Benchmarking @LogicBlox

George Kollias (LogicBlox)

LDTC 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
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
“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
- Multi-way join
  - Variant of Sort-Merge Join

- Beyond Worst-Case Analysis for Joins with Minesweeper
  - Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, Atri Rudra
  - PODS ‘14
- Multi-way join
Incremental Maintenance

- Incremental Maintenance for Leapfrog Triejoin
  - Todd L. Veldhuizen
  - March ‘13

- Each rule is incrementally maintained

- The work done to maintain the rule is proportional to the number of updates
Transaction Repair: Full Serializability Without Locks

Todd L. Veldhuizen

Lock-free, scalable transaction processing that achieves full serializability

ABSTRACT
Transaction repair is a method for lock-free, scalable transaction processing that achieves full serializability. It demonstrates parallel speedups even in situations where all prior solutions have significant scalability issues. In the transaction repair approach, each transaction runs in isolation and, as long as transactions do not interfere with each other, we detect and repair them. These repairs are performed efficiently in parallel, and the net effect is that of serial processing. Within transactions, we run actors. This frees users from the complexities and performance nuances of locks, and from the semantics of subserializable isolation levels. Our approach builds on an incremental variant of locking trinquis, an algorithm for existential queries that is stateless and optimal for handwritten queries, and on self-stabilizing techniques from programming languages, declarative languages, and functional data structures, incremental computation, and inductive equations.

1. INTRODUCTION
1.1 Scenario
Consider the following adversarial scenario to highlight essential issues. A database stores available quantities of warehouse items identified by site number (which is unique). Each transaction makes an update to the database, and each transaction update quantity is chosen independently with probability one. A failed pair of transactions would occur when both had selected a site among the four that are in stock. The expected number of such conflicts is $\frac{n^2}{4} - \frac{n}{4} + \frac{1}{4}$, an instance of the Birthday Paradox.

Level locking is a bottleneck when $n > 1$ since most transactions have site in common, they quickly encounter lock conflicts and are put to sleep. Figure 1 (left) shows parallel speedup of transaction throughput for $n = 0, 1, 2, 3, 4, 5, 6$, and $n = 10$, using an efficient implementation of two-level locking on a commodity machine. Note that for $n = 10$ there is no parallel speedup; there are so many conflicts that throughput is reduced to that of a single core.

Our approach, which we call transaction repair, is rather different. The Logfile database has been engineered from the ground-up to use purely functional and versioned data structures. Transactions run atomically, with no locking, such is 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 conflict between transactions (Figure 1, right).

It does not assume anything to report that transactions execute atomically, sequentially. Our approach is described. Similarly, our technique applies to arbitrary mixtures of complex transactions.

1.2 Transaction repair
Transaction repair conditions three major ingredients:

1. Level locking. Each transaction in our system consists of two or more tasks written in our declarative language LogQL, a substantial augmentation of SQL which preserves the clean lines of the original. This is a key differentiator between transaction repair and the LockfreeLog algorithm of the original trinquis algorithm. We define an algorithm for existential queries which is stateless and optimal for handwritten queries.

2. Incremental maintenance of rules: Using efficient incremental maintenance algorithms that are designed to achieve cost proportional to the number of new messages introduced to transaction graphs. We apply this algorithm to repair individual rules when conflicts occur between transactions. In operation, the maintenance algorithm modifies existing messages that are in conflict or introduces new messages as the transaction proceeds. The net result is that modifying a rule does not affect the shareable semantics of the transaction.

3. 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 approach would bring at the level of single transactions, and thereby low 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
  - $\gg \#$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
What we benchmark

<table>
<thead>
<tr>
<th>TPC-{H,DS}</th>
<th>TPC-C, Micro.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLAP</td>
<td>TPC-CH, iibench</td>
</tr>
<tr>
<td>OLTP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LUBM, Clique, Path, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Custom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-world</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Data Source</th>
<th>Current</th>
<th>Historical by date</th>
<th>Historical by job evaluation</th>
<th>Current analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TPC-H</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1b-4 non-entity</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-4 non-entity-opt</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-4 non-entity-serial</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-4 non-entity-topk</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-4 entity</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-4 entity-tops</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-4 entity-measurecomparison</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td><strong>TPC-DS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1b-4 non-entity-topk</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-4 non-entity-serial</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-4 entity-tops</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-4 entity-measurecomparison</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td><strong>Microbench</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallBank</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-solver</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-web-protocol</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>Shopping Cart</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-solver</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-web-protocol</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>TinyBank</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-solver</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-web-protocol</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-web-protocol-no-protocol-control</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-topk</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-measure</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>1b-measure-protocol</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>runtime</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
<tr>
<td>comparison</td>
<td>Current</td>
<td>Historical by date</td>
<td>Historical by job evaluation</td>
<td>Current analysis</td>
</tr>
</tbody>
</table>
Performance Monitoring

create-ws duration (sec)

create-ws memory (gb)
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

<table>
<thead>
<tr>
<th>LB - LogiQL</th>
<th>Virtuoso - SparQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>_(x,y,z) &lt;-</td>
<td>SELECT ?X ?Y ?Z</td>
</tr>
<tr>
<td>Student(x),</td>
<td>WHERE</td>
</tr>
<tr>
<td>Faculty(y),</td>
<td>{}</td>
</tr>
<tr>
<td>Course(z),</td>
<td>?X rdf:type ub:Student .</td>
</tr>
<tr>
<td>advisor(x,y),</td>
<td>?Y rdf:type ub:Faculty .</td>
</tr>
<tr>
<td>teacherOf(y,z),</td>
<td>?Z rdf:type ub:Course .</td>
</tr>
<tr>
<td></td>
<td>?X ub:takesCourse ?Z</td>
</tr>
</tbody>
</table>
“Optimal Join Algorithms: from Theory to Practice”
(paper under submission)
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 (Programming 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*, 4 PhD*)
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

```nix
{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
**OWL Schema Example**

```xml
<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**

```logiql
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
```
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.

WHY LB IS SO FAST?

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

Todd L. Veldhuizen
LogicBlox Inc.
Two Matthews Plaza
1340 West Peachtree Street NW
Suite 1850
Atlanta, GA 30309
toddvl@logicblox.com

ABSTRACT

Recent years have seen exciting developments in join algorithms. In 2008, Amster, Charle and Max (henceforth MCM) proposed a trie-based join algorithm, which compared favorably well to the NPBJ algorithm in preliminary benchmarks. This paper aims 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 NPBJ. We improve on the results of NPBJ by proving that leapfrog triejoin achieves worst-case optimality for tree-structured classes of database instances, such as those defined by constraints on projection coordinates. We show that NPBJ is not worst-case optimal for such classes, giving a counterexample where its running triejoin time in $O(n \log n)$ time, compared to $O(n^{1.5})$ time for NPBJ. On a practical note, leapfrog triejoin can be implemented using conventional data structures such as trees, and extends naturally to $k$-ary trees. We believe our algorithm offers a useful addition to the existing tradition of join algorithms, being easy to analyze, simple to implement, and having a cleaner optimality proof.

General Terms

Algorithm Theory

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

©2013. LogicBlox. All Rights Reserved.
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