Tesseract

Distributed Graph Database
FOSDEM 2015

Courtney Robinson
courtney@zcourts.com
31 Jan 2015
Background from Gephi
• I can be found around the web as “zcourts”, Google it…
• The web is one very prominent example of a graph
• Too big for a single machine
• So we must split or “partition” it over multiple
• Partitioning is hard…in fact, it has been shown to be np-complete
• All we can do is edge closer to more “optimal” solutions
• The Tesseract is an ongoing research project
• Its focus is on distributed graph partitioning
• The rest of this presentation is a series of solutions, which together, takes one step closer to faster distributed graph processing
Terminology

**Graph** - A graph $G$ is made up of a set of vertices and edges, 
$$G = (V,E)$$

**Vertex** - Smallest unit of user accessible datum

**Edge** - Connects two vertices, may have a direction

**Property** - Key value pair available on an Edge or Vertex
Aims of the Tesseract

1. Implement distributed eventually consistent graph database

2. Develop a distributed graph partitioning algorithm

3. Develop a computational model able to support both real time and batch processing on a distributed graph
Aims of the Tesseract

1. **Implement distributed eventually consistent graph database**

2. Develop a distributed graph partitioning algorithm

3. Develop a computational model able to support both real time and batch processing on a distributed graph
CRDTs...in one slide™
CRDTs...in one slide™

Conflict free replicated data types
CRDTs...in one slide™

Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.
CRDTs...in one slide™

Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

(1+2) + 3 = 1 + (2+3)

Associative
Conflict free replicated data types 

i.e provably eventually consistent \( (\text{Shapiro \ et \ al}) \) replicated & distributed data structures.

\[
(1+2) + 3 = 1 + (2+3) \\
1 + 2 = 2 + 1
\]
Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

(1+2) + 3 = 1 + (2+3)
1 + 2 = 2 + 1
1 + 1 ≠ 1

Associative
Commutative
Not Idempotent
CRDTs... in one slide™

Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

(1 + 2) + 3 = 1 + (2 + 3)
1 + 2 = 2 + 1
1 + 1 ≠ 1

Associative

Commutative

Not Idempotent
CRDTs...in one slide™

Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

Un fortunately addition isn’t enough. The CIA properties are required to have a CRDT

\[(1+2) + 3 = 1 + (2+3)\]

\[1 + 2 = 2 + 1\]

\[1 + 1 \neq 1\]

Commutative

Associative

Not Idempotent
Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

\[(1+2) + 3 = 1 + (2+3)\]
\[1 + 2 = 2 + 1\]
\[1 + 1 \neq 1\]

Unfortunately addition isn’t enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… **Sets!**
CRDTs...in one slide™

Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

\[(1+2) + 3 = 1 + (2+3)\]
\[1 + 2 = 2 + 1\]
\[1 + 1 \neq 1\]

Unfortunately addition isn’t enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… Sets!
CRDTs...in one slide™

Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

\[(1+2) + 3 = 1 + (2+3)\]
\[1 + 2 = 2 + 1\]
\[1 + 1 \neq 1\]

Associative
Commutative
Not Idempotent

Unfortunately addition isn’t enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… Sets!

**Addition with sets is done using** \(\cup\)

zcourts.com
CRDTs...in one slide™

Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

(1+2) + 3 = 1 + (2+3)
1 + 2 = 2 + 1
1 + 1 ≠ 1

Unfortunately addition isn’t enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… Sets!

Addition with sets is done using \( \cup \)

\[(1 \cup 2) \cup 3 = 1 \cup (2 \cup 3)\]
CRDTs...in one slide™

Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

(1+2) + 3 = 1 + (2+3)
1 + 2 = 2 + 1
1 + 1 ≠ 1

Unfortunately addition isn’t enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… Sets!

Addition with sets is done using ∪

(1 ∪ 2) ∪ 3 = 1 ∪ (2 ∪ 3)
1 ∪ 2 = 2 ∪ 1

Commutative
Associative
Not Idempotent
CRDTs...in one slide™

Conflict free replicated data types
i.e provably eventually consistent (Shapiro et al) replicated & distributed data structures.

\[(1+2) + 3 = 1 + (2+3)\]
\[1 + 2 = 2 + 1\]
\[1 + 1 \neq 1\]

- **Commutative**
- **Associative**
- **Not Idempotent**

Unfortunately addition isn’t enough. The CIA properties are required to have a CRDT

Luckily, graphs can be represented by a common mathematical structure which exhibits all 3 properties… **Sets**!

Addition with sets is done using \( \cup \)

\[(1 \cup 2) \cup 3 = 1 \cup (2 \cup 3)\]
\[1 \cup 2 = 2 \cup 1\]
\[1 \cup 1 = 1\]

- **Commutative**
- **Associative**
- **Idempotent!**
I lied, two slides...™?
I lied, two slides...™?

• Several types of CRDTs are available.
Several types of CRDTs are available.
They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
Several types of CRDTs are available. They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge. OR-set i.e. “Observed Removed”…add wins!
I lied, two slides…™?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!
I lied, two slides…™?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!

\[
\begin{align*}
S \\
S1 \\
S2 \\
S3
\end{align*}
\]

\[
\begin{align*}
\{\} & \rightarrow \text{add(a)} & \{\} \\
\{\} & \rightarrow \{\} \\
\{\} & \rightarrow \{\}
\end{align*}
\]

Green = insert, Pink = merge, Purple = replicate, Red = remove
Several types of CRDTs are available. They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.

OR-set i.e. “Observed Removed”…add wins!
I lied, two slides...™?

• Several types of CRDTs are available.
• They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
• OR-set i.e. “Observed Removed”...add wins!

- $\text{S1}$
- $\text{S2}$
- $\text{S3}$

1. $\text{add(a)}$
2. $\{\}$
3. $\{\text{a}_\pi\}$
4. $\text{add(a)}$
5. $\{\}$
6. $\{\text{a}_\lambda\}$

- $\text{S}$

$\text{set} = \text{insert}$
$\text{merge} = \text{merge}$
$\text{replicate} = \text{replicate}$
$\text{remove} = \text{remove}$
I lied, two slides...™?

- Several types of CRDTs are available.
- They provide us with "Strong Eventual Consistency" i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. "Observed Removed"...add wins!
I lied, two slides...™?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”...add wins!

```
S
S1
S2
S3
```

```
\begin{align*}
\text{add}(a) & \quad \{\} \rightarrow \{a_\pi\} \\
\text{add}(a) & \quad \{\} \rightarrow \{a_\lambda\} \\
\text{add}(a) & \quad \{\} \rightarrow \{a_\lambda, a_\pi\}
\end{align*}
```
Several types of CRDTs are available.

They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.

OR-set i.e. “Observed Removed”...add wins!

Each node adds “a” with a unique tag locally

- = insert  = merge  = replicate  = remove
• Several types of CRDTs are available.
• They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
• OR-set i.e. “Observed Removed”…add wins!
I lied, two slides…™?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”…add wins!
I lied, two slides...™?

• Several types of CRDTs are available.
• They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
• OR-set i.e. “Observed Removed”…add wins!

\[ S \]

\[ S_1 \]

\[ S_2 \]

\[ S_3 \]

- \( \{ \} \) = insert
- \( \{ \} \) = merge
- \( \{ \} \) = replicate
- \( \{ \} \) = remove

\( \text{add}(a) \)

\( \{a, \} \) = insert

\( \{a, a\} \) = merge

\( \{a, a\} \) = replicate

\( \{a, a\} \) = remove
I lied, two slides...™?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”...add wins!

Mark only -a_π as deleted.

- = insert  - = merge  - = replicate  - = remove
I lied, two slides…™?

• Several types of CRDTs are available.
• They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
• OR-set i.e. “Observed Removed”…add wins!
I lied, two slides…™?

• Several types of CRDTs are available.
• They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
• OR-set i.e. “Observed Removed”…add wins!

S
S1
S2
S3

add(a) = insert
del(a) = remove

\{\}\rightarrow\{a_\pi\}\rightarrow\{a_\pi\}\rightarrow\{-a_\pi\}\rightarrow\{a_\lambda,-a_\pi\}

\{\}\rightarrow\{a_\lambda\}\rightarrow\{-a_\pi\}\rightarrow\{a_\lambda,-a_\pi\}

\{\}\rightarrow\{a_\lambda\}\rightarrow\{a_\lambda,a_\pi\}\rightarrow\{a_\lambda,-a_\pi\}

\{\}\rightarrow\{a_\lambda\}\rightarrow\{a_\lambda\}\rightarrow\{-a_\pi\}\rightarrow\{a_\lambda,-a_\pi\}
I lied, two slides...™?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”...add wins!

\[ \text{S}\]
\[ \text{S}_1 \]
\[ \text{S}_2 \]
\[ \text{S}_3 \]

- \( \text{add}(a) \)
  - \( \{\} \rightarrow \{a_\pi\} \)
  - \( \{\} \rightarrow \{a_\lambda\} \)
  - \( \{\} \rightarrow \{a_\lambda\} \)

- \( \text{del}(a) \)
  - \( \{a_\pi\} \rightarrow \{-a_\pi\} \)
  - \( \{a_\lambda\} \rightarrow \{-a_\pi\} \)
  - \( \{a_\lambda,a_\pi\} \rightarrow \{a_\lambda,a_\pi\} \)

\(=\) insert  \(=\) merge  \(=\) replicate  7  \(=\) remove
I lied, two slides...™?

- Several types of CRDTs are available.
- They provide us with “Strong Eventual Consistency”, i.e. given states propagate we’re provably guaranteed to converge.
- OR-set i.e. “Observed Removed”...add wins!

\[
\begin{align*}
S & \quad \{\} \\
S1 & \quad \{a_\pi\} \\
S2 & \quad \{a_\lambda\} \\
S3 & \quad \{\} \\
\end{align*}
\]

\[
\begin{align*}
\text{add}(a) & \quad \{a_\pi\} \\
\text{del}(a) & \quad \{-a_\pi\} \\
\text{merge} & \quad \{a_\lambda, -a_\pi\} \\
\text{replicate} & \quad \{a_\lambda, -a_\pi\} \\
\end{align*}
\]

Merge takes symmetrical difference of the local and remote sets resulting in \(a_\lambda\) being in the set.
Several types of CRDTs are available.

They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.

OR-set i.e. “Observed Removed”...add

Merge takes symmetrical difference of the local and remote sets resulting in $a_{\lambda}$ being in the set.
Several types of CRDTs are available.

They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.

OR-set i.e. “Observed Removed”…add wins!

- = insert
- = merge
- = replicate
- = remove
Several types of CRDTs are available.

They provide us with “Strong Eventual Consistency” i.e. given states propagate we’re provably guaranteed to converge.

OR-set i.e. “Observed Removed”...add wins!

User never sees tags!

Query time checks are used to enable DAGs (if violation of DAG constraint is detected then the runtime simply says the violating edge does not exist and triggers clean up)

Note, the deleted “a” is optionally kept as a tombstone if the runtime is configured to support “snapshots”
Aims of the Tesseract

1. Implement distributed eventually consistent graph database

2. Develop a distributed graph partitioning algorithm

3. Develop a computational model able to support both real time and batch processing on a distributed graph
One very important property of a CRDT is:

\{a,b,c,d\} \Leftrightarrow \{a,b\} \cup \{c,d\}

Those two sets being logically equivalent is a desirable property.

Enables partitioning (with rendezvous hashing for e.g.)
Naïve “cascading vertices”

- Naïve graph partitioning
- Depends on the query model to make up for its Naïvety
- Uses hashing to place data
- Two cascading algorithms formulated from:
  \( V \) = the vertex to cascade
  \( n \) = max nodes to cascade across
  \( n' \) = auto-determined value of \( n \), using logistics growth model
  \( d \) = \( \text{deg}(v) \) = Degree of \( V \)
  \( e \) = \( \langle \forall \text{deg}(v) \in G \rangle \) i.e. average degree of all vertices in the graph
  \( |nV| \) = Max number of edges per node for a vertex
    i.e. cascading point (min number of edges before cascading occurs)

1. \( |nV| = d / n \) - user provides \( n \), split evenly across nodes
2. \( |nV| = \max(d,e) / n \) - user provides \( n \), split evenly based on \( d \) or \( e \) if \( e \) is bigger
“Cascading vertices” by example

• Let’s use Twitter followers as an example
• Each letter represents a unique follower
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

\[
\begin{align*}
S \quad S1 \quad S2 \quad S3 \\
\{\} \quad \{\} \quad \{\} \\
\end{align*}
\]

- Green circle = insert
- Purple circle = cascade
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

```
S
S1
S2
S3
```

```
{ }
{ } = insert
{ } = cascade
```
“Cascading vertices” by example

• Let’s use Twitter followers as an example
• Each letter represents a unique follower

add(…) performs a cascade(deg(V))

S
S1
S2
S3

\{\}
\{\}
\{\}

= insert  = cascade
“Cascading vertices” by example

• Let’s use Twitter followers as an example
• Each letter represents a unique follower

\[
\text{add(…)} \times f
\]

\[
\{\}
\]

\[
\{\}
\]

\[
\{\}
\]

= insert  = cascade
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

![](image)
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

```
add(...) x f
{}
{a,b...n/threshold}
{}
{r,s...n/2*threshold}
{}
```

- = insert
- = cascade
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

\[ \text{cascade}(\text{deg}(v)) \geq \text{threshold} \]

add(...) x f
“Cascading vertices” by example

• Let’s use Twitter followers as an example
• Each letter represents a unique follower

Let's use Twitter followers as an example. Each letter represents a unique follower.
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

\[ \text{add(…)} \times f \]

\[ \{a,b\ldots n/\text{threshold}\} \]

\[ \{r,s\ldots n/2*\text{threshold}\} \]

\[ \{\} \]

\[ \{\} \]

\[ \{\} \]

\[ = \text{insert} \]

\[ = \text{cascade} \]
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

\[
\text{S} \quad \text{S1} \quad \text{S2} \quad \text{S3}
\]

- \( \text{add(…)} \times f \)
- \( \{a,b,…n/\text{threshold}\} \)
- \( \{r,s,…n/2^\text{threshold}\} \)

**Diagram Notes**
- Green circle = insert
- Purple circle = cascade
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

Let’s use Twitter followers as an example. Each letter represents a unique follower.
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

\[ \text{add}(\ldots) \times f \]

\{a, b, \ldots n/threshold\}

\{r, s, \ldots n/2*threshold\}

\{w, x, \ldots n/3*threshold\}

\( \text{add}(\ldots) \times f \)

\( \text{insert} \)

\( \text{cascade} \)
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

wrap and repeat

\[
\text{add}(\ldots) \times f
\]

\[
\emptyset
\]

\[
\{a,b\ldots n/\text{threshold}\}
\]

\[
\{\}
\]

\[
\{r,s\ldots n/2*\text{threshold}\}
\]

\[
\{\}
\]

\[
\{w,x\ldots n/3*\text{threshold}\}
\]

\[
\{\}
\]

= insert

= cascade
“Cascading vertices” by example

- Let’s use Twitter followers as an example
- Each letter represents a unique follower

\[
\text{add}(\ldots) \times f
\]

\{a, b, \ldots, n/\text{threshold}\}

\{r, s, \ldots, n/2\ast\text{threshold}\}

\{w, x, \ldots, n/3\ast\text{threshold}\}

\[= \text{insert}\]

\[= \text{cascade}\]
Aims of the Tesseract

1. **Implement distributed eventually consistent graph database**

2. Develop a distributed graph partitioning algorithm

3. Develop a computational model able to support both real time and batch processing on a distributed graph
Distributed computation
Localised calculations

Amortisation
Memoization
Amortisation

- Optimise to perform more “cheap” computations
- This allows us to occasionally pay the cost of more “expensive” operations such that they computationally balance out
- e.g. Checking data locally on a node vs querying over a network
Amortisation

- Optimise to perform more “cheap” computations
- This allows us to occasionally pay the cost of more “expensive” operations such that they computationally balance out
- e.g. Checking data locally on a node vs querying over a network

```
<table>
<thead>
<tr>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
```
Memoization

- Cache the results of computations
  - A luxury afforded by immutability
- Sacrifices disk space and memory
- Provides improved query performance
Memoization

- Cache the results of computations
  - A luxury afforded by immutability
- Sacrifices disk space and memory
- Provides improved query performance
Memoization

- Cache the results of computations
  - A luxury afforded by immutability
- Sacrifices disk space and memory
- Provides improved query performance

1st query ➔ Traverse n secs ➔ Cache results

2nd query ➔ Cache n/r
Wormhole traversals

- Immutability offers guarantees
- Place markers at every n vertex intervals
- When traversing, don’t visit every vertex, jump to markers instead.
- Markers at A, G, F, D
- By pass B, C, E during traversal, almost halving the time.
- The resulting data has any skipped vertex asynchronously fetched
- A key part of this is in the use of “Path summaries”
- Path summary is an optimisation that enables the runtime to skip network requests
- Allows traversal to continue locally and async request is made to gather the remote results
Going functional
Going functional

• Early implementation was in Haskell
Going functional

• Early implementation was in Haskell
• Why? Because it did everything I wanted.
Going functional

• Early implementation was in Haskell
• Why? Because it did everything I wanted.
• Later realised it’s not Haskell in particular I wanted
  • …but its semantics
    • Immutability
    • Purity
    • and some other stuff
    • and, well…functions!
Going functional

• Early implementation was in Haskell
• Why? Because it did everything I wanted.
• Later realised it’s not Haskell in particular I wanted
  • …but its semantics
    • Immutability
    • Purity
    • and some other stuff
    • and, well…functions!
• The whole graph thing is an optimisation problem
  • The properties of a purely functional language enables a run time to make a lot of assumptions
  • These assumptions open possibilities not otherwise available (some times by allowing us to pretend a problem isn’t there)
Distributed Query Model: TQL, Tesseract Query Language

• Haskell?
• …before you start sneaking out the back doors
• What would that even look like…?
Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- …before you start sneaking out the back doors
- What would that even look like…?

```sql
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3) E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3) E(v1 "respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```
Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- ...before you start sneaking out the back doors
- What would that even look like...?

```tql
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "siblings" v2) E(v1 "siblings" v3) E(v2 "siblings" v3)
E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
E(v1 <->"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "siblings" )
```

- What you’re looking at is TQL
Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- ...before you start sneaking out the back doors
- What would that even look like...?

v1 = V("Courtney")

v2 = V("Damion", age = 20)

v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "siblings" v2) E(v1 "siblings" v3) E(v2 "siblings" v3)
    E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
    E(v1 <->"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "siblings" )

- What you’re looking at is TQL
- a pure
Distributed Query Model: TQL, Tesseract Query Language

- Haskell?
- ...before you start sneaking out the back doors
- What would that even look like…?

```plaintext
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3) E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3) E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )
```

- What you’re looking at is TQL
  - a pure
  - functional language
Distributed Query Model: TQL, Tesseract Query Language

• Haskell?
• ...before you start sneaking out the back doors
• What would that even look like…?

v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

\[
\text{INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3) E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3) E(v1 <-"respects" v3) E(v1 "knows"-> $3) }
\]

\[
\text{SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )}
\]

• What you’re looking at is TQL
• a pure
• functional language
• it has type inferencing and all the cool functional widgets!
Distributed Query Model: TQL pt2
Distributed Query Model: TQL pt2

- How was that functional?
Distributed Query Model: TQL pt2

• How was that functional?
• It employed use of:
  • Functions - relation between a set of input and a set of permissible outputs
  • Monads - structures that allow you to define computation in terms of the steps necessary to obtain the results of the computation.
  • Monoids - a set with a single associative \((1+ 2) + 3 == 1 + (2+3)\) binary operation an identity element (an element where, when applied to any other in the set, the value of the other element remains unchanged. e.g. given * as the binary operation and the set \(S={1,2,3}\), 1 is the identity element since \(1 * 1 = 1, 2 * 1 = 2\) and \(3 * 1 = 3\))
  • Currying - where a function which takes multiple arguments is converted into a series of functions which take a single argument, the currying technique produces partially applied functions.
  • Higher order functions - functions which takes other functions as its parameter
  • Function composition - the process of making the result of one function the argument of another
Distributed Query Model: TQL pt2

• How was that functional?
• It employed use of:
  • Functions - relation between a set of input and a set of permissible outputs
  • Monads - structures that allow you to define computation in terms of the steps necessary to obtain the results of the computation.
  • Monoids - a set with a single associative \((1 + 2) + 3 == 1 + (2+3)\) binary operation an identity element (an element where, when applied to any other in the set, the value of the other element remains unchanged. e.g. given \(*\) as the binary operation and the set \(S=\{1,2,3\}\), 1 is the identity element since \(1 * 1 = 1, 2 * 1 = 2\) and \(3 * 1 = 3\) )
  • Currying - where a function which takes multiple arguments is converted into a series of functions which take a single argument, the currying technique produces partially applied functions.
  • Higher order functions - functions which takes other functions as its parameter
  • Function composition - the process of making the result of one function the argument of another

• Don’t believe me? Let’s look at a definition for “INSERT” shown on the previous slide
Distributed Query Model: TQL pt3
Distributed Query Model: TQL pt3

\[
\text{INSERT} :: ( (\text{String} \rightarrow (V\ldots) \rightarrow (E\ldots) \rightarrow \text{PartialTransform}) ) \rightarrow \text{Transform}
\]
Distributed Query Model: TQL pt3

INSERT :: ( (String -> (V…)) -> (E…) -> PartialTransform) ) -> Transform
Distributed Query Model: TQL pt3

\[
\text{INSERT} :: ( (\text{String} \to (V\ldots)) \to (E\ldots) \to \text{PartialTransform}) ) \to \text{Transform}
\]
Distributed Query Model: TQL pt3

\[
\text{INSERT} :: ( (\text{String} \rightarrow (\text{V} \ldots) \rightarrow (\text{E} \ldots) \rightarrow \text{PartialTransform}) ) \rightarrow \text{Transform}
\]
Distributed Query Model: TQL pt3

INSERT :: ( (String -> (V…)) -> (E…) -> PartialTransform) ) -> Transform

- Function name
- Graph namespace
- Vertex type
- Edge type
Distributed Query Model: TQL pt3

... = var-arg
+ Homogeneous

INSERT :: ( (String -> (V...)) -> (E...) -> PartialTransform ) -> Transform
Distributed Query Model: TQL pt3

```
... = var-arg
+ Homogeneous
```

```
INSERT :: ( (String -> (V…)) -> (E…) -> PartialTransform) ) -> Transform
```

Results of "INSERT"
Distributed Query Model: TQL pt3

**Function name**: INSERT

**Graph namespace**: (String -> (V...) -> (E...) -> PartialTransform) -> Transform

**Vertex type**: ... = var-arg + Homogeneous

**Edge type**: Result of "INSERT"

**Result of lambda function**: PartialTransform
Distributed Query Model: TQL pt3

\[ \text{INSERT} :: ( (\text{String} \to (V\ldots)) \to (E\ldots) \to \text{PartialTransform}) ) \to \text{Transform} \]

- Function name
- Graph namespace
- Vertex type
- Edge type
- Result of "INSERT"
- Result of lambda function

• Lambda function you say?
Distributed Query Model: TQL pt3

\[ \text{INSERT} :: ( (\text{String} \rightarrow (V\ldots)) \rightarrow (E\ldots) \rightarrow \text{PartialTransform}) ) \rightarrow \text{Transform} \]

- Result of "INSERT"
- Function name
- Graph namespace
- Result of lambda function
- Lambda function you say?
- Where, where?
Distributed Query Model: TQL pt3

\[
\text{INSERT} :: ( (\text{String} \rightarrow (V…)) \rightarrow (E…) \rightarrow \text{PartialTransform}) ) \rightarrow \text{Transform}
\]

- Lambda function you say?
- Where, where?
Distributed Query Model: TQL pt3

v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
    E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
    E(v1 <--"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )

INSERT :: ( (String -> (V...) -> (E...) -> PartialTransform) ) -> Transform

• Lambda function you say?
• Where, where?
Distributed Query Model: TQL pt3

v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
    E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
    E(v1 <--"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )

INSERT :: ( (String -> (V...) -> (E...) -> PartialTransform) ) -> Transform
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

\[
\begin{align*}
\text{INSERT INTO } & G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3) \\
& E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3) \\
& E(v1 <--"respects" v3) E(v1 "knows"-> $3) \\
\text{SELECT } & V[name, age] E \text{ FROM } G \text{ WHERE } E \text{ EXISTS AND ( E("knows") OR } E.\text{relationship} == "sibling" )
\end{align*}
\]

\[
\text{INSERT :: ( (String -> (V...) -> (E...) -> PartialTransform) ) -> Transform}
\]
Distributed Query Model: TQL pt3

\[
v1 = V("Courtney")
\]
\[
v2 = V("Damion", \text{ age } = 20)
\]
\[
v3 = V("Carlos")
\]

\[
\text{INSERT INTO } G v1 v2 V("Mark")\ E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3) \\
E(v1 "older"\rightarrow v2) E(v1 "older"\rightarrow v3) E(v2 "older"\rightarrow v3) \\
E(v1 \leftarrow "respects" v3) E(v1 "knows"\rightarrow £3)
\]

\[
\text{SELECT } V[\text{name, age}] E \text{ FROM } G \text{ WHERE } E \text{ EXISTS AND ( E("knows") OR } E.\text{relationship} == "sibling"
\]

\[
\text{INSERT :: } ((\text{String }\rightarrow (V\ldots)\rightarrow (E\ldots)\rightarrow \text{PartialTransform}) )\rightarrow \text{Transform}
\]

From here…

…to here!
Distributed Query Model: TQL pt3

v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
E(v1 "<"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )

INSERT :: ( (String -> (V...) -> (E...) -> PartialTransform) ) -> Transform
Distributed Query Model: TQL pt3

\[
v1 = V("Courtney")
\]
\[
v2 = V("Damion", \text{age} = 20)
\]
\[
v3 = V("Carlos")
\]

\[
\text{INSERT INTO } G \text{ v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3) E(v1 "older"\rightarrow v2) E(v1 "older"\rightarrow v3) E(v2 "older"\rightarrow v3) E(v1 \lnot"respects" v3) E(v1 "knows"\rightarrow 3)}
\]

\[
\text{SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )}
\]

\[
\text{INSERT :: ( (String -> (V…) -> (E…) -> PartialTransform) ) -> Transform}
\]

- Types are optional and are inferred using Hindley–Milner style type system
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
    E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
    E(v1 <--"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )

INSERT :: ( (String -> (V…) -> (E…) -> PartialTransform) ) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
Distributed Query Model: TQL pt3

v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
    E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
    E(v1 <--"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )

INSERT :: ( (String -> (V…) -> (E…) -> PartialTransform) ) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM
Distributed Query Model: TQL pt3

v1 = V("Courtney")

v2 = V("Damion", age = 20)

v3 = V("Carlos")

\[
\text{INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3) E(v1 "older" \rightarrow v2) E(v1 "older" \rightarrow v3) E(v2 "older" \rightarrow v3) E(v1 \leftarrow "respects" v3) E(v1 "knows" \rightarrow \$3) }
\]

\[
\text{SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E\.relationship == "sibling" )}
\]

\[
\text{INSERT :: ( (String \rightarrow (V\ldots) \rightarrow (E\ldots) \rightarrow \text{PartialTransform}) ) \rightarrow \text{Transform}}
\]

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM
- TQL comes with a useful “standard” library like most languages
Distributed Query Model: TQL pt3

v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )

INSERT :: ( (String -> (V...) -> (E...) -> PartialTransform) ) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM
- TQL comes with a useful “standard” library like most languages
- An “Algorithms & machine learning” module will ship as an add-on module
Distributed Query Model: TQL pt3

v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3) 
    E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3) 
    E(v1 <->"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )

INSERT :: ( (String -> (V…) -> (E…) -> PartialTransform) ) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM
- TQL comes with a useful “standard” library like most languages
- An “Algorithms & machine learning” module will ship as an add-on module
- Ability to define new modules/add or override functions
v1 = V("Courtney")
v2 = V("Damion", age = 20)
v3 = V("Carlos")

INSERT INTO G v1 v2 V("Mark") E(v1 "sibling" v2) E(v1 "sibling" v3) E(v2 "sibling" v3)
  E(v1 "older"-> v2) E(v1 "older"-> v3) E(v2 "older"-> v3)
  E(v1 <-"respects" v3) E(v1 "knows"-> $3)

SELECT V[name, age] E FROM G WHERE E EXISTS AND ( E("knows") OR E.relationship == "sibling" )

INSERT :: ( (String -> (V…) -> (E…) -> PartialTransform) ) -> Transform

- Types are optional and are inferred using Hindley–Milner style type system
- Functions are translated to “enriched” lambda calculus for reduction & evaluation
- Built on top of LLVM
- TQL comes with a useful “standard” library like most languages
- An “Algorithms & machine learning” module will ship as an add-on module
- Ability to define new modules/add or override functions
- Include additional modules (yours or a third party’s)
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations.
- Client interface remains consistent.
- While ongoing research can improve the run time without major client changes.
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations.
- Client interface remains consistent.
- While ongoing research can improve the run time without major client changes.

Tesseract runtime
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations.
- Client interface remains consistent.
- While ongoing research can improve the run time without major client changes.

Tesseract runtime

TQL
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations.
- Client interface remains consistent.
- While on-going research can improve the run time without major client changes.

Diagram:
- CRDTs
- Tesseract runtime
- TQL
Distributed Query Model: Runtime

• The model places a lot of additional work server side.
• Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations.
• Client interface remains consistent.
• While on going research can improve the run time without major client changes.

Diagram:
- Tesseract runtime
  - CRDTs
  - Cascading vertices

TQL
Distributed Query Model: Runtime

• The model places a lot of additional work server side.
• Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations
• Client interface remains consistent
• While on going research can improve the run time without major client changes

CRDTs

Cascading vertices

Tesseract runtime

Wormhole traversals

TQL
Distributed Query Model: Runtime

- The model places a lot of additional work server side.
- Previously enumerated properties enable the server to make a lot of assumptions and by proxy optimisations.
- Client interface remains consistent.
- While on-going research can improve the run time without major client changes.
Compaction & Garbage collection
Compaction & Garbage collection

- Immutability means we store data that’s no longer needed i.e. garbage
Compaction & Garbage collection

- Immutability means we store data that’s no longer needed i.e. garbage
- CRDTs can accumulate a large amount of garbage
  - This can be avoided by not keeping tombstones at all
  - Without tombstones the system is unable to do a consistent snapshot
- If snapshots are disabled, tombstones are not needed
- Short synchronisation are used out of the query path to do some clean up (currently evaluating RAFT for GC consensus)
Compaction & Garbage collection

- Immutability means we store data that’s no longer needed i.e. garbage
- CRDTs can accumulate a large amount of garbage
  - This can be avoided by not keeping tombstones at all
  - Without tombstones the system is unable to do a consistent snapshot
  - If snapshots are disabled, tombstones are not needed
  - Short synchronisation are used out of the query path to do some clean up (currently evaluating RAFT for GC consensus)
- Current work is modelled off of JVM’s generational collectors
Compaction & Garbage collection

- Immutability means we store data that’s no longer needed i.e. garbage
- CRDTs can accumulate a large amount of garbage
  - This can be avoided by not keeping tombstones at all
  - Without tombstones the system is unable to do a consistent snapshot
- If snapshots are disabled, tombstones are not needed
- Short synchronisation are used out of the query path to do some clean up (currently evaluating RAFT for GC consensus)
- Current work is modelled off of JVM’s generational collectors
- Algorithm needs more investigation…
Compaction & Garbage collection

- Immutability means we store data that’s no longer needed i.e. garbage
- CRDTs can accumulate a large amount of garbage
  - This can be avoided by not keeping tombstones at all
  - Without tombstones the system is unable to do a consistent snapshot
- If snapshots are disabled, tombstones are not needed
- Short synchronisation are used out of the query path to do some clean up (currently evaluating RAFT for GC consensus)
- Current work is modelled off of JVM’s generational collectors
- Algorithm needs more investigation…
- Compaction also serves as an opportunity to optimise data location
  - Write only means vertex properties and edges aren’t always next to each other in a data file
  - During compaction we re-arrange contents
  - Helps reduce the amount of work required by spindle disks to fetch a vertex’s data
First release due in 2-3 months

Will be Apache v2 Licensed

github.com/zcourts/Tesseract
End...

Questions?

Courtney Robinson
Google “zcourts”
courtney@zcourts.com
github.com/zcourts/Tesseract