Introduction

• Linked Shared Data Structure are:
  – Hard to change.
  – Hard to respect all invariants.
  – Concurrency make it harder.
  – Easy to get wrong.
Introduction

- Auto-construction of efficient Data Structures from easy High-Level interface.
Introduction

• New approach:
  - Data Structures are manipulated at a high-level code as *relations and Functional dependencies* (Invariants).
  - A Data Structures Designer provides *Decomposition* (Data Structures representation in memory).
  - Conversion from (Decomposition, Relation) to low-level implementation is done by the RelC Compiler...
Summary

- Relational Abstraction:
  - How to represent and manipulate Data as relations.

- Decomposition and Decomposition Instances:
  - How to represent relations in memory as a collection of data-structures.

- Querying and Updating Decomposed Relations:
  - How to compile the relational operations into Code tailored to a particular decomposition.

- Autotuner:
  - The best decomposition to a relation.

- Experiments:
  - The “RelC” Compiler
Example

Operating System process scheduler..

- Each process has:
  - A name space “\textit{ns}”
  - An ID “\textit{pid}” (Integer)
  - A “\textit{State}” (R, S)
  - “\textit{Cpu}”: Time consumed (Integer)
Relational abstraction

Definition:

- A relational specification is a set of Columns “C” and Functional Dependencies “▲”.

- In the scheduler example, we model the processes as a relation with columns \{ns, pid, state, cpu\}.

- For all meaningful sets of process:
  - \(ns, pid \rightarrow state, cpu\): relational specification
Relational abstraction

**Functional Dependencies:**
- C1 → C2 : FD of the relation “r”
  - <=> ∀ (t1, t2) ∈ r | ∀ c ∈ C1, t1(c) = t2(c)
  - => ∀ c' ∈ C2, t1(c') = t2(c')
- Scheduler example: {ns, pid} → {cpu, state}

**Relational Algebra:** Usual meaning of:
- Intersection: ∩
- Union: ∪
- Set different: \ 
- Join: ⊳⊲
Relational abstraction

Relational Operations:
- Relations are ML-Like references to a set of tuples:
  - Ref x : Relation with one tuple “x”
  - !r : fetches the current values of relation “r”
  - r ← v: sets the current value of “r” to “v”
- 5 Operations to manipulate relations:
  - Empty = ref (∅) : create an empty relation
  - Insert (r, t) = r ← !r ∪ {t}
  - Remove (r, s) = !r \ {t ∈!r | t ⊇ s}
  - Update (r, s, u) = r ← {if t ⊇ s, then t < u, else t | t ∈ !r}
  - Query (r, s, C) = projection_onto_C{t ∈!r | t ⊇ s}: Gives Values of Columns
    C in t | t ⊇ s & t ∈ r.
Relational abstraction

In Scheduler example:

<table>
<thead>
<tr>
<th>ns</th>
<th>pid</th>
<th>cpu</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>23</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>15</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>52</td>
<td>25</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>15</td>
<td>S</td>
</tr>
</tbody>
</table>

Insert <ns:1, pid:12, cpu:10, state: R>

<table>
<thead>
<tr>
<th>ns</th>
<th>pid</th>
<th>cpu</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>23</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>15</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>52</td>
<td>25</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>15</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>10</td>
<td>R</td>
</tr>
</tbody>
</table>
Relational abstraction

In Scheduler example:

<table>
<thead>
<tr>
<th>ns</th>
<th>pid</th>
<th>cpu</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>23</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>15</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>52</td>
<td>25</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>15</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>10</td>
<td>R</td>
</tr>
</tbody>
</table>

Remove <ns:1, pid:52>

<table>
<thead>
<tr>
<th>ns</th>
<th>pid</th>
<th>cpu</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>23</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>15</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>15</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>10</td>
<td>R</td>
</tr>
</tbody>
</table>
Relational abstraction

In Scheduler example:

<table>
<thead>
<tr>
<th>ns</th>
<th>pid</th>
<th>cpu</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>23</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>15</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>15</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>10</td>
<td>R</td>
</tr>
</tbody>
</table>

Update <ns:2, pid:52> <State: S>
Update <ns:2, pid:42> <State: R>

<table>
<thead>
<tr>
<th>ns</th>
<th>pid</th>
<th>cpu</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>23</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>15</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>15</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>10</td>
<td>R</td>
</tr>
</tbody>
</table>
Relational abstraction

In Scheduler example:

<table>
<thead>
<tr>
<th>ns</th>
<th>pid</th>
<th>cpu</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>23</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>15</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>15</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>10</td>
<td>R</td>
</tr>
</tbody>
</table>

Query <State:R> {ns, pid, cpu} = [ <ns:1, pid: 42, cpu: 23>, <ns: 2, pid: 42, cpu: 15> <ns: 1, pid: 12, cpu: 10>]
Decomposition and Decomposition Instances

- About how to represent relations as primitive Data Structures.

- Ensure that the Low-Level representation of a relation faithfully implements its high-Level specification.
  - Decomposition:
    - Static description of the data structures.
  - Decomposition instance:
    - The dynamic counterpart part of Decomposition.
Decomposition

- Rooted, directed acyclic graph → How to represent a particular relation.
- Every node of a decomposition correspond to a set of nodes in an instance representation.
- Each edge of the decomposition describes a way of breaking up a relation into a set of smaller relations.
- Each node is a sub-relation.
Decomposition

{ns} -> ( {pid} -> {cpu} )

{state} -> ( {ns, pid} -> {cpu} )
Memory decomposition (Left)
Memory decomposition (Right)
Decomposition instance

- Rooted, directed acyclic graph → How to represent a relational specification.
- Every sub-graph rooted at each node of the decomposition describes how too represent part of the original relation.
## Decomposition instance

<table>
<thead>
<tr>
<th>ns</th>
<th>pid</th>
<th>cpu</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>23</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>15</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>52</td>
<td>25</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>15</td>
<td>S</td>
</tr>
</tbody>
</table>
Formal representation of Relation's Decomposition

- Let-binding language:
  - Let $v: B \rightarrow C$ in $d \leftrightarrow$ allow us to share instances of sub-relation “v” (decompositions “p”) between multiple parts of decomposition “d”

```
Let w: \{ns, pid\} \rightarrow \{cpu\} = \{cpu\} in
Let y: \{ns\} \rightarrow \{pid, cpu\} = \{pid\} \rightarrow w in
Let z: \{state\} \rightarrow \{ns, pid, cpu\} = \{ns, pid\} \rightarrow w in
Let x: \emptyset \rightarrow \{ns, pid, cpu, state\} = (\{ns\} \rightarrow y) >> (\{state\} \rightarrow z) in x
```
Adequacy conditions

- Not every relation can be represented by every decomposition.
- A decomposition can only represent relations with specific columns satisfying certain functional dependencies.
- All columns are represented.
- Nodes are consistent with functional dependencies.
- Columns bound to paths leading to a common node must functionally determine each other.
Adequacy conditions

\{\text{ns, pid} \leftrightarrow \{\text{state, ns, cpu}\}\}
Adequacy conditions

Columns bound on a path to an object \( x \) must functionally determine columns bound on any other path to \( x \) must functionally determine each other.
Querying and Updating Decomposed Relations:

- Queries:
  - Definition:
    Given a relation “r”, a tuple “t”, and a set of columns “C”.
    A query returns the projection onto Column C of the Tuples of “r” that match tuple “t”;

- Query's Implementation in two stages:
  - Query planning: To find the most efficient execution plan q for a query.
  - Query execution: To evaluate a particular query plan over a decomposition instance.
Querying and Updating Decomposed Relations:

- **Query plan:**

  A tree of query plan operators:
  
  \[ q = \text{qunit} | \text{qscan}(q) | \text{qlookup}(q) | \text{qlr}(q, lr) | \text{qjoin}(q1, q2, lr) | \]

  - **Unit:** returns the unique tuple represented by a unit decomposition instance if that tuple matches t. it return thz umpty otherwise.
  
  - **Scan:** \( \text{qscan}(q) \) invokes operator “q” for each child node
  
  - **Lookup:** \( \text{qlookup}(q) \) looks up a particular set of key value in a map decomposition, the operator q invoked on the resulting sub-decomposition.
  
  - **Left/Right:** \( \text{qlr}(q, lr) \) invoke operator q on the left/right of a join.
  
  - **Join:** \( \text{qjoin}(q1, q2, lr) \) operator performs a join across both sides of a join decomposition: if \( lr = \text{left} \rightarrow \text{“q1”} \) executed on the left side of the join decomposition, than “q2” executed on the right side of the join for each tuple returned by tuple “q1”.
  
  05/06/14
Querying and Updating Decomposed Relations:

• Example:
  - Query \( r \langle ns:1, pid: 23 \rangle \{cpu\} \):
    
    Possible query plan: \( q = qlr(qlookup(qlookup(qunit)), \text{left}) \)

\( dqexec q d \langle ns:7, pid 42 \rangle : \) query execution..
Querying and Updating Decomposed Relations:

- **Query planner:**
  - To pick good implementation for each query.
  - Evaluation criteria: Lowest cost (measured by a heuristic cost estimation function).
  - Done by the RelC Compiler:
    - From every edge from node “v1” to node “v2” in a decomposition “d” → “c(v1, v2)”: estimation of the number of instances of “v1” and “v2”.
    - “c” can be given by the user / recorded as part of profiling run.
Querying and Updating Decomposed Relations:

- **Empty**: $dempty(d)$: Create an empty instance of decomposition “$d$”.
- **Insert**: $dinsert(d, t, d')$: Insert the tuple “$t$” into a decomposition instance “$d'$” of “$d$”.
- **Remove**: $dremove(d, s, d')$ remove tuples matching “$s$” from an instance $d'$ of decomposition $d$
  
  $→$ Remove any nodes and edges from $d'$ that form part of the representation of tuples that only match “$s$”.
- **Updates**:
  
  - Function $update(d, s, u, d')$: update tuples matching “$s$” using values from $u$ in an instance $d$ of the decomposition $d'$.
  $→$ Semantically: $update = remove + insert$

- **Cut notion**: A cut is a partition $(X, Y)$ of the nodes of “$d$” into nodes $y_A ∈ Y$ that can only be part of the representation of tuples matching “$t$” (that is $A → C$) and nodes $x_B ∈ X$ that may form part of the representation tuples that do not match “$t$” (That is Not($B → C$))
Querying and Updating Decomposed Relations:

- Cut notion:

```plaintext
remove( {ns:1, pid:23})
```
Auto-tuner:

• How to find the best decomposition for a given relation?

→ Auto-tuner:

• From a program written to the relational interface and attempts to infer the best possible decomposition for that program.
Experiments: Microbenchmarks

- A benchmark based on the Process Scheduler example.
- A graph benchmark.
- A cache benchmark based on a real system.
Experiments: Microbenchmarks

- A benchmark based on the Process Scheduler example.
- A graph benchmark.
- A cache benchmark based on a real system.
Graph benchmark

- Reads in directed weighted graph from a text file and measures the time to:
  - Construct the edge relation.
  - To perform forwards and backwards depth-first searches over the whole graph.
  - And to remove all the edge one by one.
- The Graph's edges:
  - Relation: \{src, dst, weight\}.
  - Functional dependency: src, dst \rightarrow weight.
- The Graph's nodes:
  - Relation: \{id\}
Graph benchmark

- The Auto-Tuner generate 84 decompositions:
Thank you for your attention