External Memory k-Bisimulation Reduction of Big Graphs

Yongming Luo, George H.L. Fletcher, Jan Hidders, Yuqing Wu and Paul De Bra

The research of YW is supported by Research Foundation Flanders (FWO) during her sabbatical visit to Hasselt University, Belgium. The research of YL, GF, JH and PD is supported by the Netherlands Organisation for Scientific Research (NWO).
Bisimulation example
Bisimulation example
Bisimulation example

- For nodes 1 and 2, the transitions 'l' and 'w' show the bisimulation relationship.
- Nodes 3, 4, 5, and 6 are connected with 'l' transitions, indicating another bisimulation.
- The diagram on the right represents another instance of bisimulation with nodes 1, 2, 4, and 6.
Return nodes that are 2 hops away from 1.
Bisimulation example

Return nodes that are 2 hops away from 1
Outline

Motivation

Definition

Data structures

Construction algorithm

Maintenance algorithms

Experiment results

Conclusions and Future work
Motivation

Why graph bisimulation?

Graphs are ubiquitous, e.g., web graph, social network, linked open data.

Bisimulation is ubiquitous, e.g., modal logic, structural index construction, graph analytics.

Why $k$-bisimulation?

Consider nodes only within a local neighborhood of radius $k \geq 0$.

Pay-as-you-go nature, practical to use.

Equivalent to full bisimulation in the limit.

Why external memory based algorithms?

Huge graphs ↔ in-memory based algorithms.

Call for distributed/parallel/external-memory based solutions.

We propose the first external memory algorithm on $k$-bisimulation for arbitrary graph.
Motivation

Why *graph bisimulation*?

- Graphs are ubiquitous, e.g., web graph, social network, linked open data
- Bisimulation is ubiquitous, e.g., modal logic, structural index construction, graph analytics
Motivation

Why *graph bisimulation*?

- Graphs are ubiquitous, e.g., web graph, social network, linked open data
- Bisimulation is ubiquitous, e.g., modal logic, structural index construction, graph analytics

Why $k$-bisimulation?

- Consider nodes only within a local neighborhood of radius $k \geq 0$
- Pay-as-you-go nature, practical to use
- Equivalent to full bisimulation in the limit

Why external memory based algorithms?

- Huge graphs
- Call for distributed/parallel/external-memory based solutions

We propose the first external memory algorithm on $k$-bisimulation for arbitrary graph.
Why graph bisimulation?

- Graphs are ubiquitous, e.g., web graph, social network, linked open data
- Bisimulation is ubiquitous, e.g., modal logic, structural index construction, graph analytics

Why $k$-bisimulation?

- Consider nodes only within a local neighborhood of radius $k \geq 0$
- Pay-as-you-go nature, practical to use
- Equivalent to full bisimulation in the limit
Motivation

Why graph bisimulation?

- Graphs are ubiquitous, e.g., web graph, social network, linked open data
- Bisimulation is ubiquitous, e.g., modal logic, structural index construction, graph analytics

Why $k$-bisimulation?

- Consider nodes only within a local neighborhood of radius $k \geq 0$
- Pay-as-you-go nature, practical to use
- Equivalent to full bisimulation in the limit

Why external memory based algorithms?

- Huge graphs
- Call for distributed/parallel/external-memory based solutions
- We propose the first external memory algorithm on $k$-bisimulation for arbitrary graph.
Motivation

Why graph bisimulation?
- Graphs are ubiquitous, e.g., web graph, social network, linked open data
- Bisimulation is ubiquitous, e.g., modal logic, structural index construction, graph analytics

Why \( k \)-bisimulation?
- Consider nodes only within a local neighborhood of radius \( k \geq 0 \)
- Pay-as-you-go nature, practical to use
- Equivalent to full bisimulation in the limit

Why external memory based algorithms?
- Huge graphs \( \leftrightarrow \) in-memory based algorithms
- Call for distributed/parallel/external-memory based solutions
Motivation

Why graph bisimulation?
- Graphs are ubiquitous, e.g., web graph, social network, linked open data
- Bisimulation is ubiquitous, e.g., modal logic, structural index construction, graph analytics

Why \( k \)-bisimulation?
- Consider nodes only within a local neighborhood of radius \( k \geq 0 \)
- Pay-as-you-go nature, practical to use
- Equivalent to full bisimulation in the limit

Why external memory based algorithms?
- Huge graphs ↔ in-memory based algorithms
- Call for distributed/parallel/external-memory based solutions

We propose the first external memory algorithm on \( k \)-bisimulation for arbitrary graph.
Outline

Motivation

Definition

Data structures

Construction algorithm

Maintenance algorithms

Experiment results

Conclusions and Future work
Definitions

Definition (*k*-bisimilarity)

Let $k$ be a non-negative integer and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. $\lambda_N$ is a function from $N$ to a set of node labels $\mathcal{L}_N$, and $\lambda_E$ is a function from $E$ to a set of edge labels $\mathcal{L}_E$. Nodes $u, v \in N$ are called *k*-bisimilar (denoted as $u \approx^k v$), iff the following holds:

1. $\lambda_N(u) = \lambda_N(v)$,
2. if $k > 0$, then for any edge $(u, u') \in E$, there exists an edge $(v, v') \in E$, such that $u' \approx^{k-1} v'$ and $\lambda_E(u, u') = \lambda_E(v, v')$, and
3. if $k > 0$, then for any edge $(v, v') \in E$, there exists an edge $(u, u') \in E$, such that $v' \approx^{k-1} u'$ and $\lambda_E(v, v') = \lambda_E(u, u')$. 
Definitions

Definition (*k*-bisimilarity)
Let \( k \) be a non-negative integer and \( G = \langle N, E, \lambda_N, \lambda_E \rangle \) be a graph. \( \lambda_N \) is a function from \( N \) to a set of node labels \( \mathcal{L}_N \), and \( \lambda_E \) is a function from \( E \) to a set of edge labels \( \mathcal{L}_E \). Nodes \( u, v \in N \) are called *k*-bisimilar (denoted as \( u \approx^k v \)), iff the following holds:

1. \( \lambda_N(u) = \lambda_N(v) \),
2. if \( k > 0 \), then for any edge \( (u, u') \in E \), there exists an edge \( (v, v') \in E \), such that \( u' \approx^{k-1} v' \) and \( \lambda_E(u, u') = \lambda_E(v, v') \), and
3. if \( k > 0 \), then for any edge \( (v, v') \in E \), there exists an edge \( (u, u') \in E \), such that \( v' \approx^{k-1} u' \) and \( \lambda_E(v, v') = \lambda_E(u, u') \).

Bisimulation partition
Node set \( N \) can be partitioned under equivalence relation \( \approx^k \). We call each set of *k*-bisimilar nodes a partition block.
Example of $k$-bisimilarity

$1 \sim l \sim 3 \sim l \sim 5$

$2 \sim w \sim 4 \sim l \sim 6$
Example of $k$-bisimilarity

$k = 0$

\[ \begin{array}{c}
\begin{array}{c}
\text{1} \quad \text{2} \quad \text{3} \quad \text{4} \quad \text{5} \quad \text{6}
\end{array}
\end{array} \]
Example of $k$-bisimilarity

$k = 0$

local info
Example of $k$-bisimilarity

$k = 0$

Local info

$k = 1$
Partition Identifier
A $k$-partition identifier for graph $G = \langle N, E, \lambda_N, \lambda_E \rangle$ and $k \geq 0$ is a set of $k + 1$ functions $\mathcal{P} = \{\text{pid}_0, \ldots, \text{pid}_k\}$ such that, for each $0 \leq i \leq k$, $\text{pid}_i$ is a function from $N$ to the integers, and, for all nodes $u, v \in N$, it holds that $\text{pid}_i(u) = \text{pid}_i(v)$ iff $u \approx^i v$. 
Partition identifier and signature

Partition Identifier
A \(k\)-partition identifier for graph \(G = \langle N, E, \lambda_N, \lambda_E \rangle\) and \(k \geq 0\) is a set of \(k + 1\) functions \(P = \{\text{pid}_0, \ldots, \text{pid}_k\}\) such that, for each \(0 \leq i \leq k\), \(\text{pid}_i\) is a function from \(N\) to the integers, and, for all nodes \(u, v \in N\), it holds that \(\text{pid}_i(u) = \text{pid}_i(v)\) iff \(u \approx^i v\).

Signature
Let \(G = \langle N, E, \lambda_N, \lambda_E \rangle\) be a graph, \(k \geq 0\), and \(P = \{\text{pid}_0, \ldots, \text{pid}_k\}\) be a \(k\)-partition identifier for \(G\). The \(k\) bisimulation signature of node \(u \in N\) is the pair \(\text{sig}^k(u) = (\text{pid}_0(u), L)\) where:

\[
L = \begin{cases} 
\emptyset & \text{if } k = 0, \\
\{ (\lambda_E(u, u'), \text{pid}_{k-1}(u')) | (u, u') \in E \} & \text{if } k > 0.
\end{cases}
\]
The basic idea of our algorithms

Proposition

\[ \forall u, v \in \mathbb{N}, \text{pid}_k(u) = \text{pid}_k(v) \text{ if and only if } \text{sig}_k(u) = \text{sig}_k(v) (k \geq 0). \]
**The basic idea of our algorithms**

**Proposition**

\[ \forall u, v \in N, \text{pid}_k(u) = \text{pid}_k(v) \text{ if and only if } \text{sig}_k(u) = \text{sig}_k(v) \ (k \geq 0). \]

**Basic idea**

- Maintain the one-to-one mapping between partition IDs and signatures
- Construct signatures, get partition IDs, iteratively
The basic idea - example

\[
\begin{array}{cccc}
\text{nid} & \text{pid}_0(\text{nid}) & \text{sig}_1(\text{nid}) & \text{pid}_1(\text{nid}) & \text{sig}_2(\text{nid}) & \text{pid}_2(\text{nid}) \\
1 & A & A, \{(w, A), (l, B)\} & C & A, \{(w, C), (l, D)\} & G \\
2 & A & A, \{(w, A), (l, B)\} & C & A, \{(w, C), (l, E)\} & H \\
3 & B & B, \{(l, B)\} & D & B, \{(l, F)\} & I \\
4 & B & B, \{} & E & B, \{} & J \\
5 & B & B, \{(l, A)\} & F & B, \{(l, C)\} & K \\
6 & B & B, \{(l, A)\} & F & B, \{(l, C)\} & K \\
\end{array}
\]
The basic idea - example

<table>
<thead>
<tr>
<th>$nid$</th>
<th>$pid_0(nid)$</th>
<th>$sig_1(nid)$</th>
<th>$pid_1(nid)$</th>
<th>$sig_2(nid)$</th>
<th>$pid_2(nid)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>A,{(w, A), (l, B)}</td>
<td>C</td>
<td>A,{(w, C), (l, D)}</td>
<td>G</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>A,{(w, A), (l, B)}</td>
<td>C</td>
<td>C</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>B,{(l, B)}</td>
<td>D</td>
<td>B,{(l, F)}</td>
<td>I</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>B,{}</td>
<td>E</td>
<td>B,{}</td>
<td>J</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>B,{(l, A)}</td>
<td>F</td>
<td>B,{(l, C)}</td>
<td>K</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>B,{(l, A)}</td>
<td>F</td>
<td>B,{(l, C)}</td>
<td>K</td>
</tr>
</tbody>
</table>
The basic idea - example

<table>
<thead>
<tr>
<th>nid</th>
<th>pid₀(nid)</th>
<th>sig₁(nid)</th>
<th>pid₁(nid)</th>
<th>sig₂(nid)</th>
<th>pid₂(nid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>A, {(w, A), (l, B)}</td>
<td>C</td>
<td>A, {(w, C), (l, D)}</td>
<td>G</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>A, {(w, A), (l, B)}</td>
<td>C</td>
<td>A, {(w, C), (l, E)}</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>B, {(l, B)}</td>
<td>D</td>
<td>B, {(l, F)}</td>
<td>I</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>B, {}</td>
<td>E</td>
<td>B, {}</td>
<td>J</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>B, {(l, A)}</td>
<td>F</td>
<td>B, {(l, C)}</td>
<td>K</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>B, {(l, A)}</td>
<td>F</td>
<td>B, {(l, C)}</td>
<td>K</td>
</tr>
</tbody>
</table>
The basic idea - example

<table>
<thead>
<tr>
<th>nid</th>
<th>$pid_0(nid)$</th>
<th>$sig_1(nid)$</th>
<th>$pid_1(nid)$</th>
<th>$sig_2(nid)$</th>
<th>$pid_2(nid)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>$(w, A), (l, B)$</td>
<td>C</td>
<td>$(w, C), (l, D)$</td>
<td>G</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>$(w, A), (l, B)$</td>
<td>C</td>
<td>$(w, C), (l, E)$</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>$(l, B)$</td>
<td>D</td>
<td>$B, {l, F}$</td>
<td>I</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>${}$</td>
<td>E</td>
<td>$B, {}$</td>
<td>J</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>$(l, A)$</td>
<td>F</td>
<td>$B, {(l, C)}$</td>
<td>K</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>$(l, A)$</td>
<td>F</td>
<td>$B, {(l, C)}$</td>
<td>K</td>
</tr>
</tbody>
</table>
Outline

Motivation

Definition

Data structures

Construction algorithm

Maintenance algorithms

Experiment results

Conclusions and Future work
Data structures

- Node table $N_t$ and edge table $E_t$

Note: There are many possibilities for implementing these data structures, we choose the simple yet effective implementations, leaving the door open for further optimization/tuning.
Data structures

- Node table $N_t$ and edge table $E_t$
  - records stored sequentially on disk
  - store inputs and intermediate results

Note: There are many possibilities for implementing these data structures, we choose the simple yet effective implementations, leaving the door open for further optimization/tuning.
Data structures

- Node table $N_t$ and edge table $E_t$
  - records stored sequentially on disk
  - store inputs and intermediate results
- Signature storage facility $S$

Note: There are many possibilities for implementing these data structures, we choose the simple yet effective implementations, leaving the door open for further optimization/tuning.
Data structures

- Node table $N_t$ and edge table $E_t$
  - records stored sequentially on disk
  - store inputs and intermediate results

- Signature storage facility $S$
  - implement the one-to-one mapping between signatures and partition IDs
  - $pid_k(u) \leftarrow S.insert(sig_k(u))$
  - think of it as a dictionary for the moment

Note: There are many possibilities for implementing these data structures, we choose the simple yet effective implementations, leaving the door open for further optimization/tuning.
Data structures

- Node table $N_t$ and edge table $E_t$
  - records stored sequentially on disk
  - store inputs and intermediate results
- Signature storage facility $S$
  - implement the one-to-one mapping between signatures and partition IDs
  - $\text{pid}_k(u) \leftarrow S.\text{insert}(\text{sig}_k(u))$
  - think of it as a dictionary for the moment

Note:
There are many possibilities for implementing these data structures, we choose the simple yet effective implementations, leaving the door open for further optimization/tuning.
Outline

Motivation

Definition

Data structures

Construction algorithm

Maintenance algorithms

Experiment results

Conclusions and Future work
Construction algorithm, example run, 1 iteration

1. sort $N_t$ on $nid$
2. sort $E_t$ on $tid$
3. merge join $N_t$ and $E_t$ on $nid$ and $tid$, fill in $E_t.pid_{old\_tid}$
4. sort $E_t$ on $sid$, project on $(sid, eLabel, pid_{old\_tid})$, remove duplicates, get $F$
5. merge join $N_t$ and $F$ on $nid$ and $sid$, get signature for each node
6. call $S.insert()$ for each such signature, get new $pid$ back
Construction algorithm, example run, 1 iteration

1. sort \( N_t \) on \( nid \)
2. sort \( E_t \) on \( tid \)
3. merge join \( N_t \) and \( E_t \) on \( nid \) and \( tid \), fill in \( E_t.pid_{old\_tid} \)
4. sort \( E_t \) on \( sid \), project on \((sid, eLabel, pid_{old\_tid})\), remove duplicates, get \( F \)
5. merge join \( N_t \) and \( F \) on \( nid \) and \( sid \), get signature for each node
6. call \( S.insert() \) for each such signature, get new \( pid \) back

\[
N_t
\]

<table>
<thead>
<tr>
<th>( nid )</th>
<th>( nLabel )</th>
<th>( pid_{0_nid} )</th>
<th>( sig_1(nid) )</th>
<th>( pid_{1_nid} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( M )</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( M )</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( P )</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( P )</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( P )</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( P )</td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Construction algorithm, example run, 1 iteration

1. sort $N_t$ on $nid$
2. sort $E_t$ on $tid$
3. merge join $N_t$ and $E_t$ on $nid$ and $tid$, fill in $E_t.pid_{old\_tid}$
4. sort $E_t$ on $sid$, project on $(sid,eLabel,pid_{old\_tid})$, remove duplicates, get $F$
5. merge join $N_t$ and $F$ on $nid$ and $sid$, get signature for each node
6. call $S.insert()$ for each such signature, get new $pid$ back

<table>
<thead>
<tr>
<th>$nid$</th>
<th>$nLabel$</th>
<th>$pid_{0_nid}$</th>
<th>$sig_1(nid)$</th>
<th>$pid_{1_nid}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$M$</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$M$</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$sid$</th>
<th>$eLabel$</th>
<th>$tid$</th>
<th>$pid_{old_tid}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$l$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$w$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$w$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>$l$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>$l$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>$l$</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>$l$</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Construction algorithm, example run, 1 iteration

1. sort $N_t$ on $nid$
2. sort $E_t$ on $tid$
3. merge join $N_t$ and $E_t$ on $nid$ and $tid$, fill in $E_t.pid_{old\_tid}$
4. sort $E_t$ on $sid$, project on $(sid,eLabel,pid_{old\_tid})$, remove duplicates, get $F$
5. merge join $N_t$ and $F$ on $nid$ and $sid$, get signature for each node
6. call $S.insert()$ for each such signature, get new $pid$ back

<table>
<thead>
<tr>
<th>$nid$</th>
<th>nLabel</th>
<th>$pid_{0_nid}$</th>
<th>sig$_1(nid)$</th>
<th>$pid_{1_nid}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$M$</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$M$</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$sid$</th>
<th>eLabel</th>
<th>$tid$</th>
<th>$pid_{old_tid}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$l$</td>
<td>1</td>
<td>$A$</td>
</tr>
<tr>
<td>1</td>
<td>$w$</td>
<td>2</td>
<td>$A$</td>
</tr>
<tr>
<td>2</td>
<td>$w$</td>
<td>2</td>
<td>$A$</td>
</tr>
<tr>
<td>6</td>
<td>$l$</td>
<td>2</td>
<td>$A$</td>
</tr>
<tr>
<td>1</td>
<td>$l$</td>
<td>3</td>
<td>$B$</td>
</tr>
<tr>
<td>2</td>
<td>$l$</td>
<td>4</td>
<td>$B$</td>
</tr>
<tr>
<td>3</td>
<td>$l$</td>
<td>5</td>
<td>$B$</td>
</tr>
</tbody>
</table>
Construction algorithm, example run, 1 iteration

1. sort $N_t$ on $nid$
2. sort $E_t$ on $tid$
3. merge join $N_t$ and $E_t$ on $nid$ and $tid$, fill in $E_t.pid_{old\_tid}$
4. sort $E_t$ on $sid$, project on $(sid, eLabel, pid_{old\_tid})$, remove duplicates, get $F$
5. merge join $N_t$ and $F$ on $nid$ and $sid$, get signature for each node
6. call $S.insert()$ for each such signature, get new $pid$ back

$N_t$

<table>
<thead>
<tr>
<th>$nid$</th>
<th>$nLabel$</th>
<th>$pid_{0_nid}$</th>
<th>$sig_1(nid)$</th>
<th>$pid_{1_nid}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$M$</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$M$</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$P$</td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$F$

<table>
<thead>
<tr>
<th>$sid$</th>
<th>$eLabel$</th>
<th>$pid_{old_tid}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$w$</td>
<td>$A$</td>
</tr>
<tr>
<td>1</td>
<td>$l$</td>
<td>$B$</td>
</tr>
<tr>
<td>2</td>
<td>$w$</td>
<td>$A$</td>
</tr>
<tr>
<td>2</td>
<td>$l$</td>
<td>$B$</td>
</tr>
<tr>
<td>3</td>
<td>$l$</td>
<td>$B$</td>
</tr>
<tr>
<td>5</td>
<td>$l$</td>
<td>$A$</td>
</tr>
<tr>
<td>6</td>
<td>$l$</td>
<td>$A$</td>
</tr>
</tbody>
</table>
Construction algorithm, example run, 1 iteration

1. sort $N_t$ on $nid$
2. sort $E_t$ on $tid$
3. merge join $N_t$ and $E_t$ on $nid$ and $tid$, fill in $E_t.pid_{old\_tid}$
4. sort $E_t$ on $sid$, project on $(sid,eLabel,pid_{old\_tid})$, remove duplicates, get $F$
5. merge join $N_t$ and $F$ on $nid$ and $sid$, get signature for each node
6. call $S.insert()$ for each such signature, get new $pid$ back

<table>
<thead>
<tr>
<th>$N_t$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nid$</td>
<td>$sid$</td>
</tr>
<tr>
<td>$nLabel$</td>
<td>$pid_{0_nid}$</td>
</tr>
<tr>
<td>$1$</td>
<td>$M$</td>
</tr>
<tr>
<td>$2$</td>
<td>$M$</td>
</tr>
<tr>
<td>$3$</td>
<td>$P$</td>
</tr>
<tr>
<td>$4$</td>
<td>$P$</td>
</tr>
<tr>
<td>$5$</td>
<td>$P$</td>
</tr>
<tr>
<td>$6$</td>
<td>$P$</td>
</tr>
</tbody>
</table>
Construction algorithm, example run, 1 iteration

1. sort $N_t$ on $nid$
2. sort $E_t$ on $tid$
3. merge join $N_t$ and $E_t$ on $nid$ and $tid$, fill in $E_t.pid_{old\_tid}$
4. sort $E_t$ on $sid$, project on $(sid, eLabel, pid_{old\_tid})$, remove duplicates, get $F$
5. merge join $N_t$ and $F$ on $nid$ and $sid$, get signature for each node
6. call $S.insert()$ for each such signature, get new $pid$ back

<table>
<thead>
<tr>
<th>nid</th>
<th>nLabel</th>
<th>pid$<em>0$$</em>{\text{nid}}$</th>
<th>sig$_1$(nid)</th>
<th>pid$<em>1$$</em>{\text{nid}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$M$</td>
<td>$A$</td>
<td>$A, {(w, A), (l, B)}$</td>
<td>$C$</td>
</tr>
<tr>
<td>2</td>
<td>$M$</td>
<td>$A$</td>
<td>$A, {(w, A), (l, B)}$</td>
<td>$C$</td>
</tr>
<tr>
<td>3</td>
<td>$P$</td>
<td>$B$</td>
<td>$B, {(l, B)}$</td>
<td>$D$</td>
</tr>
<tr>
<td>4</td>
<td>$P$</td>
<td>$B$</td>
<td>$B, {}$</td>
<td>$E$</td>
</tr>
<tr>
<td>5</td>
<td>$P$</td>
<td>$B$</td>
<td>$B, {(l, A)}$</td>
<td>$F$</td>
</tr>
<tr>
<td>6</td>
<td>$P$</td>
<td>$B$</td>
<td>$B, {(l, A)}$</td>
<td>$F$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>eLabel</th>
<th>pid$_{old_tid}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$w$</td>
<td>$A$</td>
</tr>
<tr>
<td>1</td>
<td>$l$</td>
<td>$B$</td>
</tr>
<tr>
<td>2</td>
<td>$w$</td>
<td>$A$</td>
</tr>
<tr>
<td>2</td>
<td>$l$</td>
<td>$B$</td>
</tr>
<tr>
<td>3</td>
<td>$l$</td>
<td>$B$</td>
</tr>
<tr>
<td>5</td>
<td>$l$</td>
<td>$A$</td>
</tr>
<tr>
<td>6</td>
<td>$l$</td>
<td>$A$</td>
</tr>
</tbody>
</table>
Construction algorithm, example run, 1 iteration

1. sort \( N_t \) on \( nid \)
2. sort \( E_t \) on \( tid \)
3. merge join \( N_t \) and \( E_t \) on \( nid \) and \( tid \), fill in \( E_t.pid_{old\_tid} \)
4. sort \( E_t \) on \( sid \), project on \((sid,eLabel,pid_{old\_tid})\), remove duplicates, get \( F \)
5. merge join \( N_t \) and \( F \) on \( nid \) and \( sid \), get signature for each node
6. call \( S.insert() \) for each such signature, get new \( pid \) back

Complexity analysis result

- I/O complexity: \( O(k \cdot sort(|E_t|) + k \cdot scan(|N_t|) + sort(|N_t|)) \)
Outline

Motivation

Definition

Data structures

Construction algorithm

Maintenance algorithms

Experiment results

Conclusions and Future work
Maintenance algorithms

- Off-line, bulk update
- `add_nodes()`, `add_edges()` (node and edge deletions are the reverse procedures)
- Following the similar idea as construction algorithm
- Use a disk-based priority queue to propagate changes
- Need not to recompute everything from scratch but only touch the essential part
- I/O complexity: $O(k \cdot \text{sort}(|E_t|) + k \cdot \text{sort}(|N_t|))$
Outline

Motivation

Definition

Data structures

Construction algorithm

Maintenance algorithms

Experiment results

Conclusions and Future work
# Datasets

Description and statistics of the experiment datasets

| Data Name   | Description                                      | |Node|   | |Edge|   |
|-------------|--------------------------------------------------|---|-----|---|-----|
| Twitter     | A following relationship graph of Twitter        | 41.65M | 1468.37M |
| Jamendo     | A repository of music metadata in RDF format     | 0.49M | 1.05M |
| LinkedMDB   | A repository of movie metadata in RDF format     | 2.33M | 6.15M |
| DBLP        | An RDF format DBLP dump                          | 23M | 50.2M |
| WikiLinks   | A page-to-page linking graph of Wikipedia        | 5.71M | 130.16M |
| DBPedia     | An early RDF dump of DBPedia                     | 38.62M | 115.3M |
| SP2B        | A RDF data generator for arbitrarily large DBLP-like data | 280.9M | 500M |
| BSBM        | A RDF data generator for e-commerce use case     | 8.89M | 34.87M |
Running time for the construction algorithm for 10 iterations, single machine comparing with MR algorithm on 72 cluster machines
Construction algorithm - scalability

Time and I/O spent on each edge on average for SP2B datasets with different size ($k = 10$)
Edge updates vs. construction algorithm

I/O (left) and time (right) improvement ratio $\frac{\text{cost} (\text{BUILD}_B\text{ISIM}())}{\text{cost} (\text{ADD}_E\text{DGES}())}$ for batch edge updates ($k = 10$)
Outline

Motivation

Definition

Data structures

Construction algorithm

Maintenance algorithms

Experiment results

Conclusions and Future work
Conclusion and Future work

- We present the first known I/O efficient algorithms for constructing and maintaining $k$-bisimulation partitions on massive disk-resident graphs.

- Theoretical analysis and extensive empirical study have shown that our algorithms are not only efficient and practical to use, but also scale well with the size of the graph.

- Many interesting things to further investigate:
  - Alternative data structures, join algorithms
  - Bisimulation partition graph properties (GRADES@SIGMOD 2013)
  - Cope with new parallel platforms (Pregel, Graphlab, ...)
  - And many more
Conclusion and Future work

- We present the first known I/O efficient algorithms for constructing and maintaining $k$-bisimulation partitions on massive disk-resident graphs.
- Theoretical analysis and extensive empirical study have shown that our algorithms are not only efficient and practical to use, but also scale well with the size of the graph.
- Many interesting things to further investigate:
  - Alternative data structures, join algorithms
  - Bisimulation partition graph properties (GRADES@SIGMOD 2013)
  - Cope with new parallel platforms (Pregel, Graphlab, ...)
  - And many more
Conclusion and Future work

- We present the first known I/O efficient algorithms for constructing and maintaining $k$-bisimulation partitions on massive disk-resident graphs.
- Theoretical analysis and extensive empirical study have shown that our algorithms are not only efficient and practical to use, but also scale well with the size of the graph.
- Many interesting things to further investigate:
  - Alternative data structures, join algorithms
  - Bisimulation partition graph properties (*GRADES@SIGMOD 2013*)
  - Cope with new parallel platforms (Pregel, Graphlab, ...)
  - And many more
Thank you! Q&A

For more information, just google “seeqr project”
Related work (1)

- Hellings’ work studies I/O efficient algorithms for computing full bisimulation for DAGs, and applying the techniques to several variants of XML structural indexes.

- Apart from that, there has been to our knowledge no work on computing bisimulation and \( k \)-bisimulation partitions on arbitrary graph structures in external memory.

<table>
<thead>
<tr>
<th></th>
<th>Hellings’ work</th>
<th>Our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different problem</td>
<td>full bisimulation on DAG</td>
<td>localized bisimulation on arbitrary graph</td>
</tr>
<tr>
<td>Different techniques</td>
<td>rank based, time-forward processing</td>
<td>iteration based</td>
</tr>
</tbody>
</table>

J. Hellings, G. H. L. Fletcher, and H. Haerkort. Efficient external-memory bisimulation on DAGs.

In *SIGMOD*, pages 553–564, Scottsdale, AZ, USA, 2012
Related work (2)

- First MapReduce-based algorithm to calculate $k$-bisimulation, scalable
- Efficient, with an order of magnitude speed-up to single-machine algorithm
- Skew-resistant


At the same time:

A. Schätzle, A. Neu, G. Lausen, and M. Przyjaciel-Zablocki. Large-scale bisimulation of rdf graphs.
In *SWIM*, New York, NY, USA, 2013
The I/O efficient implementation of S

- There are I/O efficient string sorting algorithms for arbitrary string length
- After we sort signatures, we scan and assign pids for them ($\text{scan} (E_t)$)
- Treat short and long strings differently. Sort fragments of long strings and merge

Resemblance of PageRank

BSP version of the algorithm

For each node $v$:
   send $v$’s pid to its parent
---
For each node $v$:
   construct the signature from neighbors,
   send $(v, \text{signature})$ to map
---
For each signature in map:
   assign a unique pid,
   send pid to all related nodes
Reference

On sorting strings in external memory.  

J. Hellings, G. H. L. Fletcher, and H. Haverkort.  
Efficient external-memory bisimulation on DAGs.  

Y. Luo, Y. de Lange, G. H. L. Fletcher, P. De Bra, J. Hidders, and Y. Wu.  
Bisimulation Reduction of Big Graphs on MapReduce.  

A. Schätzle, A. Neu, G. Lausen, and M. Przyjaciel-Zablocki.  
Large-scale bisimulation of rdf graphs.  