1. Leapfrog triejoin: Each transaction in our system consists of one or more *rules* written in our declarative language LogiQL, a substantial augmentation of Datalog which preserves the clean lines of the original. Each LogiQL rule is evaluated using *leapfrog triejoin*, an algorithm for existential rules for which a significant optimality property was recently proven [14].

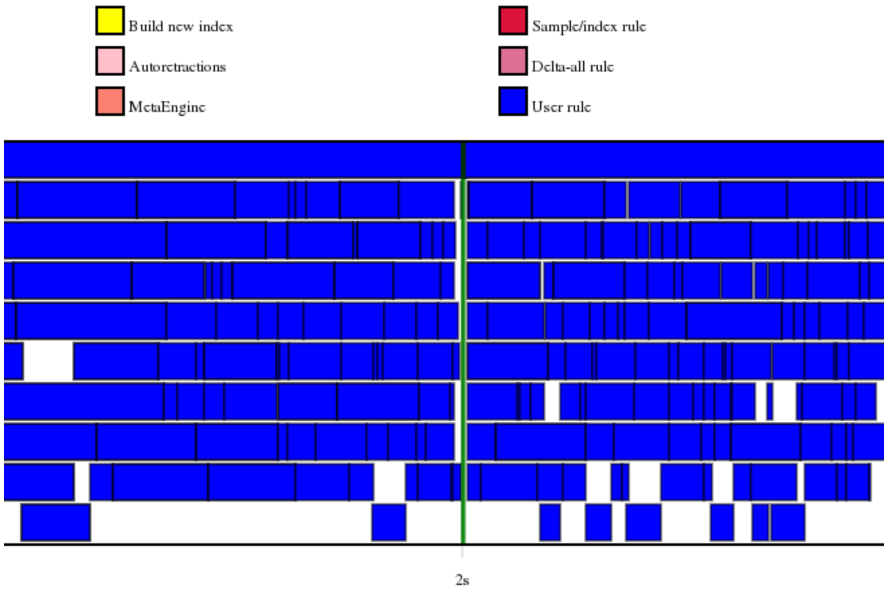
2. Incremental maintenance of rules: Leapfrog triejoin admits an efficient incremental maintenance algorithm that is designed to achieve cost proportional to the trace-edit distance of leapfrog triejoin traces [13]. We employ this algorithm to repair individual rules when conflicts occur between transactions. In operation, the maintenance algorithm collects *sensitivity indices* that precisely specify database state to which a rule is sensitive, in the sense that modifying that state could alter the observable outcomes of the transaction. Maintenance of individual rules is extended to maintenance of entire transactions by propagating changes through a dependency graph of the transaction rules (Section 5).

The third ingredient is transaction repair circuits, which we broadly outline in Section 1.4, and describe in detail in subsequent sections.

A bottom-up exposition would begin at the level of single rules and leapfrog triejoin, and describe how transaction repair is built on these foundations. However, the novelty of

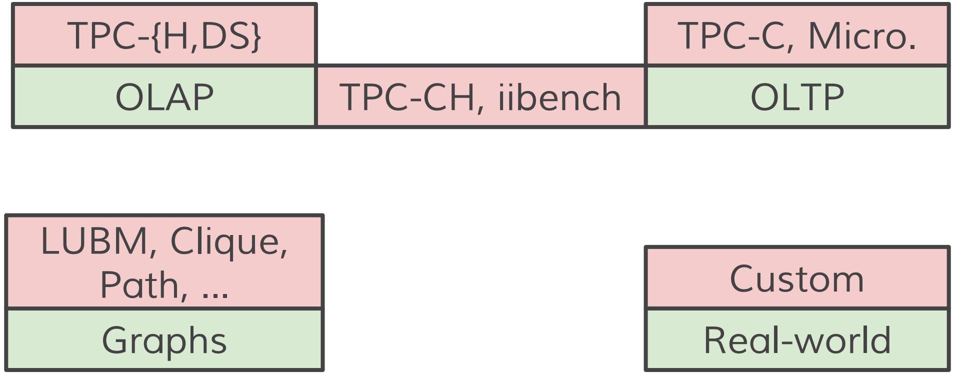
Intra-Query Parallelism

- Dynamic & adaptive domain decomposition (~dynamic sharding)
- Decomposition results into many small subdomains
  - >> #cpu cores, for large enough domains
- Each subdomain is going to require about the same amount of work
- Query applied on subdomains in parallel, without leaving any core idle



Benchmarking

What we benchmark



Variants

- Physical layer
  - E.g. iibench: normalized VS de-normalized schema
- Logical layer
  - E.g. TPC-CH aggregate queries: rules VS plain queries
- API layer
  - E.g. microbenchmarks: different API abstractions
    - engine API VS
    - low-level custom protocol over TCP VS
    - low-level custom protocol over HTTP VS
    - high-level custom protocol over HTTP

Reporting Site

TPC-H

lb4-non-entity	Current	Historical by date	Historical by job evaluation	Current analysis
lb4-non-entity-opt	Current	Historical by date	Historical by job evaluation	Current analysis
lb4-non-entity-serial	Current	Historical by date	Historical by job evaluation	
lb4-non-entity-tdx	Current	Historical by date	Historical by job evaluation	
lb4-entity	Current	Historical by date	Historical by job evaluation	
lb4-entity-tdx	Current	Historical by date	Historical by job evaluation	
lb4-entity-measure	Current	Historical by date	Historical by job evaluation	Current analysis
comparison		Historical by date		

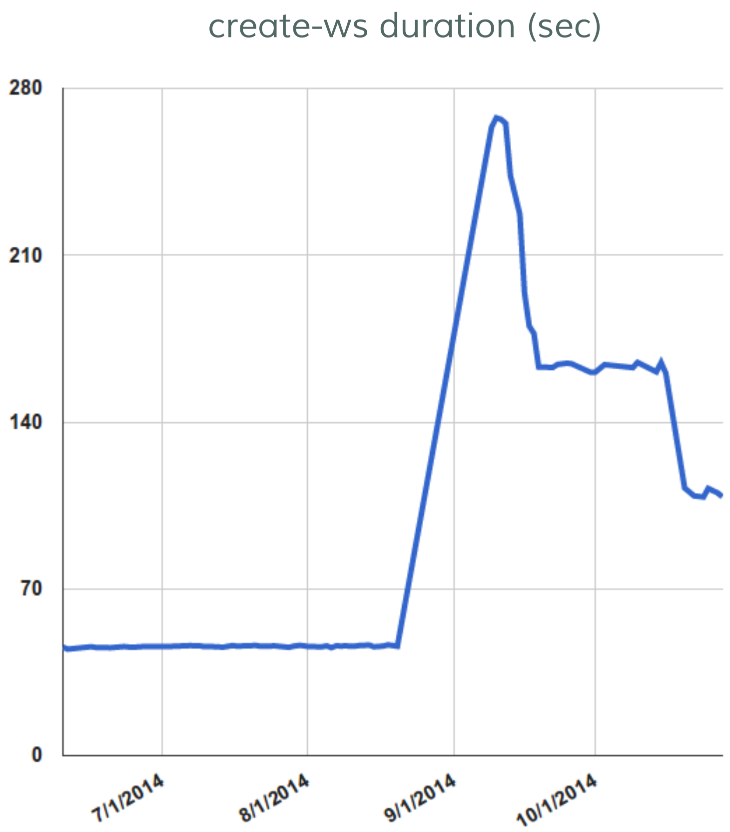
TPC-DS

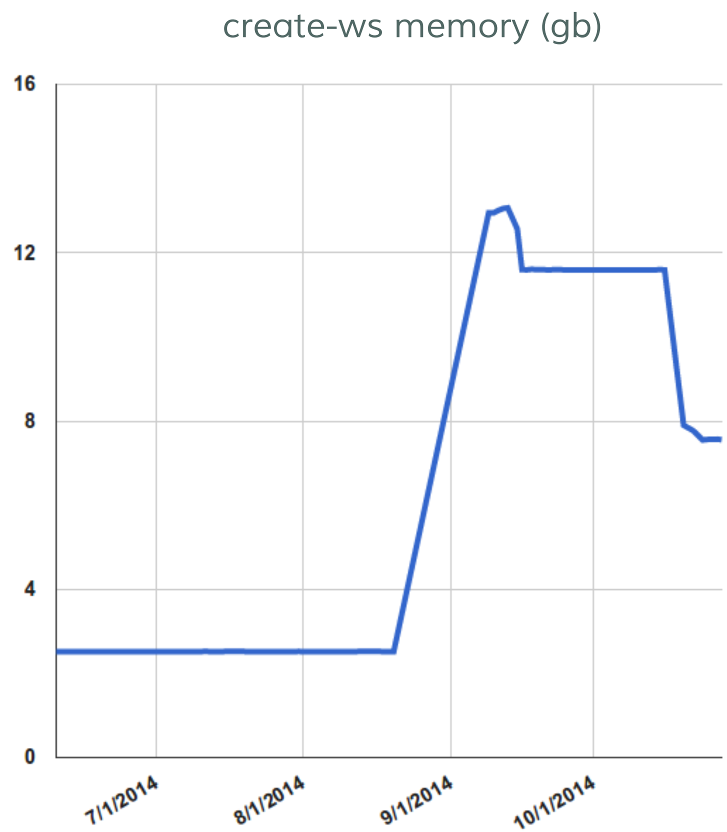
lb4-non-entity-tdx	Current	Historical by date	Historical by jobset evaluation
lb3-non-entity-tdx	Current	Historical by date	Historical by jobset evaluation
lb4-entity-tdx	Current	Historical by date	Historical by jobset evaluation
lb3-entity-tdx	Current	Historical by date	Historical by jobset evaluation
comparison		Historical by date	

Microbench

Smallbank	lb-server	Current	Historical by date	Historical by jobset evaluation
	lb-web-protobuf	Current	Historical by date	Historical by jobset evaluation
Shopping Cart	lb-server	Current	Historical by date	Historical by jobset evaluation
	lb-web-protobuf	Current	Historical by date	Historical by jobset evaluation
TinyBank	lb-server	Current	Historical by date	Historical by jobset evaluation
	lb-web-protobuf	Current	Historical by date	Historical by jobset evaluation
	lb-web-protobuf-no-http-control	Current	Historical by date	Historical by jobset evaluation
	lb-tdx	Current	Historical by date	Historical by jobset evaluation
	lb-measure	Current	Historical by date	Historical by jobset evaluation
	lb-measure-proto-params	Current	Historical by date	Historical by jobset evaluation
	runtime	Current	Historical by date	Historical by jobset evaluation
	comparison		Historical by date	Historical by jobset evaluation

Performance Monitoring





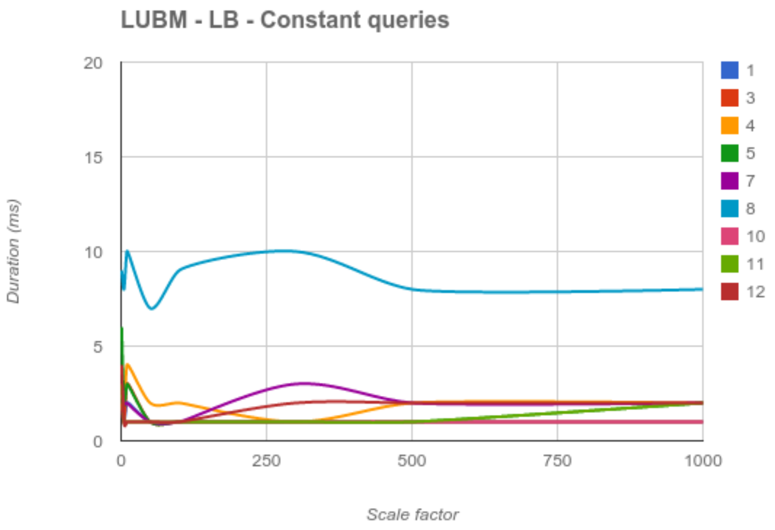
Benchmarking Graphs

Lehigh University Benchmark (LUBM)

- Evaluates Semantic Web repositories
- Original schema is described in OWL
  - All LUBM Ontology inference/constraints can be captured in LogiQL (with rules/constraints/subtyping)
    - This is not generally true
- Each dataset scale factor denotes the number of Universities in the Ontology
  - Datasets grow linearly
- 14 queries over a University Ontology
  - fixed resultset + a few simple joins : q1, q3, q4, q5, q7, q8, q10, q11, q12, q13
  - linearly growing resultset + 1 clique join : q2, q9
  - linearly growing resultset + no join : q6, q14

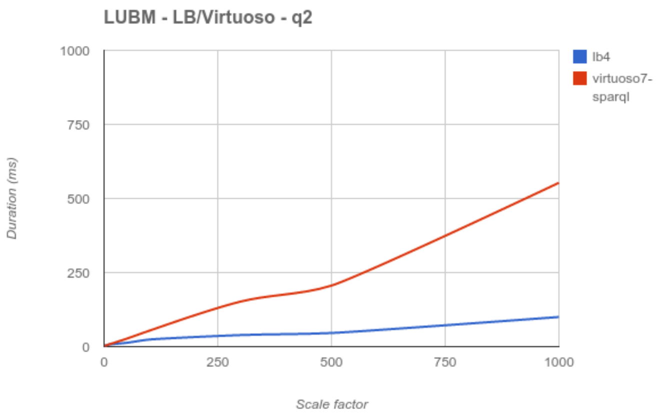
LUBM "fixed resultset" queries

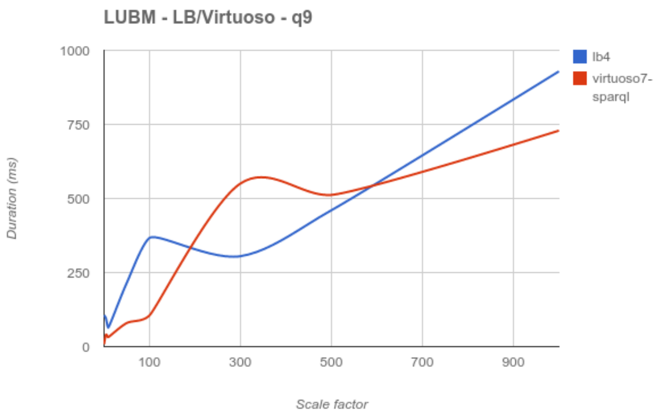
- All these queries return the same resultset regardless of the scale
- GraphDB: "Going from one node to a neighbour takes constant time"
- So a "fixed resultset" query should take the same time across all scales in a good GraphDB
  - It seems LB is a good GraphDB!
- LB: indexed binary relation (edge) + efficient join algorithm (LFTJ)
  - constant time



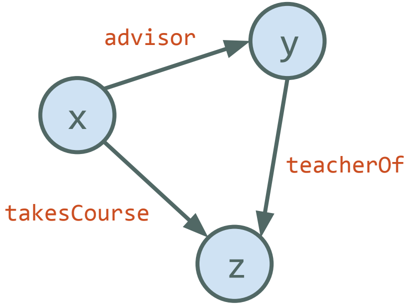
LUBM clique queries

- Clique queries are the most complex joins in LUBM
- LB & Virtuoso perform similarly





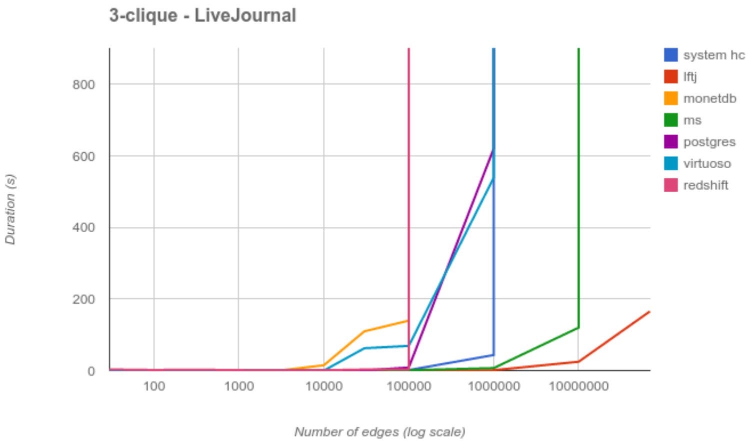
LUBM q9

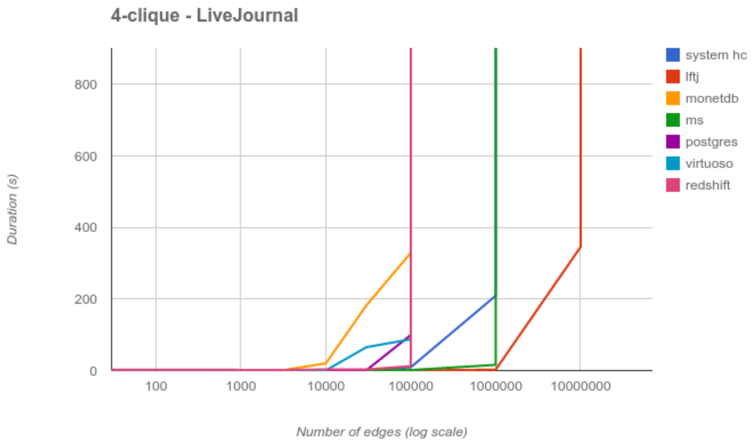


LB - LogiQL	Virtuoso - SparQL

`_(x,y,z) <-`
`    Student(x),`
`    Faculty(y),`
`    Course(z),`
`    advisor(x,y),`
`    teacherOf(y,z),`
`    takesCourse(x,z).`
	`SELECT ?X ?Y ?Z`
	`WHERE`
	`{`
	`  ?X rdf:type ub:Student .`
	`  ?Y rdf:type ub:Faculty .`
	`  ?Z rdf:type ub:Course .`
	`  ?X ub:advisor ?Y .`
	`  ?Y ub:teacherOf ?Z .`
	`  ?X ub:takesCourse ?Z`
	`}`

Pure clique queries





"Optimal Join Algorithms: from Theory to Practice"
(paper under submission)

Academic Collaborations

Current and past collaborators

Berkeley (Databases - Bill Marczak)
Columbia (Statistics - Andrew Gelman, Eric Johnson, and 1 Post-doc)
Columbia (Databases- Ken Ross^)
Davis (Databases - TJ Green*, Bertram Ludascher, Daniel Zinn*, 1 PhD)
Delft (Programming Languages – Eelco Visser and 2 Post-docs*, 1 PhD*)
Georgia State University (Databases - Raj Sunderraman and 2 PhD’s* and 1 Masters*)
Georgia Tech (Machine Learning - Nick Vasiloglou and 4 PhD’s* and 2 Masters*)
Georgia Tech (Machine Learning – Polo Chau and 1 PhD)
Georgia Tech (Operations Research – Dave Goldsman and 1 PhD’s)
Georgia Tech (Software Engineering - Spencer Rugaber* and 1 PhD)
Georgia Tech (Accelerators - Sudha Yalamanchili and 3 PhD’s*)
Groningen (Herman Balsters and 1 Masters)
Gent (Constraint Satisfaction – Tom Schrijvers and 1 PhD, 1 Masters)
Hasselt University (Databases - Frank Neven and 2 PhD’s)
Indiana (Programming Languages – Jeremy Siek)
MIT (Stats and Operations Research - Rama Ramakrishnan),
MIT(Operations Research - Edgar Blanco)

- * full-time at LogicBlox, ^ part-time at LogicBlox

Current and past collaborators

Michigan State University (Software Engineering - Kurt Stirewalt*, L Dillon and 1 Post-doc*, 1 PhD)
Neumont & INTI University (Modeling - Terry Halpin and Matt Curland)
Northwestern (Operations Research - Bob Fourer, Diego Klabjan, 1 Post-doc, 1 PhD)
Oregon State (End User Software Engineering – Chris Scaffidi, 1 PhD)
Oxford (Databases - Dan Olteanu for 1 year sabbatical)
Penn (Databases & Networking - Boon Loo, Val Tannen, and 1 PhD candidate, 1 undergrad)
Penn (Programing Languages – Benjamin Pierce and 1 PhD candidate)
Portland State (Programming Languages – Tim Sheard^)
Rice (PL and Theorem Provers - Walid Taha and 1 Post-docs* and 1 PhD*)
Rice (Databases- Chris Jermaine and 1 Post-doc*)
Stanford (Databases & ML – Chris Re, 1 Post-doc)
SUNY at Buffalo (Theory - Atri Rudra, Hung Q Ngo, 1 PhD)
University of Athens (PL - Yannis Smaragdakis and 1 Post-doc*, 4PhD*)
University of Chicago (Computational Logic & AI – Tim Hinrichs)
University of Georgia (Software Engineering – Eileen Kraemer)
Virginia Tech (Multi-paradigm programming - Eli Tilevich and 1 Masters Student)
Waterloo (Software Engineering- Todd Veldhuizen*, Krzysztof Czarnecki and 2 PhD's* and 1 Masters)
Waterloo (Databases – Ashraf Aboulnaga and 1 PhD)

THANK YOU. QUESTIONS?

How we benchmark

Nix

- Purely-functional software configuration management system
  - composable
  - maintainable
- Reproducible
  - Takes care of dependencies, daemons, configuration

`lubm.nix`

```
{
  src ? ./lubm,
  platform,
  data_sets,
  data_dir ? "",
  memory ? 8,
  db_dir ? ".",
  db_timeout ? 3600,
  query_timeout ? 1800,
  features ? ["machine-type"]
}:
{
  # benchmark body
}
```

Infrastructure

- Integrated into our buildfarm
  - Special machines for benchmarking
    - Identical to each other

- Hydra
  - Nix-based distributed continuous build system
  - Build tasks in Nix

- Regular benchmark runs (builds)
  - After each commit
    - Fine-grained regression tracking
  - Once per day
    - Heavier variants

- Incremental benchmark runs (builds)
  - New run only if either the benchmark or the engine changed

Data Structures

- Fully persistent DS
  - each transaction branches a version of the database
    - O(1)
  - perfect read-only transactions scaling
    - they don’t wait write transactions
    - they don’t block write transactions
- Write-optimized DS
  - LSM-like trees
- High data compression rates

LUBM schema translation

OWL Schema Example

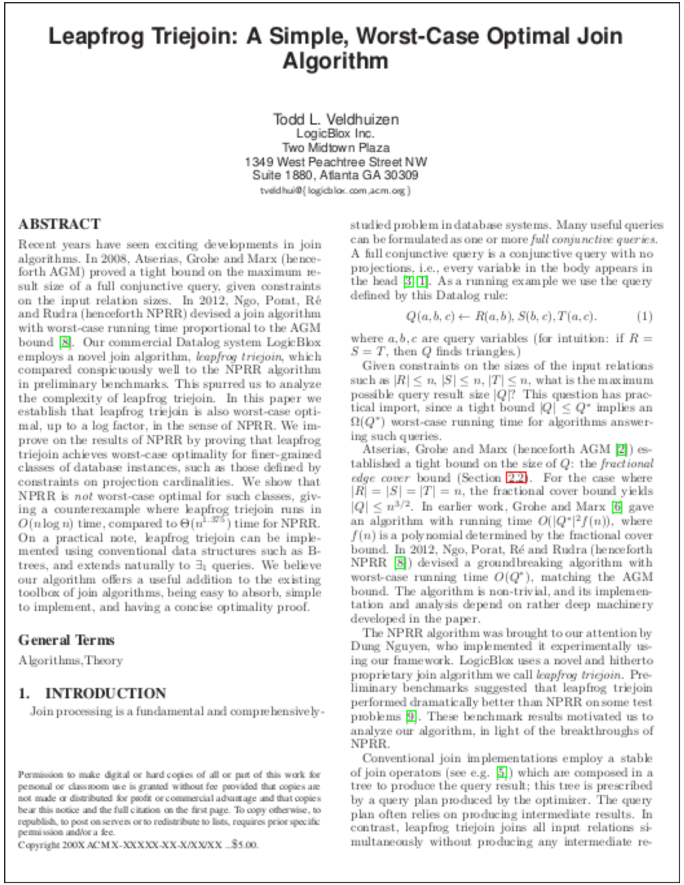
```
<owl:Class rdf:ID="University">
  <rdfs:label>university</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Organization" />
</owl:Class>
<owl:Class rdf:ID="Department">
  <rdfs:label>university department</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Organization" />
</owl:Class>
<owl:Class rdf:ID="ResearchGroup">
  <rdfs:label>research group</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Organization" />
</owl:Class>

<owl:TransitiveProperty rdf:ID="subOrganizationOf">
  <rdfs:label>is part of</rdfs:label>
  <rdfs:domain rdf:resource="#Organization" />
  <rdfs:range rdf:resource="#Organization" />
</owl:TransitiveProperty>
```

LogiQL Schema Example

```
University(o) -> Organization(o).
lang:entity(`University).

Department(o) -> Organization(o).
lang:entity(`Department).

ResearchGroup(o) -> Organization(o).
lang:entity(`ResearchGroup).

subOrganizationOf(o1,o2) -> Organization(o1),
                            Organization(o2).
subOrganization(x,y) <- subOrganizationOf(x,y).
subOrganization(x,y) <- subOrganizationOf(x,z),
                        subOrganization(z,y). //TC
```

WHY LB IS SO FAST?

- Leapfrog Triejoin takes into account all relations of the join simultaneously, so it can narrow down the resultset much more quickly than typical pairwise join algorithms.



Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm

Todd L. Veldhuizen
LogicBlox Inc.
Two Midtown Plaza
1349 West Peachtree Street NW
Suite 1880, Atlanta GA 30309
tveldhui@{logicblox.com,acm.org}

ABSTRACT

Recent years have seen exciting developments in join algorithms. In 2008, Atserias, Grohe and Marx (henceforth AGM) proved a tight bound on the maximum result size of a full conjunctive query, given constraints on the input relation sizes. In 2012, Ngo, Porat, Ré and Rudra (henceforth NPRR) devised a join algorithm with worst-case running time proportional to the AGM bound [8]. Our commercial Datalog system LogicBlox employs a novel join algorithm, *leapfrog triejoin*, which compared conspicuously well to the NPRR algorithm in preliminary benchmarks. This spurred us to analyze the complexity of leapfrog triejoin. In this paper we establish that leapfrog triejoin is also worst-case optimal, up to a log factor, in the sense of NPRR. We improve on the results of NPRR by proving that leapfrog triejoin achieves worst-case optimality for finer-grained classes of database instances, such as those defined by constraints on projection cardinalities. We show that NPRR is *not* worst-case optimal for such classes, giving a counterexample where leapfrog triejoin runs in $O(n \log n)$ time, compared to $\Theta(n^{1.375})$ time for NPRR. On a practical note, leapfrog triejoin can be implemented using conventional data structures such as B-trees, and extends naturally to $\exists_1$ queries. We believe our algorithm offers a useful addition to the existing toolbox of join algorithms, being easy to absorb, simple to implement, and having a concise optimality proof.

General Terms

Algorithms,Theory

1. INTRODUCTION

Join processing is a fundamental and comprehensively-studied problem in database systems. Many useful queries can be formulated as one or more *full conjunctive queries*. A full conjunctive query is a conjunctive query with no projections, i.e., every variable in the body appears in the head [3][1]. As a running example we use the query defined by this Datalog rule:

$$Q(a,b,c) \leftarrow R(a,b), S(b,c), T(a,c). \tag{1}$$

where a,b,c are query variables (for intuition: if $R = S = T$, then Q finds triangles.)

Given constraints on the sizes of the input relations such as $|R| \leq n$, $|S| \leq n$, $|T| \leq n$, what is the maximum possible query result size $|Q|$? This question has practical import, since a tight bound $|Q| \leq Q^*$ implies an $\Omega(Q^*)$ worst-case running time for algorithms answering such queries.

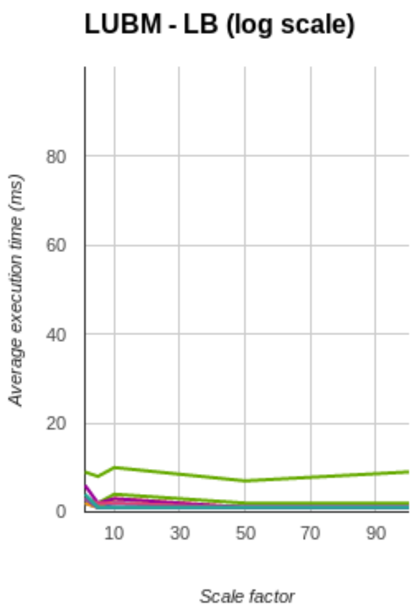
Atserias, Grohe and Marx (henceforth AGM [2]) established a tight bound on the size of Q: the *fractional edge cover* bound (Section 2.2). For the case where $|R| = |S| = |T| = n$, the fractional cover bound yields $|Q| \leq n^{3/2}$. In earlier work, Grohe and Marx [6] gave an algorithm with running time $O(|Q^*|^2 f(n))$, where $f(n)$ is a polynomial determined by the fractional cover bound. In 2012, Ngo, Porat, Ré and Rudra (henceforth NPRR [8]) devised a groundbreaking algorithm with worst-case running time $O(Q^*)$, matching the AGM bound. The algorithm is non-trivial, and its implementation and analysis depend on rather deep machinery developed in the paper.

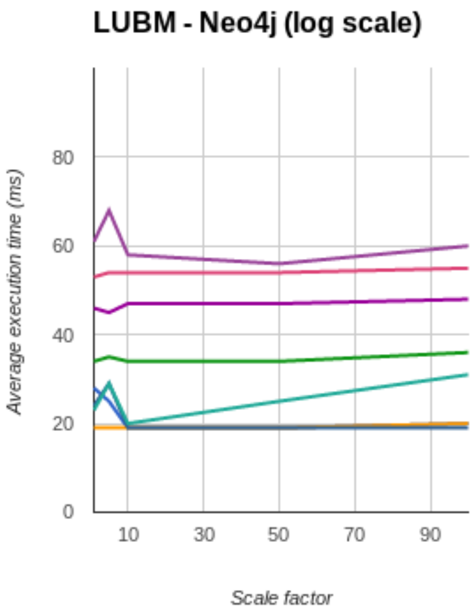
The NPRR algorithm was brought to our attention by Dung Nguyen, who implemented it experimentally using our framework. LogicBlox uses a novel and hitherto proprietary join algorithm we call *leapfrog triejoin*. Preliminary benchmarks suggested that leapfrog triejoin performed dramatically better than NPRR on some test problems [9]. These benchmark results motivated us to analyze our algorithm, in light of the breakthroughs of NPRR.

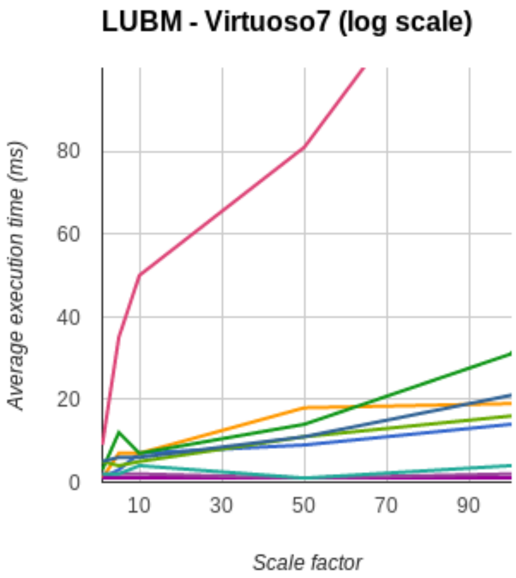
Conventional join implementations employ a stable of join operators (see e.g. [5]) which are composed in a tree to produce the query result; this tree is prescribed by a query plan produced by the optimizer. The query plan often relies on producing intermediate results. In contrast, leapfrog triejoin joins all input relations simultaneously without producing any intermediate re-

LUBM RESULTS

- All LUBM queries except q2, q6, q9, q14, return the same resultset for **all** scales, so these queries should take the same time for **all** scales on a good graphdb.
  - They do on Neo4j & Virtuoso. They do on LB too! So all of them are good graphdbs!
  - q2, q6, q9, q14 should grow linearly since datasets scale linearly too







Choose many?

- Using plethora of specialized systems means increased:
  - development cost
  - integration cost
  - maintenance cost

- Specialized systems are only worth it if 10x-100x better
  - reversing Stonebraker's argument